A Protection Scheme for Mobile Agents on Java

D. Hagimont [†], *L. Ismail* [‡]

SIRAC Project (IMAG-INRIA)

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

Internet: {Daniel.Hagimont, Leila.Ismail}@imag.fr

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

‡ INPG (Institut National Polytechnique de Grenoble)

Abstract: This paper describes a protection scheme for mobile agents 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 agents. Each agent defines the access control rules that must be enforced when interacting with other agents.

An important advantage of the proposed scheme is that the definition of the protection policy of an agent (i.e. how access rights are granted to other agents) is completely separated from the application code of that agent. It is described in an extended Interface Definition Language (IDL) at the interface level, thus enforcing modularity and ease of expression.

A prototype has been implemented and experiments with simple agent-based applications have shown the feasibility and the advantage of this method.

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.

One of the emerging paradigm for structuring applications over the Internet is agent-based programming [KGR96]. An agent is roughly a process with its own context, including code and data, that may travel among several sites in order to perform its task. Generally, an agent can invoke objects exported either by the servers it visits or by other agents running on these servers.

In the context of agent-based programming, protection has become one of the most important issues since it is a strict condition to the acceptance of this new paradigm: nobody will use it if any untrusted agent can bypass the protection mechanisms and damage information of the servers or of other agents running on these servers.

Java [GM95] is probably the best known runtime environment which provides facilities for mobile code. Java allows the development of mobile-code based applications through a C++ like language. 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 protection in agent-based systems, but this is not sufficient to allow interacting agents and servers to control access rights in a flexible way.

In this paper, we propose a protection scheme, based on software capabilities [Levy84] which allows mutually suspicious agents 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 agents during execution.
- Decentralization. Each agent is responsible for the definition of its own protection policy, which is linked with the code of the agent and moved with the agent. There's no need for registering these rules into a third party protection server.
- Mutual suspicion. All agents are equal with respect to protection. The system does not impose a hierarchy among interacting agents.
- Modularity. The protection scheme enforces modularity since the definition of protection is totally separated from the application code.
- Portability. The implementation of this protection scheme uses no features that are specific to Java. We therefore

believe that it could be integrated into other existing agent-based platforms.

This protection scheme for mobile agents has been prototyped on top of Java. Although a full mobile agent-based distributed system has not been implemented, all the features that are necessary to validate our proposal for protection have been implemented.

The rest of the paper is structured as follows. In section 2, we provide the requirements for the proposed protection mechanisms. Section 3 provides an overview of the Java features necessary to understand the rest of the paper. Our protection model is presented in section 4 and its implementation in Java described in section 5. After a comparison with related work in section 6, we conclude in section 7.

2 Motivations and Requirements

We will not insist on the interest of agent-based programming, whose advantages are overviewed in [CHK95]. In this section, we present the motivations and requirements we set when designing our protection model.

When considering protection in a mobile agent-based system, three important problems have to be considered:

- Isolation. Isolation refers to agents confinement. When an agent is hosted by a server, the server must assure that the agent only has access to its own objects or objects for which it has been granted access to by other agents[1]. The agent must not be able to bypass this rule and to directly address objects without the permission of its owner.
- Access Control. Access control is the definition of the policy that determines the rights given by an agent to other agents. For this purpose, an agent must be able to export object references while controlling the access rights it provides. The protection policy determines when and how these rights are exported.
- Authentication. Authentication refers here to the ability to associate an identity with each agent. This identity may be the identity of a user, a machine, a group of users, etc. According to this authentication, an incoming agent may then be provided with an initial protection environment. Then, this protection environment can evolve dynamically according to the agent's execution.

As pointed out, the first problem is not an open issue since the use of a strongly typed language such as Java solves it. The problem of authentication is not the issue we focused on, although we believe it would not be difficult to integrate an authentication service within our protection scheme. Instead, we focused on providing agents with the ability to easily control the access rights they grant to other agents. In the rest of the paper, we will use the word *protection* to mean access control on objects.

When designing this protection scheme for mobile code, we had the following requirements:

- Evolution. In an agent based system, it is very important to allow access rights to evolve dynamically. When an agent moves to a site, it is first authenticated and according to this authentication, it is provided with a reduced access rights environment. Then, during execution, it can be granted additional rights, depending on the agents with which it interacts on the host machine.
- Decentralization. Each agent is responsible for the definition of its own protection policy. When this policy is expressed by a programmer, he does not yet know which machines the agent will visit. Therefore, the protection rules cannot be registered in the kernel (or a centralized protection server), as it is the case in traditional operating systems. In agent-based programming, the protection rules should be linked with the agent and moved with the agent.
- Mutual suspicion. Since we consider mutually suspicious interacting agents, all the agents are equal regarding protection. The system must not impose a hierarchy between interacting agents as it is the case in a system that implements a master/slave model.
- Modularity. We don't want to provide extensions to be used from the programming language, since it would make programs much more complex and difficult to maintain. Instead, our goal is to enforce modularity and simplicity by separating the definition of protection from the application code.
- Portability. Finally, we tried to design a protection scheme which is quite general and applicable to different agent based environments. Our protection model does not rely on features that are specific to Java and it could be ported to any platform based on a safe language like Java. However, in the rest of the paper, we will describe this model using Java's abstractions in order to be more practical and precise.

3 The Java Environment

In this section, we recall the aspects of the Java environment that are relevant to our experiment and we explain how mobile agents can be managed on top of Java.

3.1 Aspects of Java

Java is a C++ like object-oriented programming language that is used to generate programs that can be executed on a portable runtime environment that runs on almost every machine types. Since Java is very popular and well known, we will only describe the features used in our experiment: code mobility, polymorphism and dynamic binding.

A 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 in order to confine error propagation. 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 during execution. 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 that 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. We must notice here that Java also provides (in its last release [Sun96]) an object serialization feature that allows instances to be transferred between different runtimes.

Another very interesting aspect of Java we used is polymorphism. Polymorphism refers to the ability to define *Interfaces* (or types) and classes separately. An interface is a

[1] The hosting server may be viewed as a particular agent, except that agents have to trust the server.

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.

Java also implements dynamic binding which is crucial to mobile code. Dynamic binding means the ability to determine at run-time the code to be executed for a method invocation.

Since Java allows classes to be dynamically loaded, a variable of an interface type can be assigned to a reference that points to an object, which class was loaded dynamically. Java postpones the binding of the code (of this variable) until invocation time, thus allowing the execution of dynamically loaded classes.

3.2 Managing Mobile Agents on Java

Three main issues have to be addressed when managing agents on a runtime environment such as Java:

- Execution of agents on the runtime. A machine that hosts incoming agents is running the Java runtime environment. This runtime must implement facilities for executing agents concurrently. Java provides the ability to run several threads on a Java runtime running in a standard Unix process. Therefore, it is easy to manage several concurrent application processes (called agents) on the same runtime. When an agent needs to execute on the runtime, the runtime creates a new thread which executes the agent's program.

- Migration of agents between different runtimes. The second important issue when implementing a mobile agent distributed environment is agent migration. When an agent migrates, two kinds of objects have to be transferred: classes and instances. The classes constitute the program executed by the agent while the instances compose the execution context of the agent. For code transfer, Java provides the ability to load classes dynamically, these classes being local or remote classes. Thus the migration procedure can dynamically transfer the classes that compose the agent's program. Java's serialization feature allows instances to be transferred between different runtimes.

  In the framework of the experiment described in this paper, we didn't focus on agent mobility. But it has no influence on the demonstration of the protection model described in this paper. In the prototype, agents are loaded dynamically and cooperate with other agents.

- Sharing of objects between agents on one runtime. Sharing of object between agents that execute within a single Java runtime is implicit because all these objects are managed in the same virtual address space (in which the Java runtime executes). However, in order to share objects, agents must exchange object references (Java object pointers). Thus, the runtime must provide a name server that allows objects references to be exchanged, i.e. to associate symbolic names with object references. We assume that this name server is used by agents in order to start cooperating. When an agent obtains a reference to an object from the name server, it can invoke the object by using the Java interface that this object is supposed to implement (the two agents must agree on an interface in order to cooperate).

4 Protection Model

In this section, we present our protection model based on software capabilities.

4.1 A Capability-Based Protection Model

The protection model we propose is based on software capabilities [Levy84]. The advantage of capabilities is that they allow access rights to evolve dynamically, which is one of our objectives.

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 agent must own a capability to that object with the required access rights. When an object is created, a capability is returned to the creator, 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 agent, 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 agent.

Therefore, each agent executes in a protection environment in which it is granted access to the objects it owns. This agent 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 agent enough access rights to use the reference.

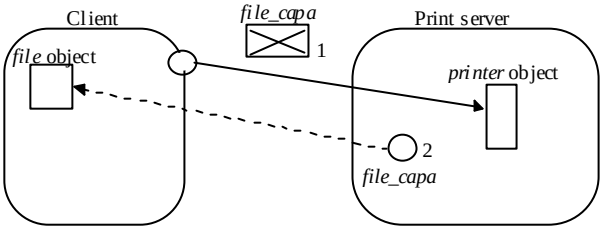

Figure 1. Print server example

In order to illustrate capability based protection, let us consider the example[2] of a *Printer* object, exported by a print server, that allows a client to print a file (Figure 1).

A capability on the *Printer* object is given to the client agents providing them with the right to print files. When a client wants to print a file (*File* object), the *Printer* object needs to get read rights for this file; therefore the client will pass, at invocation time (1), a read-only capability on the file (*file_capa*) to the callee agent. This capability allows the *Printer* object to read the contents of the file (2).

4.2 Exchanging Capabilities

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

---

2 This example only aims at explaining the protection model. We don't use it as an example of mobile agent based application.

One strong motivation for our protection model is modularity. Indeed, we don't want to provide extensions to the programming language that allow an agent to express capability parameter passing when another agent's object is invoked. This would overload programs and make them much more difficult to maintain.

To achieve this goal, our idea is to define capability exchanges between interacting agents 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, thus enhancing modularity.

Therefore, an IDL has been defined that allows the agent 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 structured as shown in figure 2. It 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.

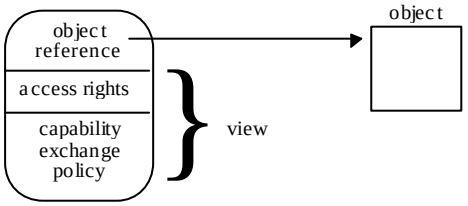

Figure 2. Structure of a capability

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 *File* 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* (*File f* **pass** ***reader***). This signature expresses that a capability with the *reader* view must be passed to the callee agent along with the reference to a file 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 agents are mutually suspicious. Each agent must have full control over the capabilities it exports to other agents (both the caller and the callee). Moreover, we want to ensure agents independence (decentralization). More precisely, it is not possible for an agent programmer to verify the protection policy defined by an agent that exports a service since at programming time, the programmer may not yet know which agents it is going to interact with.

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

The view defined by the callee agent *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 agent 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 agent *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 agent accepts to export.

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

When an agent exports an object reference through the name server (described in section 3.2), it defines the view associated with the reference, i.e. the capability which is exported for this exported reference. This way, the agent also defines the capabilities that may be exported subsequently to an invocation of that object.

When an agent fetches the reference from the name server, it also defines the view associated (on its side) with the reference it obtained. This way, the agent 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 agents.

4.3 Example

In order to illustrate the expression scheme of our protection model, let's consider again the print server example (a little bit modified).

```
interface Printer_itf {
  void init ();                    // initialize the printer
  Job_itf run (Text_itf text);     // send text to printer
}
interface Text_itf {
  String read();                   // read the text
  void write (String s);           // write the text
}
interface Job_itf {
  void stop ();                    // kill the current job
}
```

Here are the Java interfaces of the *Printer* application. These interfaces are shared between the caller and the callee. In order to make the print service available to the clients, the *Printer* agent 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 agent fetches this instance from the name server and can invoke a method (*init* or *run*) on this instance, using the *Printer_itf* interface (in the name server, instances have no type, i.e. they are of class *Object*[3], which means the client needs an interface in order to force the type of the reference and to invoke the instance).

When the client wants to print a file, it invokes the method *run* and passes a reference to an instance of class *Text* which implements interface *Text_itf*. The *run* method returns a reference to an instance of class *Job* that implements interface *Job_itf*. The client agent can invoke this instance in order to stop the *job*.

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 following views:

client

```
view client implements Printer_itf {
        void init ();
        Job_itf run (Text_itf text pass reader);
}
view reader implements Text_itf {
        String read();
        void not write (String s);
}
```

print server

```
view server implements Printer_itf {
        void not init ();
        Job_itf run (Text_itf text);
}
```

Each agent 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 protects. 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 *run*. The client defines the view *client* which says that, when a reference to a text is passed as a parameter of method *run*, 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 agents.

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

5 Experimentation

The protection model described in the previous section has been implemented on Java. In this section, we present this implementation and we present a first application that uses this model.

5.1 Implementation on Java

For the implementation of this protection model, we used the fact that Java object references are almost capabilities[4]. 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 agent does not export a reference to one of its objects, these objects are protected against other agents. 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 [Shapiro86], 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 agent, a filter class is generated (by a pre-processor) and an instance of that class is inserted to protect the agent.

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 agent providing the reference.

This filter class implements all the methods declared in the interface of the view. It defines an instance variable 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 agent. In fact, this filter instance is inserted by the filter used for the current invocation.

---

[3] *Object* is in Java the most general class. All classes inherit from *Object*.

[4] This is the case for any object-oriented strongly typed language (safe language).

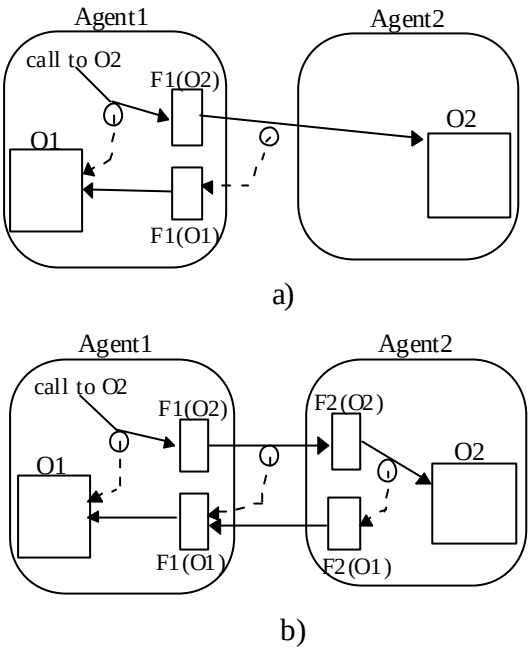

Figure 3. Management of filters.

In figure 3a, the invocation of *O2* performed by *Agent1* passes a reference to *O1* as parameter. The filter *F1(O2)*, which corresponds to the protection policy of *Agent1* 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.

Conversely, when a reference is received by an agent, a reference to a filter instance is passed instead of the received parameter, which class is generated from the view specified by the agent that receives the parameter. In figure 3b, the filter *F2(O2)*, which corresponds to the protection policy of *Agent2* 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 *Agent1* to *Agent2*. These two filters behave as follows:

- *F1(O1)*: it enforces that only authorized methods can be invoked by *Agent2* and it inserts filters on the account of *Agent1* for the parameters of invocations on *O1* performed by *Agent2*.
- *F2(O1)*: it inserts filters on the account of *Agent2* for the parameters of invocations on *O1* performed by *Agent2*.

Below is the code of the filter classes for the print server example.

client

```
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 !!!
    }

    Printer_itf obj;
    public Printer_stub(Printer_itf o) {
        obj = o;
    }
    public void init() {
        obj.init();
    }
    public Job_itf run(Text_itf text) {
        reader stub = new reader(text);
        return obj.run(stub);
    }
}
```

print server

```
public class server implements Printer_itf {
    Printer_itf obj;
    public Printer_stub(Printer_itf o) {
        obj = o;
    }
    public void init() {
        Exception !!!
    }
    public Job_itf run(Text_itf te xt) {
        return obj.run(text);
    }
}
```

The filter class *reader* of the client forwards *read* invocations to the actual instance, but it does not forward *write* invocations. Similarly, the filter class *server* of the print server only forwards invocations of the *run* method. In the filter class *client* of the client, method *run* takes as parameter a reference *text* for which a capability with the *reader* view must be passed. For this parameter, the *run* method of the filter class creates an instance of the filter class *reader* and initializes it with the actual parameter, and forwards the invocation, passing as parameter the created filter instance instead of the actual parameter.

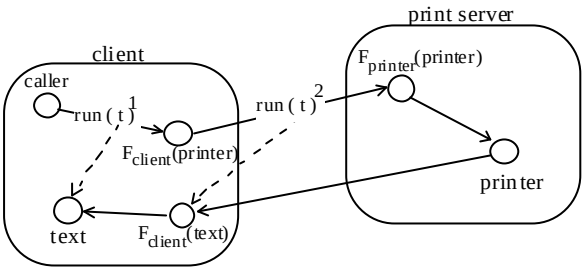

Figure 4. Filter objects management in the print server.

Figure 4 illustrates the management of filter objects in the print server example. At step 1, the invocation of the *run* method passes the reference of the actual *text* object. This invocation is performed on the filter object of the client for the reference to the *printer* object. This filter object creates a filter object for the text parameter and forwards the invocation to the filter object of the print server (step 2), passing the filter object of the text parameter. The invocation is then forwarded to the actual *printer* object. Later, the invocation of the *read* method on the *text* object goes through the filter object that was inserted by the client.

Notice that if the client had defined a view *job_view* for the reference to the instance of class *Job* returned by method *run*, the filter class *client* would have the following *run* method:

```
public Job_itf run(Text_itf text) {
  reader stub = new reader(text);
  return new job_view(obj.run(stub));
}
```

This method creates two filter objects, one for the (text) input parameter and one for the (job) returned parameter. In the case of the print server described earlier, view *job_view* is not necessary.

To sum up, we have presented the implementation of our protection model on Java. The definition of the protection policy of an agent is based on the expression of view that are used to generate object filters that are inserted between the caller and the callee. Each agent defines its own protection policy independently from any other agent or server. The generated filter classes are linked with the code of the agent and moved with the agent.

The current prototype is composed of a runtime built on Java that manages cooperating agents and a view processor that generates filter classes from views. Agent programs are developed with the Java language and are loaded dynamically by our runtime. The view processor is used to generate filters independently from the application code.

5.2 Application Example

The print server example we used earlier to illustrate our protection model only aimed at explaining the design and implementation on a very simple example.

In order to validate our protection model, we implemented an application where several mutually suspicious agents cooperate. This application is a cooperative calendar. In the execution scenario we considered, each user keeps a copy of its calendar on his personal computer, but may send an agent to organize an appointment between the members of a group. This agent contains a copy of the user's calendar and it visits other users' calendars in order to find a date that is free for everybody. One of the advantages of mobile agent programming is that the users of the application may be disconnected. For instance, a user may send an agent to organize a meeting and disconnect his computer. The agent visits other machines with other calendars (and eventually wait for currently inaccessible ones) before it returns to its original machine (eventually waiting for the machine to be reconnected).

If we want to allow cooperation between calendars, we cannot let any application read freely the content of the calendars. A calendar agent should only be able to look up free spaces in another user's calendar.

In this application, when two calendars meet on a machine, the initiator of the meeting can look in the other agent's calendar in order to check whether a date is free or not. This initiator can also create a *proposal* object in the calendar in order to propose a meeting. This proposal is supposed to be validated or rejected by the owner of the calendar, the result being collected later by the initiator of the meeting.

This application example has been developed on Java. The result of this experiment with the calendar application is that it was very easy to describe the protection policy of the application.

6 Related Work

The research in the area of mobile agent programming is very active. Several projects have implemented runtime environments for mobile agents. In these projects, the crucial issue of protection is addressed as described in this section.

In general, these projects address the problem of protection mainly from the point of view of isolation and authentication. Regarding isolation, most of them rely on a safe language and interpretation in order to guarantee that an agent cannot bypass protection. Similarly, our approach is to rely on the safety of the Java language. Regarding authentication, they provide functions for authenticating the entity that invokes an operation on an object. As mentioned earlier (section 2), authentication is a very important issue that has many implications on access control. In continuation to this work, we plan to integrate an authentication service with our protection scheme for providing agents with initial access rights environments (initial sets of capabilities).

In the rest of this section, we compare our work to others as regards the expression of access control policies.

Regarding access control, we can relate our work to three ongoing projects: AgentTcl [GCKR96], AgletIbm [IBM96] and Telescript [Magic96].

AgentTcl is a mobile agent based system in which programs are written with the Tcl interpreted language. Protection in AgentTcl [Gray96] relies on an extension of Tcl called SafeTcl [OLW96], which allows a program to be executed on a safe interpreter. A safe interpreter only provides a subset of the primitives of Tcl. Cooperation between agents is possible by means of message passing. An agent can send a message to another agent, and the receiving agent can authenticate the agent that sent the message, i.e. know about the server from which the message was sent or the identity of the agent that sent the message. The receiving agent uses these authenticated attributes to respond appropriately to the incoming message, implementing its own access control policy. Compared to our proposal, AgentTcl is not object oriented and does not allow data sharing. Moreover, a protection policy in an application is much more difficult to implement.

AgletIbm is a runtime for mobile agents built on Java. In AgletIbm, cooperation between agents is only possible by means of message passing. The runtime provides an agent authentication service. Knowing its identity, an agent can send a message to another agent in order to cooperate with it. This message passing may be local to a machine (within the same Java virtual machine) or remote if the agents don't meet on one machine. As for

AgentTcl, object sharing is not possible and protection policies have to be totally implemented by the programmer.

In the Telescript system, agents can interact through shared objects. Telescript provides mechanisms for authenticating the entity that invokes an operation on an object. This entity may notably be the calling object (its owner), the calling process (an agent). Access control must be managed in programs that must check if the operation is allowed according to the calling entity. This has to be managed manually in programs. Compared to Telescript, our proposal differs by the fact that we separate access control definition from the code of the application, protection being defined at the level of the agent interface. This modularity eases protection definition.

7 Conclusion and Perspectives

In this paper, we presented a protection model which allows the definition of the access control policy of a mobile agent.

Access control is defined at the level of the agent 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 agents. In this model, each agent defines its protection policy independently from any other machine or agents. This policy is enforced dynamically during execution.

Our protection scheme has been prototyped on the Java runtime environment and experiments with simple applications revealed the advantages of this approach.

This work currently goes on. The main objective is to implement a full mobile agent based system on top of Java and to experiment with larger mobile applications. We plan to identify more precisely the requirements of the different classes of applications and complete our proposal with new functions, notably with authentication functions.

Bibliography

[CHK95] D. Chess, C. Harrison, A. Kershenbaum, "Mobile Agents: Are They a Good Idea ?", IBM Research Division, T.J. Watson Research Center, Yorktown Heights, New York, March 1995, URL: http://www.cs.dartmouth.edu/?agent/papers/chapter.ps.Z

[GCKR96] R. Gray, G. Cybenko, D. Kotz, and D. Rus, "Agent TCL", Department of Computer Science, Dartmouth College, Hanover, NH, May 1996. URL: http://www.cs.dartmouth.edu/?agent/papers/chapter.ps.Z

[GM95] J. Gosling and H. McGilton, "The Java Language Environment: a White Paper", Sun Microsystems Inc., 1995. URL: http://java.sun.com/whitePaper/java-whitepaper-1.html

[Gray96] R. Gray, "A Flexible and Secure Mobile-Agent System," Fourth Annual TC/TK Workshop'96, July 1996.

[IBM96] Iternational Business Machines Corporation: "Mobile Agent Facility Specification," OMG TC Document cf/96-08-01, August 1996. URL: http:/www.trl.ibm.co.jp/aglets/maf.ps

[KGR96] D. Kotz, R. Gray, and D. Rus, "Transportable Agents Support Worldwide Applications", Department of Computer Science, Dartmouth College, Hanover, NH, March 1996.

[Levy84] H. M. Levy, "Capability-Based Computer Systems", Digital Press, 1984.

[Magic96] Magic Inc, "An Introduction to Safety and Security in Telescript", Magic Inc. document, URL: http://www.genmagic.com/Telescript/security.html

[OLW96] J. Ousterhout, J. Levy, andB. Welch, "The Safe-TCL Security Model", Draft, Novembre 16, 1996.

[Shapiro 86] M. Shapiro, "Structure and Encapsulation in Distributed Systems: The Proxy Principle", Proc. of the 6th International Conference on Distributed Computing Systems, pp. 198-204, 1986.

[Sun96] Sun Microsystems, "JDK 1.1 Documentation", Sun Microsystems, URL: http://www.javasoft.com/products/jdk/1.1/docs/index.html