Non-functional capability-based access control in the Java environment*

D. Hagimont[1], N. De Palma[2]

[1] INRIA Rhône-Alpes, 655 avenue de l'Europe, 38334 Saint-Ismier Cedex, France
Daniel.Hagimont@inrialpes.fr
[2] France Telecom R&D, 28 chemin du vieux chêne, BP98, 38243 Meylan Cedex, France
Noel.Depalma@rd.francetelecom.com

Abstract. This paper describes a capability-based access control mechanism implemented on a Java environment. In this scheme, access to objects is controlled by means of software capabilities that can be exchanged between mutually suspicious interacting applications. Each application defines the access control rules that must be enforced when interacting with other applications. The definition of access right is managed as a non-functional aspect in the sense it is completely separated from the application code, thus enforcing modularity and ease of expression. It is described in an extended Interface Definition Language (IDL) at the interface level. We have experimented with two prototypes that show how this access control mechanism can be efficiently implemented on a standard Java environment.

1 Introduction

With the development of the Internet, distributed applications often rely on mobile software components which can be downloaded from a remote location or moved to a server machine in order to perform a given task. Examples of such mobile components are Sun's applets [6] which may be downloaded in a Web browser or mobile agents systems [3] which allow computation units to migrate and visit Internet locations.

In this context, protection is a crucial aspect. It is a prerequisite to the deployment of applications distributed over the Internet: the Internet connection should not be a trap-door for the local host.

Java [6] is probably the best known runtime environment which provides facilities for the development of mobile applications. Java was specifically designed to provide features for managing portable and mobile objects. The code generated by the Java compiler is interpreted by the Java virtual machine, thus enabling code transfer between heterogeneous sites. From a protection point of view, the main advantage of Java is that the Java language is type-safe (the language is interpreted and does not allow the use of virtual addresses). Type-safety is the key-technique used to achieve

* This work is supported by the Réseau National des Technologies Logicielles (RNTL Arcad).

protection, but this is not sufficient to allow interacting applications to control access rights in a flexible way.

In this paper, we propose a protection scheme for managing access control in the Java environment. This protection model is based on software capabilities [8] and it allows mutually suspicious applications to dynamically exchange access rights according to their execution context. This protection scheme has the following advantages:

- Evolution: since it is based on capabilities, access rights can be dynamically exchanged between applications during execution.
- Mutual suspicion and decentralization: each application is responsible for the definition of its own protection policy, which is linked with code of the application. There's no need for registering protection policies in a third party protection subsystem.
- Modularity: the protection scheme enforces modularity since the definition of protection is totally separated from the application code. Access control is therefore managed as a non-functional aspect.

We describe the design of this access control model on top of the Java virtual machine and we describe two prototypes that show how it can be efficiently implemented.

The rest of the paper is structured as follows. Section 2 presents the access control model that we designed. Section 3 presents a first implementation which relies on proxies (or indirection objects). In section 4, a second implementation is described, which avoids any indirection object and therefore performs much better. Section 5 presents the results of our measurements, showing the cost of access control with both prototypes. After a brief description of the related works in section 6, we conclude in section 7.

2 Access Control Model

2.1 A Capability-Based Protection Model

The protection model we propose is based on software capabilities.

A capability is a token that identifies an object and contains access rights, i.e. the subset of the object's methods whose invocation is allowed. In order to access an object, an application must own a capability to that object with the required access rights. When an object is created, a capability is returned to the creator application, that usually contains all rights on the object. The capability can thus be used to access the object, but can also be copied and passed to another application, providing it with access rights on that object. When a capability is copied, the rights associated with the copy can be restricted, in order to limit the rights given to the receiving application.

Therefore, each application executes in a protection environment in which it is granted access to the objects it owns. This application can obtain additional access rights upon method invocation. When an object reference is passed as parameter of an

invocation, a capability on that object can be passed with the parameter in order to provide the receiving application enough access rights to use the reference.

In order to illustrate capability based protection, let us consider the example of a *Printer* object, exported by a print server, that allows a client to print a text (Fig. 1).

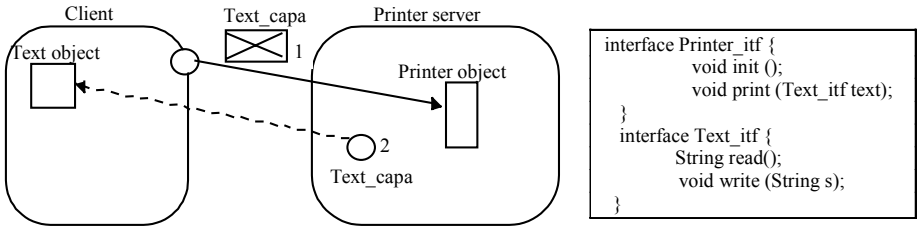

Fig. 1. Print server example

A capability on the *Printer* object is given to the client applications providing them with the right to print texts. When a client wants to print a text (*Text* object), the *Printer* object needs to get read rights for this text; therefore the client will pass, at invocation time (1), a read-only capability on the text (*text_capa*) to the callee application. This capability allows the *Printer* object to read the contents of the text (2).

2.2 Exchanging Capabilities

As explained in the previous section, software capabilities provide a model in which access right can be dynamically exchanged between applications. The issue is then to provide application programmers with a means for controlling rights exchanges with other applications.

One strong motivation for our protection model is modularity. Indeed, we don't want to provide extensions to the programming language that allow expressing capability parameter passing when another application's object is invoked. This would overload programs and make them much more difficult to maintain. Our goal is to separate protection definition from application code, i.e. to manage access control as a non functional aspect. To achieve this goal, our idea is to define capability exchanges between interacting applications using an interface definition language (IDL). Since an interface can be described independently from any implementation, describing capability exchanges at the level of the interface allows the protection definition to be clearly separated from the code of the application.

Therefore, an IDL has been defined that allows the application programmer to express the capabilities that should be transferred along with parameters in a method invocation. This IDL allows the definition of *views*. A view is an interface that includes the definition of an access control policy. A view is associated with a capability and describes:

- the methods that are authorized by the access rights associated with the capability,

– the capabilities that must be transferred between the caller and the callee along with the parameters of the methods authorized by the view. These transferred capabilities are expressed in terms of views.

Therefore, a capability is a structure which includes the identifier of the object, the access rights that the capability provides to its owner and the capability exchange policy which defines what capabilities must be passed along with parameters when the object is invoked. The access rights and the capability exchange policy are defined with a view. The definition of views is naturally recursive since it specifies the capabilities that should be transferred with parameters, this specification being in terms of view. For that reason, each protection view is given a name at definition time.

In the example of the Print server described above, two views may be associated with the *Text* class: a view *reader* that only grants access to the *read* method and a view *writer* that grants access to both methods *read* and *write*. For the *Printer* class, we define the view *user* which authorizes invocation of the method *print* which signature in the view is the following: void *print* (*Text_itf text* **pass *reader***). This signature expresses that a capability with the *reader* view must be passed to the callee along with the reference to a text object passed as parameter of *print.*

Such a protection policy, defined only on the callee side, would be sufficient if we were considering a client/server architecture where protection is only there to protect the server against its clients. Instead, we are considering an architecture where applications are mutually suspicious. Each application must have full control over the capabilities it exports to other applications (both the caller and the callee). Moreover, we want to ensure applications independence. More precisely, it is not possible for an application programmer to verify the protection policy defined by an application that exports a service since at programming time, the programmer may not yet know which applications it is going to interact with. This is particularly important when interacting applications are mobile, for instance in a mobile agents system [3].

For these reasons, each application can define its own view of the protection policy to apply when interacting with other applications. Therefore, two views are associated with a capability: the view of the caller application and the view of the callee application.

The view defined by the callee application *A* describes:

– The methods that are authorized.
– For each input parameter of a method (reference *R* received by *A*), the view describes the capabilities that are given by *A* when the reference *R* is used for method invocation. This view describes, from the callee point of view, the capabilities that the application accepts to export.
– For each output parameter of a method (reference *R* given by *A*), the view describes the capability returned with the reference *R*.

and similarly the view defined by the caller application *A* describes:

– For each input parameter of a method (reference *R* given by *A*), the view describes the capability given with the reference *R*.
– For each output parameter of a method (reference *R* received by *A*), the view describes the capabilities that are given by *A* when the reference *R* is used for method invocation. This view describes, from the caller point of view, the capabilities that the application accepts to export.

This symmetric scheme is the answer to mutual suspicion and applications independence. Both the caller and the callee specify their protection views for their objects. They are taken into account as follows.

When an application exports an object reference through the name server (similar to *rmiregistry*, but on a local machine), it defines the view associated with the reference, i.e. the capability which is exported for this exported reference. This way, the application also defines the capabilities that may be exported subsequently to an invocation of that object. When an application fetches the reference from the name server, it also defines the view associated (on its side) with the reference it obtained. This way, the application defines the capabilities that may be exported subsequently to an invocation of the object. Any invocation that derives from an invocation on that object will take into account the view definitions from both applications.

2.3 Example

In order to illustrate the expression scheme of our protection model, let's consider again the print server example. The Java interfaces of the *Printer* application were given in Fig 1. These interfaces are shared between the caller and the callee. In order to make the print service available to the clients, the *Printer* application exports an instance of class *Printer* through the name server. The *Printer* class is an implementation of the *Printer_itf* interface. On its side, the client application fetches this instance from the name server and can invoke a method (*init* or *print*) on this instance, using the *Printer_itf* interface. When the client wants to print a text, it invokes the method *print* and passes a reference to an instance of class *Text* which implements interface *Text_itf*. In the example, the definition of protection aims at avoiding the following protection problems:

- the printer doesn't want the client to invoke the *init* method on its printer objet (and to initialize the printer),
- the client doesn't want the printer to invoke the *write* method on its text object (and to modify the text of the client).

In our protection scheme, the client and the server will define the views presented in Fig. 2.

client	**Server**

view ***client*** implements Printer_itf {
  void init ();
  void print (Text_itf text *pass* ***reader***);
}
view ***reader*** implements Text_itf {
  String read();
  void **not** write (String s);
}
	view ***server*** implements Printer_itf {
	  void **not** init ();
	  void print (Text_itf text);
	}

Fig. 2. Protection views in the *Printer* application

Each application defines a set of views that define its protection policy. Each view « implements » the Java interface that corresponds to the type of the objects it pro-

tects. A **not** before a method name means that the method is not permitted. When an object reference is passed as parameter in a view, the programmer can specify the view to be passed with the reference using the key-word **pass**. If no view is specified, this means that no restriction is applied to this reference.

In this example, the print server defines the view *server* which prevents clients from invoking method *init*. No restriction is applied to the parameters of method *print*. The client defines the view *client* which says that, when a reference to a text is passed as a parameter of method *print*, the view *reader* must be passed, which prevents the print server from invoking method *write*. Notice that the client doesn't have any reason to prevent itself from invoking method *init*; this is a decision to be taken by the print server.

When the print server registers an instance of class *Printer* in the name server, it associates view *server* with it. When the client obtains this reference from the name server, it associates the view *client* with it. These two views and the nested ones (*reader*) define the access control policy of the two applications.

To sum up, each application defines its own protection policy independently from any other application or server and this policy specification is defined separately from the application implementation using views, thus enhancing modularity.

In the following, we describe two implementations of this model, respectively relying on proxies [9] and on bytecode injection [1].

3 Proxy-Based Implementation

For the implementation of this protection model, we used the fact that Java object references are almost capabilities (Java is strongly typed). Indeed, since Java is a safe language, it does not allow object references to be forged. This implies that if an object *O1* creates an object *O2*, object *O2* will not be accessible from other objects of the Java runtime, as long as *O1* does not explicitly export a reference to object *O2* towards other objects. The reference to *O2* can be exported (as a parameter) when an object invokes *O1* or when *O1* invokes another object. Therefore, as long as an application does not export a reference to one of its objects, these objects are protected against other applications. Thus, a Java object reference can be seen as a capability. However, they are all-or-nothing capabilities since it is not possible to restrict the set of methods that can be invoked using this reference. In order to implement our capabilities, we implemented a mechanism inspired from the notion of proxy [9], which allows access rights associated with a reference to be restricted.

Our implementation relies on the management of *filters* that are inserted between the caller and the callee. For each view defined by an application, a filter class is generated (by a pre-processor) and an instance of that class is inserted to protect the application.

When a reference to an object is passed as input parameter of a method call, instead of the real object, we pass a reference to an instance of the filter class generated from the view defined by the application providing the reference. This filter class implements all the methods declared in the interface of the view. It defines an instance vari-

able that points to the actual object and which is used to forward the authorized method calls. If a forbidden method is invoked on an instance of a filter class, then the method raises an exception. The reference to the filter instance, which is passed instead of the reference parameter, is inserted by the caller application. In fact, this filter instance is inserted by the filter used for the current invocation. In Fig. 3a, the invocation of *O2* performed by *App1* passes a reference to *O1* as parameter. The filter *F1(O2)*, which corresponds to the protection policy of *App1* for invocations of *02*, inserts filter *F1(O1)* before the parameter *O1*. Therefore, filters that are associated with reference parameters are installed by filters that are used upon method invocations.

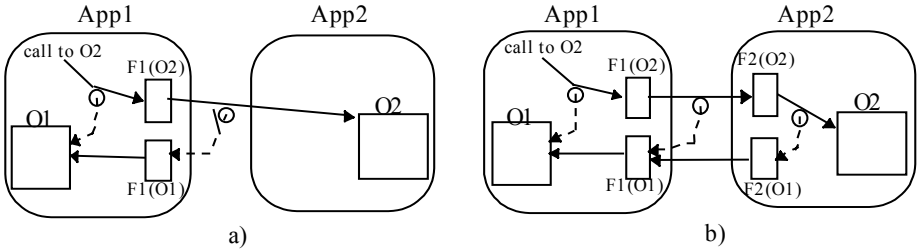

Fig. 3. Management of filters

Conversely, when a reference is received by an application, a reference to a filter instance is passed instead of the received parameter, which class is generated from the view specified by the application that receives the parameter. In Fig. 3b, the filter *F2(O2)*, which corresponds to the protection policy of *App2* for invocations of *02*, inserts filter *F2(O1)* before the received parameter. Therefore, two filter objects (*F1(O1)* et *F2(O1)*) are inserted between the caller and the callee for the parameter *O1* passed from *App1* to *App2*. These two filters behave as follows:

- *F1(O1)*: it enforces that only authorized methods can be invoked by *App2* and it inserts filters on the account of *App1* for the parameters of invocations on *O1* performed by *App2*.
- *F2(O1)*: it inserts filters on the account of *App2* for the parameters of invocations on *O1* performed by *App2*.

The code of the filter classes for the print server example is shown in Fig. 4.

client	**Server**

public class **reader** implements Text_itf {
  Text_itf obj;
  public reader(Text_itf o) {
  obj = o;
  }
  public String read() {
  return obj.read();
  }
  public void write(String s) {
  Exception !!!
  }
	class **server** implements Printer_itf {
	  Printer_itf obj;
	  public Printer_stub(Printer_itf o) {
	  obj = o;
	  }
	  public void init() {
	  Exception !!!
	  }
	  public void run(Text_itf text) {
	  obj.run(text);
	  }

```
public class client implements Printer_itf {
    Printer_itf obj;
    public Printer_stub(Printer_itf o) {
      obj = o;
    }
    public void init() {
      obj.init();
    }
    public void print(Text_itf text) {
      reader stub = new reader(text);
      obj.run(stub);
    }
}
```

Fig. 4. Code of the filter classes for the print server

In the next section, we present a second prototype implementation which avoid using indirection objects (filters) to manage capabilities.

4 Injection-Based Implementation

The general idea that we apply in this second implementation is to inject the protection related code (which was previously integrated in indirection objects) within the functional code of the application (Fig. 5). This way at runtime, we avoid two costly indirections. This code injection can be performed at compile time or at load time. We use a bytecode transformation tool (BCEL [1]) to inject in the application code, the code which implements capability management. In order to present this approach and for clarity in the rest of the paper, we will describe the transformed code in Java.

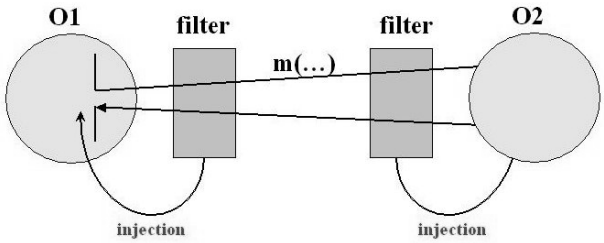

Fig. 5. Code injection technique

The protection code injection proceeds as follows:

- For each reference variable (field or local variable) pointing to a protected object, we have to inject the declaration of two new variables which represent the two views associated with the capability: the view specified by the object owner (the callee view) and the view specified by the owner of the capability (the caller view).

These two variables are view identifiers (integers). They are assigned when the reference variable is assigned (we inject the code that does it).

- For each protected method invocation using a reference, we add the callee view associated with this reference (the above integer which identifies this view), as parameter of the method. We inject at the beginning of the invoked method the code which checks whether this invocation is authorized in the definition of this callee view.
- When a reference to a protected object is passed as a method parameter, we inject the code which will initialize the caller view and callee view variables on the parameter receiver side. This initialization depends on the definitions of the views (caller and callee) associated with the capability used for the method invocation. This is further clarified on the printer example.

In the case of our printer example, when a client receives a reference to the printer (from a name server), it also receives the capability's caller and callee views (the integer identifiers). The method invocation to the printer takes two additional parameters, the callee view of the reference to the printer which allows checking access rights on the printer side, and the callee view for the text parameter which specifies the access rights that are granted to the printer on the text object (this callee view to pass with the text is defined in the view identified by *pr_caller* in the code in Fig. 6).

```
public class ClientMain {
    public static void main(String args[]) {
        Printer_itf pr; // ref to the printer
        short pr_caller; // printer caller view
        short pr_callee; // printer callee view
        Text text = new Text();
        // pr_callee : passed for checking
        //    the capability on the callee side
        // view_reader_id : the view passed with
        //   the text, depends on the (local)
        //   definition of pr_caller
        switch (pr_caller) {
            …
            pr.print(pr_callee, text, view_reader_id);
        }}}
```

Fig. 6. Code of the client after protection code injection

On the server side (Fig. 7), the printer checks that the access rights, associated with the view received as parameter, grant access to the method (for *init()* and *print()*).

In the *print()* method, the injected code initializes the view variables associated with the text received parameter. The callee view is received as parameter. The caller view for the text is defined in the callee view for the printer (depending on *pr_callee*).

```
class Printer implements Printer_itf {
    public void print (short pr_callee, Text_itf text, int text_view) {
        short text_caller; // text caller view
        short text_callee; // text callee view
        // checking the capability … based on
        // the (local) definition of pr_callee
        if (pr_callee != … ) {
            Exception !!!
        }
       // initialize the views for the text
        text_callee = text_view;
        switch (pr_callee) {
            …
            text_caller = …
        }
        text.read(text_callee);
        return;
    }}
```

Fig. 7. Code of the print server after protection code injection

With this code injection technique, it is possible to manage implement our capability-based access control mechanism without requiring (paying for) indirection objects.

5 Evaluation

In this section, we provide performance measurements for the two prototypes that we implemented. This performance evaluation shows the cost of capability management in our protection scheme. It also shows that this cost can be significantly reduced with the implementation based on code injection. For the second prototype, the implementation of the bytecode translator is in progress. The evaluation presented in this section is based on hand-written code, i.e. the code injection was performed at the application source level, applying our transformation patterns by hand.

For this performance evaluation, we consider a basic scheme where an object *o1* invokes an empty method *m()* on an object *o2*. We consider the case where method *m()* does not take any parameter and the case where it takes a reference to another object *o3* as parameter. These measurements have been done under three conditions:

- on Java without integration if any access control policy,
- with the prototype implementation based on proxy object (filters),
- with the prototype implementation based on code injection.

Here are the resulting performance figures using a 1GHz Pentium processor with 256 Mo of RAM. These results are given for 10^8 iterations over the method call.

Table 1. Performance results

Operation	Straight call	Proxy-based	Injection-based
m()	1552 ms	6458 ms	3354 ms
m(o3)	1713 ms	14400 ms	3565 ms

Compared to a direct method invocation, a method call with access control is quite costly in the proxy-based implementation, due to object indirections and proxy instantiations. In the case of a single method call with no parameter, the injection-based implementation performs 48% faster than the proxy-based implementation. This is explained because we avoid two indirection calls. In the case of a method call with a reference parameter, the improvement is of 75%. In the proxy-based implementation, when we transmit a protected object reference (to *o3*) as parameter, *o2_stub* has to instantiate *o3_skel* to protect *o3*. In a worth case, *o2_skel* could have to instantiate *o3_stub* to implement the protection policy associated with object *o3*. In the code injection version, we don't have to instantiate any stub since the protection code is embedded in the caller and callee objects (however we have to pass new parameters to implement the capability transfer).

6 Related Work

Early attempts to manage protection by means of capabilities were based on specific hardware. Several capability-based hardware addressing systems [5][12] were built for the management of protected shared objects. These machine and system architectures were very popular in the 70s, but the standardization of the hardware and the widespread use of Unix systems stopped the trend towards capability based architectures.

In a second step, software capability based systems were designed on standard hardware, capabilities being protected by encryption. In these systems, the standard addressing mechanism of the underlying hardware (i.e. virtual addresses) is used for addressing objects, and capabilities are only used for object protection and access control. Examples of software capability based systems are the Amoeba system [11] based on the client-server paradigm and the Opal single address space system [2]. However, in these systems, capabilities are made available at the programming language level through capability variables that are used explicitly for accessing objects and exchanging access rights. In this paper, we propose to manage a capability-based access control mechanism as a non-functional aspect. The definition of an access control policy is completely separated from the functional code of the application, thus enforcing modularity and ease of expression.

Our first prototype implementation of the protection model relies on proxy objects [9]. Proxies (or indirection objects) are often used to implement non-functional aspects, as in the EJB [10] or CCM [4] component-based middleware systems. Our second prototype implementation relies on code injection, which has been often used

as a foundation technique to manage aspects [7]. To our knowledge, the experiment described in this paper is the unique attempt to manage a capability-based access control model as a non functional aspect.

7 Conclusion

In this paper, we presented an access control model which allows the definition of the access control policy of an application which interoperates which other applications. Access control is managed as a non-functional aspect in the sense it is defined at the level of the application interface, thus enhancing modularity and making this definition easier and clearer. The model is based on software capabilities and allows access rights to be dynamically exchanged between mutually suspicious applications. In this model, each application defines its protection policy independently from any other machine or application. This policy is enforced dynamically during execution.

Our protection scheme has been prototyped on the Java runtime environment. We actually implemented two prototypes. The first relies on object indirection to plug the access control policy of an application. The second relies on code injection to directly insert the access control policy of an application within that application's functional code, thus avoiding the overhead of object indirections.

We are currently experimenting with different non-functional aspects. The ultimate objective is to provide a generic framework which would allow to easily describe and integrate new non-functional aspects.

References

1. BCEL, http://bcel.sourceforge.net/
2. J. Chase, H. Levy, M. Feeley, E. Lazowska, Sharing and Protection in a Single-Address-Space Operating System, ACM Transactions on Computer Systems, 12(4), November 1994.
3. D. Chess, C. Harrison, A. Kershenbaum, Mobile Agents: Are They a Good Idea ?, IBM Research Division, T.J. Watson Research Center, New York, March 1995.
4. Corba Components – Volume I, OMG TC Document orbos/99-07-01
5. D. England, Capability, Concept, Mechanism and Structure in System 250, *RAIRO-Informatique (AFCET)*, Vol 9, September 1975.
6. J. Gosling and H. McGilton, The Java Language Environment: a White Paper, Sun Microsystems Inc., 1996.
7. G. Kiczales, C. Lopes, Aspect-Oriented Programming with AspectJ, Technical Report, Xerox PARC, 1998.
8. H. Levy, Capability-Based Computer Systems, Digital Press, 1984.
9. M. Shapiro, Structure and Encapsulation in Distributed Systems: The Proxy Principle, 6th International Conference on Distributed Computing Systems, 1986.
10. Sun Microsystems, Inc. *Enterprise Java Beans Specification*, Sun Microsystems, 2000.
11. A. Tanenbaum, S. Mullender, R. Van Renesse, Using Sparse Capabilities in a Distributed Operating System, 6th International Conference on Distributed Computing Systems, 1986.
12. M. Wilkes, R. Needham, The Cambridge CAP Computer and its Operating System, North Holland, 1979.