A Selective Protection Scheme for the Java Environment

D. Hagimont[†] *, S.Krakowiak*[‡] *, J. Mossière*[¥] *, X. Rousset de Pina*[¥]

Project SIRAC (IMAG-INRIA)

INRIA, 655 av. de l'Europe, 38330 Montbonnot Saint-Martin, France

Internet: Daniel.Hagimont@imag.fr

Abstract: This paper describes a protection scheme for the Java environment. This scheme allows a programmer to specify the protection rights assigned to a class loaded from a remote site with respect to the local files. Protection is expressed in a high-level notation that extends the interface definition of the protected class. Protection is enforced by stubs that are automatically generated from the specification and that are interposed in front of the protected classes, with no possibility of by-passing. The scheme makes extensive use of the ability provided by the Java environment to redefine the class loaders, and relies on the use of "hidden" (i.e. invisible to the programmer) capabilities, a device previously proposed by some of the authors. Preliminary experiments have shown the feasibility of this method.

The code of this system will be made freely available.

1 Introduction

Protection is a crucial aspect of distributed computing, in particular when users co-operate using shared objects or shared programs. The development of distributed applications over the Internet has enhanced the interest for protection mechanisms: the Internet connection should not be a trap-door for the local host.

Sun Microsystems has recently made available an environment called Java [Gosling 95], which allows the development of distributed applications through a C++ like language. The main innovation of Java is that it allows the management of mobile code. The code generated by the Java compiler is interpreted by the Java virtual machine, thus enabling code transfer between heterogeneous sites. In particular, several Web browsers that encapsulate a Java interpreter have been developed and are used for loading simple applications called *applets* which are pointed to by HTML pages.

This ability provides a potential entry door for Trojan horses, programs that perform a covert function by by-passing protection mechanisms. In order to deal with Trojan horses, Java's language is type-safe (it does not allow the use of virtual addresses, since the language is interpreted). However, this is not sufficient if applets can access any file in the local file system, even if only files from the current user are concerned. Therefore, all the applications that encapsulate a Java interpreter (e.g. the HotJava browser) implement very strict policies :

- they forbid access to any file
- they only grant access to files that are located in a directory pointed by an environment variable.

We believe that such policies are overly restrictive for Java applications since they either make some applications impossible to write, or force the user to manage the granting of access rights "by hand" through assignments to environment variables or file copying.

We propose an application of the "need to know" principle, that is to provide exactly the access rights on files required for the execution of each operation. Our implementation is based on a novel protection scheme called *hidden software capabilities* [Hagimont 96]. For each class loaded from a remote

---

[†] Institut National de Recherche en Informatique et Automatique (INRIA)

[‡] Université Joseph Fourier, Grenoble

[¥] Institut National Polytechnique de Grenoble

site, an extended Interface Definition Language (IDL) is used to describe the protected interface of the class. This interface includes the methods' signatures, enriched with protection statements that indicate when a capability should be passed with a file name parameter. A remote class is always isolated in a protection domain and file capabilities are given to this protection domain according to the protection interface of the class. The protection interface is used for the automatic generation of protection checks.

Our implementation of this protection scheme also provides a uniform naming for Java classes. Each class has a name which is independent from its location, be it local or remote (the location is managed separately). This name can be used as if the class were local, the runtime system managing remote class loading as required.

Therefore, the specifications of a class protection and of its location are completely hidden to the programmer, thus enhancing modularity and separation of concerns. Moreover, this scheme is very flexible since it allows the protection/location of a class to be modified without any modification on the classes that are using it.

In this paper, we present the security model we have integrated in a Java environment. The paper is structured as follows. In section 2, we provide an overview of the Java environment and we analyse its requirements regarding security. We present in section 3 hidden software capabilities, the protection model we have designed and adapted in order to manage file access rights in Java. Section 4 describes the implementation of the protection model in the Java environment. We provide our conclusions and perspectives in section 5.

2 The Java Environment

This section provides an introduction to the Java environment. After a rapid overview of the language (section 2.1), we describe in section 2.2 the Java and HotJava execution models. Finally, section 2.3 is devoted to security and presents the requirements we set on our protection scheme.

2.1 The Java Language

Java is a C++ like object-oriented programming language that provides the features of the most popular OO languages: encapsulation, inheritance, polymorphism and dynamism. We discuss the last two features, which are important for the rest of the paper.

Polymorphism refers to the ability to define *Interfaces* (or types) separately from classes. An interface is a definition of the signatures of the methods of a class which is independent from any implementation. An interface can therefore be implemented by many classes, and it is possible to declare a variable whose type is an interface and which can reference objects from different classes that implement the same interface.

Dynamism means here the ability to determine at run-time the class of an object and its operations. This feature is very important because Java allows classes to be dynamically loaded and class names in classes to be dynamically resolved. A variable whose type is an interface can therefore be assigned an instance whose class has been dynamically loaded. Java delays the binding to the code of the methods until invocation time, thus allowing the use of dynamically loaded classes.

Another very important feature of Java is code mobility. Java allows classes to be dynamically loaded from remote nodes. Such mobility of code requires code portability and security enforcement.

Code portability is provided by interpretation of byte code. The *java* compiler does not generate machine code, but a code which is common to (and independent of) any type of hardware and which is interpreted by the runtime of the language.

Security is mainly enforced by the safety of the Java language. The Java language does not allow direct access to the address space of the program (objects are not manipulated through pointers, but through language level references) and it is strongly typed, implementing rigorous compile and runtime checks. In particular, for classes which are dynamically loaded, the runtime checks that the loaded code is actually byte code, and it verifies methods conformity between the caller and the callee at invocation time. Java also allows security policies to be implemented at a higher level (see section 2.3 for more details).

2.2 The Java and HotJava Execution Models

A Java application is built from Java sources and already compiled Java classes. The result is a set of Java classes that are stored in Unix files, one class per file (the file is named with the name of the class).

All the classes may reference other classes, i.e. use the names of other classes. At compile time, the compiler checks that the referenced classes exist and are conform to the use which is made of them. Usually, a class that represents the application and that contains the *main()* method is used to start the application.

In a Unix environment, a Java application executes as a Unix process. This process executes the Java interpreter which interprets Java classes. Classes in Java are dynamically loaded, and names are dynamically bound or resolved (name resolution is done at first use). This means that the Java virtual machine keeps track of already loaded classes and loads on demand not yet loaded classes from the Unix files that contain them. Class loading and binding are done by a component of the Java environment called the class loader.

In the current version of Java, there are no persistent objects. Objects are all created at run time by instantiating classes. The only form of persistence is provided through Unix files. Special Java classes are used to access Unix files: such a class may be instantiated and the instance bound to a file; the execution of a method in the instance performs an access to the file. The Java environment includes a set of classes that provide threads, I/O, networking, windowing and many other features.

HotJava is a Web browser which allows remote classes to be dynamically loaded with HTML documents. A special tag in the HTML language allows the insertion of a link to a Java application (also called *applet*) in a HTML document. When the document is browsed by HotJava, the applet is loaded by HotJava and the code of the applet is executed.

In order to allow applets to be loaded from a remote node, HotJava defines a new class loader that supplements the default Java class loader. While the default class loader only loads classes from the local file system, HotJava's class loader can load a class from a remote node.

The interface between HotJava and the applets is very simple. An applet must have been created as a subclass of class *Applet*, which defines a set of methods that are called by HotJava when some events that concern the applet occur. An applet programmer overrides these methods to define the specific behavior of its applet. Examples of methods/events are *start()* called when the applet's document is visited or *stop()* called when the applet's document is no longer visible on the screen.

In the next section, we study the security requirements of Java applications and how HotJava deals with these requirements.

2.3 Java Security and Requirements

The ability to load remote classes provides a possible entry for Trojan horses. The code loaded on one machine may have been written by a ill-disposed user who aims at corrupting local files or channeling confidential information to a remote node. Therefore, access to the local file system must be controlled by a protection scheme.

However, the protection policies that were proposed for Java applications are very restrictive regarding persistent objects, i.e. Unix files. In HotJava, access is confined to files in three directories referenced by three variables: *ReadACL* for read access, *WriteACL* for write access and *ExecACL* for execution. These variables can be assigned at most once. When HotJava is run, it first initializes these variables from some Unix environment variables, therefore confining remotely loaded classes to files in these directories. Obviously, this scheme lacks flexibility. It requires, when access rights must be modified, either to modify the environment variables and to restart HotJava or to copy files for which access is granted in the above directories.

We believe it is possible to provide a much more flexible file access control scheme that implements the "need to know" policy. The idea is to give exactly the access rights required for the execution of an operation. If a Java class loads a remote class and invokes one of its methods which takes as a parameter a file pathname for which read access is required, then we want to only provide read access on this file to the loaded class.

The second idea we propose is to hide distribution and protection to the programmer, implementing the so-called transparency.

As explained in the previous section, HotJava defines a new class loader (different from the standard Java class loader that only loads classes from the local file system), which can load remote applets. In HotJava, when an applet must be loaded, an explicit call is made to the new class loader.

A Java programmer who wants to use a remote class *TheClass* must define or reuse a class loader, and explicitly invoke that class loader for loading the class. Moreover, this programmer must cast (force the type of) the loaded class *TheClass* with an interface, in order to associate a type with it and to be able to call a method of *TheClass*. This code sequence is the following:

```
Class c = loader.loadClass("http://www.imag.fr/javacode/TheClass");
Object o = c.newInstance();
TheClass_itf o1 = (TheClass_itf)o;
o1.method(filename);
```

The first line is the loading of the class *TheClass* from a remote node. The result is a class with no specific interface. Then, an object is created as an instance of this class. This object also does not have any interface. *TheClass_itf* is an interface which describes some methods signature. The cast allows the method in the instance to be called. In the case of HotJava, a remotely loaded class is always cast with the interface of the class *Applet*.

We wish to allow the programmer to write the following, much simpler, code sequence:

```
TheClass o1;
o1.method(file_name);
```

The class loader is then responsible for class loading, the class *TheClass* being local or distant. We would like the specification of the location of the class *TheClass* to be hidden to the programmer, i.e. to be managed separatly. In the same vein as for location, we would like the specification of protection to be separated from the code of the application. This separate specification of protection would indicate, in case the class *TheClass* is remote, whether access rights for the file *filename* should be granted.

We explain in the next section how this function can be provided through a capability-based protection model.

3 Hidden Software Capabilities

In the previous section, we have described some limitations of the security management in Java. We now describe the improvements we are introducing in the Java environment; these improvements are adapted from a general capability based protection scheme, called hidden capabilities, which we have implemented in the context of a Distributed Shared Memory System [Hagimont 96] and are planning to experiment in the Orbix [Iona 94] implementation of the Corba Architecture[OMG 91]. The remainder of the section is organized as follows: in 3.1 is defined the general protection model and in 3.2 its application to Java.

3.1 Overview of Hidden Software Capabilities

The protection model is based on the concepts of capability, protection domain and domain capability.

A capability is a token that identifies an object and contains access rights that define the allowed operations on that object. In order to access an object, the agent on behalf of which the operation is performed must own a capability on that object. Capabilities can be copied and exchanged between cooperating agents.

Protection domains are protection environments characterized by the set of capabilities they contain. At a given time, an agent executes in exactly one domain and can only perform the operations authorized by the capabilities included in that domain.

Domain capabilities are entry-points which allow an agent to change protection domain in a controlled fashion. A domain capability is associated with an interface which determines the operation which allows the domain change and the operation parameters that are to be transferred between the protection domains. A protection domain crossing may involve capability transfers between protection domains.

The main idea of Hidden Software Capabilities is to hide capabilities from the programmer when they are used for access rights control, protection domain crossing and access rights transfer. Indeed, in all the proposed approaches (e.g. [Tanenbaum86, Chase 94]), capabilities are made available at the

programming language level through capability variables that are used explicitly for protection management.

With Hidden Software Capabilities, the application code is independent from protection. Capabilities checks are performed transparently by the underlying system (a user program does not have to submit a capability as a protected password) and capabilities exchanges between untrusted agents in different protection domains are specified separately from the application code in some configuration objects. Thus, Hidden Software Capabilities allow a single piece of a code to be executed with different protection configurations and ensure both economy and separation of concerns.

Managing Access to Protected Objects

Protected objects are supposed to be shared and persistent. Very often in such a case, the first time an application tries to access an object, it traps into the **object manager** which is supposed to bind the object to the language level reference (a unique object identifier or a pointer) used to access the object.

Thus, this object manager can check if a capability (for accessing the object) exists in the current protection domain. If this is the case, the object manager completes the binding and resumes the execution. If not, the object manager delivers a protection violation exception to the application.

Therefore, the application code only includes language references and does not have to include capability operations.

Managing Protection Domain Crossings

We want to allow protection domain crossing by the implicit use of a domain capability. Therefore, in our model, a protection domain crossing capability is associated with an operation and an object, and it is used only when the operation on the object cannot be performed locally.

When an application in one protection domain traps in the object manager (for an object binding) and if the protection domain does not contain a capability for the target operation, but contains a domain capability (associated with the target operation), then the same access will trigger a domain crossing, allowing the operation to be executed in the remote domain (specified by the capability).

Managing Capability Transfers

One of the main problems is then to allow capability parameters to be exchanged between protection domains while keeping code independent from protection. From now on, we assume that an operation on an object has a procedural form.

As explained above, capability transfers are specified in a configuration object. This object is expressed using an extended Interface Definition Language (IDL), which includes a description of capabilities exchange. The key feature we are using here is that an interface is defined separately from the code of the application. Just like an interface is associated with an entry point in a remote server, a protected interface can be associated with a domain capability which is an entry point in the target protection domain. This protected interface describes the behavior of this entry point according to protection.

Illustration

We illustrate the protected interfaces through the use of a *Print* procedure, exported by a printing server that allows a client to print out a file. We assume that the client trusts the printing server.

The *Print* procedure needs to execute in supervisor mode in order to access the device driver associated with the printer, but this privilege is not granted to the clients. Therefore, the *Print* procedure executes in a separate protection domain, and a domain capability is delivered (when the printer is installed) to the clients, providing them with a means to invoke the protected procedure. When a client wants to print out a file, the *Print* procedure needs to get read rights on this file; therefore the user will, at invocation time, pass a read-only capability on the file to the printer protection domain. This capability allows the *Print* procedure to read the contents of the file. Then, the protected interface (described with our extended IDL) of the *Print* entry-point is the following,

procedure Print (File file_id **capa_read**);

the *capa_read* key-word indicating that a read access right must be passed with the file object reference (file-id). This protected interface is defined separately from the application code which only

contains the *Print* procedure definition on the server side and the *Print* procedure invocation on the client side.

3.2 Application to Java

Our goal is to extend the functionalities provided by the Java runtime environment. We want to allow an application running on Java to dynamically load and execute any remote class whose URL is known to it. Furthermore, we want to grant these remote classes a selective controlled access to the local file system when they are processed locally. Such functionalities are provided by some Web browsers (e.g. Netscape and HotJava) but they do not provide selective control access to the file system.

We use the mechanisms defined by our protection model to define a protection policy which guarantees that the new functionalities do not open security trapdoors in Java. The protection policy specifies:

- Protection domain management

  A different protection domain is associated with each different remote class. Initially, this protection domain is empty, and no file access right is granted.

  Local code, the code loaded from the local file system that may invoke remote classes, executes in a protection domain that contains all the capabilities that correspond to the access rights the current user owns on the local file system. These capabilities may be transferred to protection domains associated with remote classes when these classes are invoked.
- Protection domain crossing

  Since our intent is only to control access to files, we assume that a local class has always a domain capability on those remote classes whose URLs are known to it. These capabilities are used to find out the capabilities which have to be transferred from the calling domain to the domain which processes the invocation.
- The transfer of capabilities to a protection domain associated with a remote class is described using an extended IDL, which specifies the protected interface of the class. Actually, this specification of the interface of a class already exists in Java. This interface is enriched with key-words (capa_read or capa_write) that indicate capability transfers for string parameters that correspond to the pathname of a file.

This protection policy does not use all the mechanism provided by the model. This makes possible many simplifications in the Java implementation of the model that we describe in the next section. This implementation follows the ordinary use of Java which only allows the loading of isolated applications, i.e. a remotely loaded class does not contain any external reference to another class.

4 Implementing Hidden Capabilities in the Java Environment

As explained above, the main goal of our protection scheme is:

- to describe the protection rights assigned to a class in a high-level description called a *protected interface*,
- to isolate remotely loaded classes in separate protection domains,
- to control access to local files from these protection domains,
- to grant capabilities to a protection domain according to the protected interface of the class it contains.

In Java, there is no notion of address space or of protection domain. Java provides two basic mechanisms that allow protection domains to be managed at a higher level, namely the *ClassLoader* and *SecurityManager* classes.

A class loader implements class loading and binding. The overriding of the default class loader provides a means to control how class names should be resolved. This Java facility is described in section 4.1.

A security manager is a class which implements security checks regarding the use of system classes. Java allows the security manager to be overridden, which enables a user to define his own protection policy on system objects. This second facility is described in section 4.2.

In section 4.3, we explain how dass loaders can be used to manage protection domains and to control access rights to files. We then describe the implementation of capability transfers in section 4.4

and the way class naming can be managed in a uniform way (the referenced class being local or distant) in section 4.5. Finally, section 4.6 analyzes how this protection scheme responds to potential trap doors. The skeleton of our class loader, the key element of our protection system, is described in Appendix A.

4.1. The Class Loader Class

In Java, class loaders have two functions.

- *Class loading*: classes are stored in standard files and are dynamically loaded by the Java virtual machine. A class may be loaded from a local or a remote file.
- *Class binding*: a class which uses another class contains the name of the latter. At first use, this name is translated into a reference to the class object it represents. In Java, this translation operation is called binding or the name is said to be resolved. Binding a class requires its loading.

While the Java runtime provides its own system class loader which is used to load classes from the local file system, it is possible to define a new class loader as a subclass of the class *ClassLoader*.

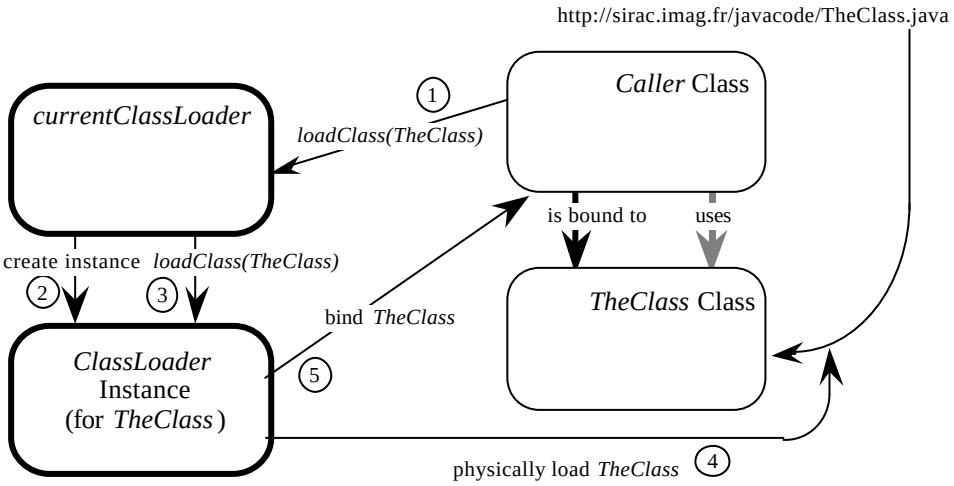
In particular, this is the way an application can load a remote class. Instead of reading the class code from a local file, this class code can be read from a remote file available, for example, on a Web server. When a class is loaded from a file (local or remote), a method (*defineClass()*) inherited from the *ClassLoader* class is used to generate a Java class (in byte code form) from the bytes that were read.

In a class loader, the *loadClass()* method is used to resolve a class name. This method should check whether the requested class is already loaded, load it if necessary, and return the class object. Therefore, a subclass of *ClassLoader* overrides this method with a method that allows remote class loading.

An important characteristic of class loaders is that they are always responsible for resolving the class names in the classes they have loaded. If a class is loaded by a class loader *CL*, then the binding of a class name in the loaded class will be performed by *CL* and no other loader. This property is essential for security since it allows security checks to be inserted in *CL*. All the class names in a class are resolved lazily (at first use).

In case a class loader does not want to manage the loading of a particular class (usually for system classes), the method *findSystemClass()* is inherited from *ClassLoader* and requests the system to resolve a reference.

A class loader manages a cache of all the names that have been bound. In order to use a class loader class, a program must instantiate this class and invoke the method *loadClass()* on the resulting instance. The binding of the name of the loaded class will be inserted in the cache. The names that this class contains will be bound lazily by the same loader and registered in the same cache. A program may instantiate this class loader several times, therefore managing several binding caches.



This figure describes the loading of class *TheClass* after invocation by the *Caller* class. At first call, the *loadClass* method of the *currentClassLoader* is invoked (1). The class loader then creates (2) a new instance of *ClassLoader*

(specialized for *TheClass*), and invokes *loadClass* on it (3). The new class loader physically loads (4) a copy of *TheClass*, and sets up the binding (5) between *Caller* and *TheClass*.

Figure 1. Loading and binding a class

4.2. The Security Manager Class

Java provides a means to associate security checks with the use of system classes such as threads, files, etc. This function is provided by the *SecurityManager* class, which defines a wide set of checking methods.

Since we are interested in file protection in the context of this work, we describe the checking methods associated with files. The security manager defines two methods *checkRead(String filename)* and *checkWrite(String filename)* that can check whether a file access is authorized.

Initially, the security manager is empty, i.e. it does not implement any check. A user program can define a new security manager as a subclass of the *SecurityManager* class, instantiate the new security manager and install it in the system using the *setSecurityManager()* system call.

The security of the security manager is based on the fact that *setSecurityManager()* can be called at most once. Therefore, if an application has installed a security manager, a remotely loaded class cannot install a new one.

Finally, a last important feature is that a security manager knows about the class loader used to bind the name of the system class it protects. This means that in the *checkRead()* method of a security manager, it is possible to obtain (with the *currentClassLoader()* method inherited from the *SecurityManager* class) the object reference of the class loader instance used to bind the class name *File* in the calling class. For example, if a remote class *C* is loaded with a class loader instance *CLI* and if *C* uses the class *File*, then the name *File* is resolved by *CLI* and when a file is read, the *checkRead()* method is invoked. The *checkRead()* method can get the identity of the loader user (*CLI*) and use it for its checks. The next section explains how this feature is used to implement protection domains.

4.3. Protection Domains and File Access Control

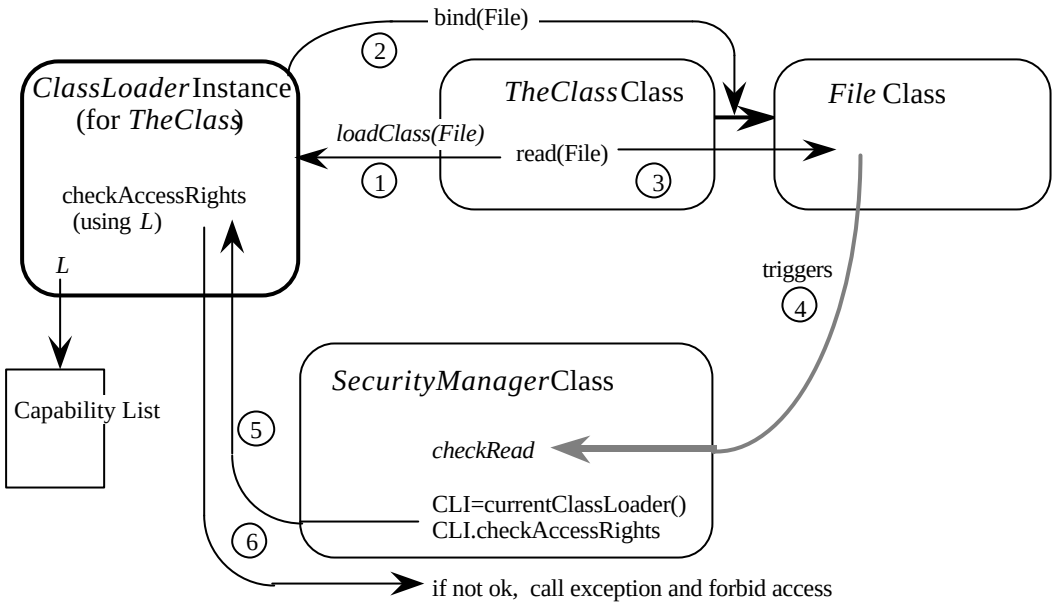
The basic idea for managing file access control is to implement and install a new security manager which checks file access permissions. However, when executing the *checkRead()* or *checkWrite()* methods in a security manager, the only way to authentify the class (which may be a remotely loaded class) which is performing a file access is to get its class loader. Therefore, for the management of our protection domains, we have to use a different class loader per protection domain, i.e. one for each remote class we load.

When a remote class needs to be loaded, we create a new instance of our class loader. This class loader instance (class loader for short) is used for loading the class and it corresponds to a new protection domain. As explained just above, the checking primitives of the security manager will be able to get the reference of this class loader in order to implement access control.

Access control consists in checking if a capability exists in the protection domain. Since the class loader corresponds (one to one) to the protection domain, we store in the class loader instance (in an instance variable *L*) the reference of the capability list object which describes the protection domain, thus allowing the checking primitive to consult the capability list. The class loader class exports a method (*checkAccessRights()*)that consults the capability list in order to verify read or write access permission for a given file name.

This capability list is created when the class loader instance is created and it is initialized empty (with no rights). Section 4.1.4 explains how capability transfers are managed.

This implementation of protection domains is illustrated on Figure 2.



This figure decribes the operation of the protection mechanism. The *read(File)* instruction is executed in the program of class *TheClass*. This causes the class File to be loaded (1) and bound (2), like *theClass* was loaded and bound in Fig. 1, except that the current instance of the class loader is used. Read access to *File* causes the *checkRead* method to be triggered (4) in the Security Manager. It first gets the reference of the current loader, then invokes (5) *checkAccessRights* in that class loader. This method determines whether the access is allowed, using the capability list of *TheClass*. It returns (6) either a permission to do the access or a protection violation exception.

Figure 2. Protection in the Java environment

4.4. Transferring Capabilities between Protection Domains

As explained in section 3.2, each remotely loaded class is managed in a separate protection domain, while local code (from the local file system) executes in a protection domain that has access to every file. The problem is therefore to create capabilities and install them in protection domains when remote classes are invoked, according to the interface defined with our extended IDL.

Our implementation consists in the installation of a stub class “in front of” the loaded class. This stub class implements the same interface as the class, except that, when the IDL specification indicates that a file capability must be passed along with a parameter, the stub adds the capability in the capability list associated with the loaded class. The stub does not need to check if this capability exists in the calling protection domain since the calling domain is always associated with local code and therefore contains capabilities for every file.

When the stub class is instantiated, it must instantiate the loaded class and store the obtained reference in the instance of the stub class (in an instance variable *O*). When a method is invoked on an instance of the stub class, the stub class invokes the same method on *O* with the capability parameter passing mentioned above.

Since the stub class implements capability passing, it needs to have a reference to the capability list of the current protection domain. A variable (a class variable *L*) in the stub class points to the capability list. This variable is initialized by the class loader when it installs the stub in front of the class it loads.

To illustrate this mechanism, this figure shows a protected interface and the associated stub class.

```
interface TheClass_itf {
  public void m(String file capa_read);
}
```

```
public class TheClass_stub  {
    TheClass O;
    public TheClass_stub() {
      O = new TheClass();
    }
    void m (String file ) {
      L.AddCapa (file, read);
      O.m (file);
    }
}
```

The stub class (*TheClass_stub*) defines the same methods as the protected interface (*TheClass_itf*). *TheClass_itf* defines that a read capability should be transfered with the *file* parameter of method *m*. Therefore, method *m* in *TheClass_stub* adds a capability in the capability list and forwards the invocation to the object it represents (*O*). *TheClass_stub*() is the constructor of the *TheClass_stub* class; it is invoked when a new instance of *TheClass_stub* is created; it creates a new instance of the class the stub represents (*TheClass*).

In order to be able to compile the stub class, we must provide a dummy class called *TheClass*. This class is also generated from the protected interface with empty method bodies, but it is only used at compile time. At run time, this class is ignored and the remote class is loaded instead.

When the class loader loads the remote class (*TheClass*), the loader loads both classes *TheClass* (which is remote) and *TheClass_stub* (which is local) and binds the class name *TheClass* in the calling class to the class *TheClass_stub*.

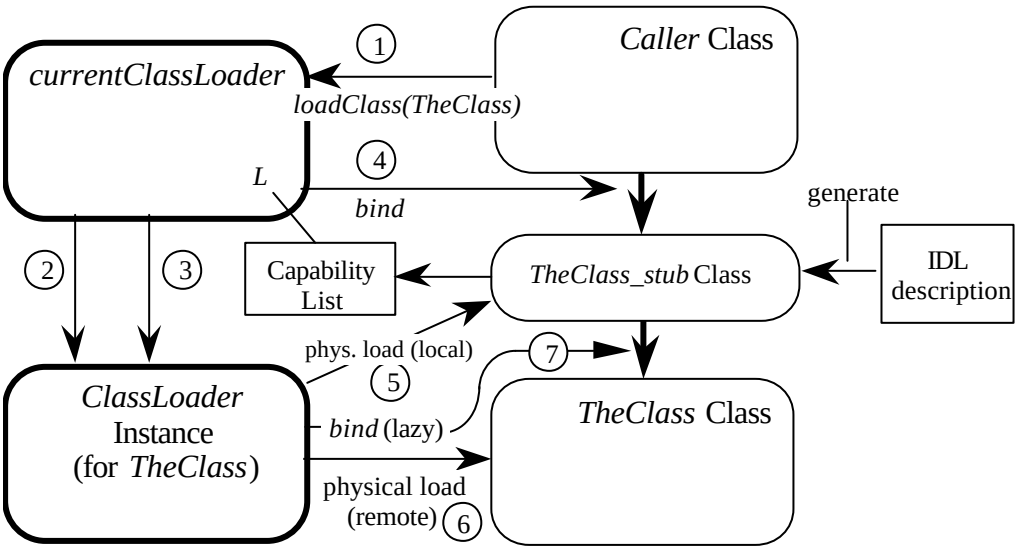
Figure 3 illustrates the management of stub classes for capability checks.

TheClass_stub forwards all the method invocations to the loaded class (*TheClass*). It transmits capabilities from the caller to the callee by inserting the capabilities in the capability list which is shared with the class loader of *TheClass* .

Note that the class loader of the calling class and the class loader of the loaded class are not the same. The loader of the calling class (*currentClassLoader*) has the binding ("TheClass", *TheClass_stub*) in its cache while the loader of the loaded class (*ClassLoader* instance for *TheClass*) has the couple ("TheClass", *TheClass*) in its cache.

4.5. Remote Class Naming

As explained in section 2.3, we want to provide a uniform naming for classes, thus avoiding programmers to have to cast loaded class with an interface for method invocation.



This figure is a refinement of Figure 1: it describes how stub classes are used to shield access to a newly loaded class. Steps 1 to 4 (up to the instantiation of the new class loader) are the same as in Fig. 1. Instead of loading *TheClass*, the new class loader actually loads and binds a (local) stub class *TheClass_stub*, that has been generated from the IDL

protected interface of *TheClass*. At the first call of a method of *TheClass*, the class loader loads that class (6) from the network, and sets up the binding (7) between *TheClass_stub* and *TheClass*.

Figure 3. Using stub classes for protection

The principle is to allow the programmer to write the following sequence whether the class is local or remote

```
TheClass o1 = new TheClass;
o1.method();
```

Our implementation is based on our specific class loader. Our class loader is not only used to load remote classes, but also to load the initial class that contains a *main()* method.

When the class loader needs to resolve a class name *TheClass* which is not yet in its cache, it checks if a stub class *TheClass_stub* exists for it in the current directory. If this file exists, the class loader reads a file *TheClass_loc* that contains the location (URL) of the remote class (http://www.imag.fr/javacode/TheClass in our case). The loader can then install the classes *TheClass* and *TheClass_stub* as explained above. If the loader do not find the stub class, then it tries to resolve the name as a local class.

In order to be able to compile the caller class, we use the same trick as for compiling the stub: an empty class is generated from the protected interface and used to compile the caller class.

The advantages of this uniform naming scheme are the following:

- it is much simpler to program,
- it is possible to modify the protected interface, to regenerate the stub class, but also to change the location of a remote class without modifying the application. All the IDL specification and location files may be grouped and shared between multiple applications that share the same administration tool.

Since the class loader is the key element of our implementation, we provide the outline of its implementation in Appendix A.

4.6. Analysis of potential trap-doors

The security of our system relies on the integrity of the lists of capacities and on the use of a special class loader to load remote classes. We analyse in this section how these two requirements are satisfied.

Firstly, a remote user cannot access a capability list. In order to access such a list, the loaded class would need to obtain a reference to the capability list object. This reference is never given to any loaded class, so the only possibility would be for the loaded class to forge this reference. Fortunately, the Java language is type-safe and there is no means to forge such a reference.

Secondly, the reliability of our protection scheme is mainly based on our class loader: it filters all class name resolutions, authorizing only the bindings that cannot induce protection violations. In the current implementation, any binding to the classes *OurClassLoader* or *ClassLoader* is rejected by the loader.

- *OurClassLoader*. A remotely loaded class that would be allowed to instantiate our class loader could initialize the capability list as needed to access any file. Therefore, our class loader forbids the binding of the class name *OurClassLoader* from a remotely loaded class. This restriction does not prevent a remote class to load another remote class, but this loading will be performed by our class loader. This facility is not yet implemented and is the subject of current work (section 6).
- *ClassLoader*. A remote class that inherits the class *ClassLoader* (this remote class is actually a kind of class loader) is allowed to invoke the *defineClass()* method inherited from *ClassLoader*. In this way, this remote class becomes the class loader of the created class and can enable any binding. Consequently, we forbid any remote class loader to be loaded.

Finally, our system is built of Java instances running on top of an operating system. Any security hole in these two components will be also present in our implementation. Moreover, file accesses are always checked in accordance with operating system rules and our protection scheme can only introduce restrictions to these rules.

5 Related Work

A few works have been published about protection schemes within the Java environment due to the fact than Java is relatively recent. However some works exist allowing controlled execution of remotely loaded code. Mainly two environments HotJava [Sun 95b] and Netscape [] allow stand alone Java applications running on a site to use such an untrusted code called applet. In both environments an applet is explicitely called. HotJava and Netscape use the basic mechanism provided by Java [Yellin 95] to control the applet execution.

The protection policy enforced by Netscape is very strict and for exemple a remotely loaded applet can not touch local files [Weiss 96]. This may made applet hard to design and program.

The protection policy enforced by HotJava [Sun 95a] allows read and write access to file contained in directories included respectively in read or write directory list pointed to by environment variables. So, in this scheme all applets share the same access privileges: if one applet can read or write a file all applets can do the same.

[Weiss 96]

[Yellin 95]

[Sun 95]

TeleScript as reported in [Weiss 96]

6 Conclusion and Perspectives

We have presented the design and implementation of a protection scheme for the Java environment. Specifically, we have extended the Java runtime environment with a mechanism that enables an application (a) to specify, in a flexible way, access rights on local files for any Java class available on the Internet; and (b) to transparently load and securely execute any such class.

Flexibility means that a remote class can be granted selective access to the files owned by the user of the application. Transparency means that the invocation of a method is performed by exactly the same code sequence for an instance of a local class and for an instance of a class loaded from a remote site. Security means that a remote class cannot corrupt an application that uses it; more precisely, it cannot get more rights on a file than those that were granted to it.

These facilities are provided by:

- a small number of Java classes, mainly the loader class, the security class and the class associated with capability management;
- an extended IDL which allows to specify access rights associated with file parameters, and its compiler, which generates the stub used each time a remote class is invoked;
- and finally, a uniform class naming scheme which allows remote classes (delivered by Web servers) to be used with simple local names.

The current implementation of the protection scheme has two limitations. First, all locally defined classes run in the same protection domain, with no capability list. By default, local classes inherit the rights of the application's user. Second, a class that is loaded from a remote site may only include references to the Java runtime class, and may only use files which it has either created or received as a parameter.

As a continuation of this work, we plan to investigate the following issues.

1) The first step would be to release partly the classes which are loaded from remote nodes from the constraint to be self-contained. Indeed, it does not seem very difficult to allow a remote class to contain references to other classes. Two problems have to be solved when a class loaded from a remote node calls another class: (a) how to locate the callee class (which may or not be co-located with the calling class); and (b) what access rights should be attached to the file parameters.

2) Developing really distributed applications with Java implies the integration of persistence in the language. Work is in progress towards that goal (e.g. [Atkinson 96]). In our current work, persistence is only achieved through the use of files. We intend to examine a possible extension of our protection environment to cover persitent Java objects.

Despite its current limitations, we think that this work is a promising step toward the development of secure distributed applications in Java.

Acknowledgments.

P. Dechamboux, J. Han, A. Knaff, E. Pérez-Cortés and F. Saunier have contributed to the design of Hidden Software Capabilities.

This work was partially supported by CNET (France Télécom) under contract 941B160.

Bibliography

[Atkinson 96] M. P. Atkinson, M. J. Jordan, L. Daynès, and S. Spence, Design Issues for Persistent Java: a type-safe, object-oriented, orthogonally persistent system, to appear in *Proc. of 7th Workshop on Persistent Object Systems* (*POS-7*), 1996.

[Chase 94] J. S. Chase, H. M. Levy, M. J. Feeley, E. D. Lazowska. Sharing and Protection in a Single-Address-Space Operating System, *ACM Transactions on Computer Systems*, 12(4), pp. 271-307, November 1994.

[Gosling 95] J. Gosling and H. McGilton, The Java Language Environment: a White Paper, Sun Microsystems Inc., 1995http://java.sun.com/whitePaper/java-whitepaper-1.html,

[Hagimont 96] D. Hagimont, J. Mossière, X. Rousset de Pina, F. Saunier. Hidden Software Capabilities, *16th International Conference on Distributed Computing Systems*, May 96

[Iona 94] Iona Technologies. Orbix distributed object technology - programmer's guide, Version 1.2, February 94

[OMG 91] OMG. The Common Object Request Broker: Architecture and Specification, OMG Document Number 91.12.1, Revision 1.1, December 1991.

[Sun 95a] Sun Microsystems, Inc. "HotJava: The Security Story", 1995
URL: http://www.javasoft.com/1.0alpha3/doc/security/security.html.

[Sun 95b] Sun Microsystems, Inc. "The HotJava Browser: A Whitepaper", 1995
URL: http://www.javasoft.com/1.0alpha3/doc/overview/hotjava/index.html

[Tanenbaum 86] A. S. Tanenbaum, S. J. Mullender, R. Van Renesse. Using Sparse Capabilities in a Distributed Operating System, *Proc. of the Sixth IEEE International Conference on Distributed Computing Systems*, pp. 558-563, 1986.

[Weiss 96] Weiss, Michael, Andy Johnson, and Joe Kiniry. "Security Features of Java and HotJava". *OSF Research Institute Technical Report*, Feb. 1996.

[Yellin 95] Yellin, Frank. "Low Level Security in Java". In World Wide Web Journal: *Fourth International World Wide Web Conference Proceedings*, December 11-14 1995, Boston, MA, pages 369-379. URL: http://java.sun.com/sfaq/verifier.html

Appendix A. The skeleton of the ClassLoader

```
public class OurClassLoader extends ClassLoader {

    // the cache for name binding
    Hashtable cache = new Hashtable();
```

```
    // the capability list of the associated protection domain
    CapaList Clist;

    // constructor of the class: list is the capability list
    // associated with the protection domain
    public OurClassLoader (CapaList list) {
        Clist = l;
    }

    // local method that loads a class from a URL
    private byte loadClassFromURL (String name)[] {
        // connect to the web server and read the file
        // that contains the class
    }

    // local method that loads a class from a local file
    private byte loadClassFromFile (String name)[] {
        // read the class from a local file
    }

    // the entry point of the class loader used to resolve a class name
    public synchronized Class loadClass (String name) {

        // get the name of the class without a location prefix
        String shortname = ....

        // try to find the class in the cache
        Class theclass = cache.get(shortname);
        if (theclass != null) return theclass;

        // if it is a HTTP address, loads the class from the URL
        if (name.startsWith(“http”)) {                                  (1)
               byte data[] = loadClassFromURL(name);
               if (data == null) return null;
               theclass = defineClass(data);
               cache.put(shortname,theclass);
               return theclass;
        }

        // check if a class stub exists for the class
        //  a  stub  file  name is the name of the class concatenated with
“_stub”
        // if the stub does not exist, try to resolve the name
        // with a local class
        File stubfile = new File(name+”_stub”);

        if (!stubfile.exists()) {
               byte data[] = loadClassFromFile(name);                   (2)

               // if we can’t find this file, try to resolve the name
               // as a system class
               // else we got the class from the local file system
               if (data == null) {
                      theclass = findSystemClass(name);
                      return theclass;
               }

               theclass = defineClass(data);
               cache.put(shortname,theclass);
```

```
            return theclass;
        }

        // if a stub file exists ...                                    (3)
        // read the location of the remote class in the location file
        // a location file name is the name of the class concatenated
        // with "_loc"
        File locfile = new File(name+"_loc");
        String loc = locfile.readLine();

        // create a new capability list for the new protection domain
        // create a new class loader instance for loading the remote class
        CapaList list = new CapaList();
        OurClassLoader loader = new OurClassLoader(list);

        // load the remote class with the new class loader
        theclass = loader.loadClass(loc);

        // load the stub of the class
        Class thestub = loader.loadClass(name+"_stub");

        // initialize the capability list in the stub class
        // A cast is here required. We don't describe it here.

        cache.put(name,thestub);
        return thestub;
    }

    // method called by the security manager for protection checks
    public boolean checkAccessRight (String file, boolean mode) {
            return Clist.Granted(file,mode);
    }
}
```

In order to comment this Java class, let's consider the two following cases:

- *loadClass()* is called for a local class. Then, this case is dealt with on line (2), be this class a class in a local file or a system class.
- *loadClass()* is called for a remote class. Then, this case is dealt with on line (3). A stub file is found and a new instance of the class loader is created. The method *loadClass()* is invoked on this instance with the actual location of the class which is an URL. This latter invocation is dealt with on line (1).