A Protection Scheme for a CORBA Environment

D. Hagimont†, O. Huet, J. Mossière¥

INRIA Rhône-Alpes - Projet Sirac

655 avenue de l'Europe

38330 Montbonnot Saint-Martin - France

Internet: {Daniel.Hagimont, Jacques.Mossiere}@imag.fr

Abstract: This paper describes a protection scheme for a CORBA environment. In this scheme, access to objects is controlled by means of software capabilities that can be exchanged between object servers.

The main advantage of our approach is that the definition of the protection policy of an application (i.e. how capabilities are exchanged between object servers) is completely disjoined from the application code and described in an extended Interface Definition Language (IDL). Therefore, protection is orthogonal to application code. This allows an application administrator to specify protection for existing modules and to easily change the protection policy of an application.

A prototype has been implemented on top of the Orbix system. Preliminary experiments with simple distributed applications have shown the feasibility and the advantage of this method.

1. Introduction

Protection is a crucial aspect of computing systems, in particular when users co-operate using shared objects. Since CORBA has become a standard in the object community, designing a protection service that is well integrated within the CORBA framework is a very important issue. Several well defined protection models have been proposed, among which access control list based models ([Kowalski 90][Hagimont 94]) and capability-based models ([Levy 84][Tanenbaum 86]), but integrating one of these protection models in a CORBA environment is still an open issue.

In this paper, we report on our experiment in designing and integrating a capability-based protection model in a CORBA environment. When designing this protection model, we wanted to keep most of the key concepts underlying CORBA: orthogonality between features (e.g. types and classes), reuse of code and modularity. From this, we reached the conclusion that we should manage a clear separation between an application code and the definition of the application protection policy:

---

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

¥ Institut National Polytechnique de Grenoble

- Orthogonality between features. The separation between code and protection ensures orthogonality between the implementation of a service (including the programming language used to develop it) and the protection policy of that service.
- Reuse. The separation between code and protection allows protecting an already developed application or reusing an unprotected module. It also allows a single application code to be executed with different protection policies.
- Modularity. If the definition of protection is not wired in the application code, it makes protection and application code more easy to develop and to maintain.

The protection model we propose allows defining the protection policy of an application only in terms of the application's interface. The idea is implemented by managing capabilities in a separate protected layer (one or several protected servers), and by providing administrators with an extended interface definition language (IDL [OMG 91]) in which the management of access rights can be expressed. This results in a framework in which application code does not depend on protection; protection-related actions (access rights checking and rights transfers) are only triggered when the application traps into the protected layer upon server invocations.

The rest of the paper is structured as follows. In section 2, we present our protection model which is based on *Hidden Software Capabilities* [Hagimont 96], a model we proposed in a recent paper. Section 3 describes our prototype implementation in the Orbix CORBA compliant object system. After a brief comparison with related work in section 4, we conclude in section 5.

2. Protection Model

2.1. General assumptions

In CORBA, objects are managed in servers[1] and there is no protection between objects managed within the same server. Protection inside servers is out of the scope of providing protection for CORBA since CORBA is supposed to be independent from in-server's object implementation. Therefore, the protection scheme we propose aims at controlling access rights between applications running in separate servers. Mutually suspicious applications are supposed to be run in separate servers in order to benefit from our protection scheme. In the rest of the paper, we will use the word application to refer to the entity that performs operations on objects, this application being physically run in one server.

[1] A server can be a client of another server.

2.2. A Capability-Based Protection Model

The protection model we propose is based on software capabilities [Levy 84] [Tanenbaum 86]. The advantage of capabilities is in their flexibility since they allow access rights to evolve dynamically.

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 on that object with the required access rights. When an object is created, a capability is returned to the creator and 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 called *protection domain*, 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 (e.g. a C++ pointer) 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.

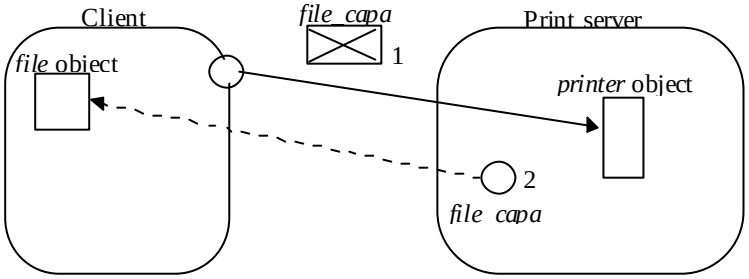


Figure 1. Print server example

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 file (Figure 1).

A capability on the *Printer* object is given to the clients 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 right for this file; therefore the client will pass, at invocation time (1), a read-only capability on the file (*file_capa*) to the callee. This capability allows the *Printer* object to read the contents of the file (2).

2.3. Exchanging Capabilities

As explained in the introduction, in order to keep access control orthogonal to the application code, a basic design choice is to define access rights management using an interface definition language (IDL) [Hagimont 96]. Since an interface can be described independently from any

implementation, describing the protection policy at the level of the interface allows it to be clearly separated from the code of the application, thus enhancing modularity.

In the context of a capability based system, access rights management mainly refers to capability exchanges between interacting applications. The definition of a protection policy specifies when (and what kind of) capabilities are exported by a server to its clients when object references are passed as invocation parameters to the clients.

Therefore, we have defined an IDL 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 for the parameters of the methods that are authorized in the view. These transferred capabilities are expressed in terms of views.

A capability is structured as shown in figure 2. It includes the reference to 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.

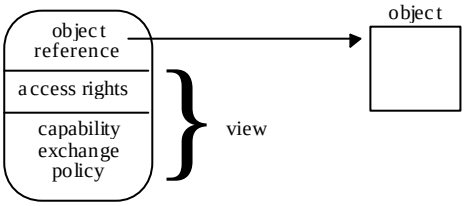


Figure 2. Structure of a capability

The definition of views is naturally recursive since it specifies the capabilities that should be transferred with parameters, this specification being in terms of views. For that reason, each view is given a name at definition time. An application may define a view for a class *C* and associate a name *V* with it. Then, *V* can further be used to specify the view of the capability that must be transferred with a parameter which is a reference to an instance of class *C*.

Views are defined by the programmer of a server. When a server must provide access to the service it implements, the server's programmer defines the views associated with the different access rights he needs to manage. Then, he can export some capabilities to his clients according to how he trusts those clients. At execution time, all the capability transfers are performed according to the views definition of the server.

For example, let us consider again the printer example described in section 2.2. The client can invoke the print server only if it owns a capability associated with an instance of class *Printer*. The view associated with the capability must authorize method *Print*. It also specifies that a capability with read right on the file must be transferred along with the reference on the *File* object to be printed.

Below are shown the interfaces and views associated with classes *Printer* and *File*.

```
class Printer {
    method PrinterStart ();
    method PrinterStop ();
    method Print ( in File f );
}

class File {
    method Read (...);
    method Write (...);
}

view Printer_user {
    method Print ( in Reader File f );
}
view Printer_admin {
    method PrinterStart ();
    method PrinterStop ();
}

view Reader {
    method Read (...);
}
view Writer {
    method Write (...);
}
```

The above definitions include two views associated with class *Printer*. View *Printer_user* corresponds to the rights that are given to a regular user of the printing service. View *Printer_admin* is supposed to be given to privileged people responsible for the printing service administration. Two views (*Reader* and *Writer*) are associated with class *File*.

When a capability for the print service is given to a client, this capability has its view attribute set to *Printer_user*. This means that the client can only invoke the *Print* method and cannot invoke *PrinterStart* nor *PrinterStop*. This also means that a capability, associated with the *f* parameter, with the *Reader* view, will be passed to the print server when the *Print* method is invoked.

Till now, we have considered that the server was providing capabilities to its clients and that the clients were accepting the capabilities. But in most of the cases, we must take into account the mutual suspicion between the client and the server. To meet the requirements of mutually suspicious interacting servers, we must allow:

- the callee to control the capabilities that enter (at call time) and leave (on return) the domain,

- the caller to control the capabilities that leave (at call time) and enter (on return) the domain.

Therefore, when a capability is transferred to an application, to fully control the access rights, both the callee and the caller should define views that correspond to their respective requirements. Then the system should match these views in order to check their compatibility. However, when a capability is first installed in the environment of a client (by the client programmer), the view associated with the capability exported by the server corresponds to the access rights required by the server to execute properly. If a client does not agree with the requirements expressed by the server, he can decide not to import the capability.

This control by the client is done statically (at administration time) when the client installs new capabilities in its environment, providing access to new servers from its environment. By checking (recursively since view are recursive) the views of the capabilities he installs, the client will know about the views of all the capabilities that may dynamically enter or leave its protection environment. Thus, the client can prevent Trojan horses capabilities from entering its protection environment. We assume here that the views tree that a client will have to control is not very deep since object's references exchanged between servers are very few compared to the number of objects managed by these servers.

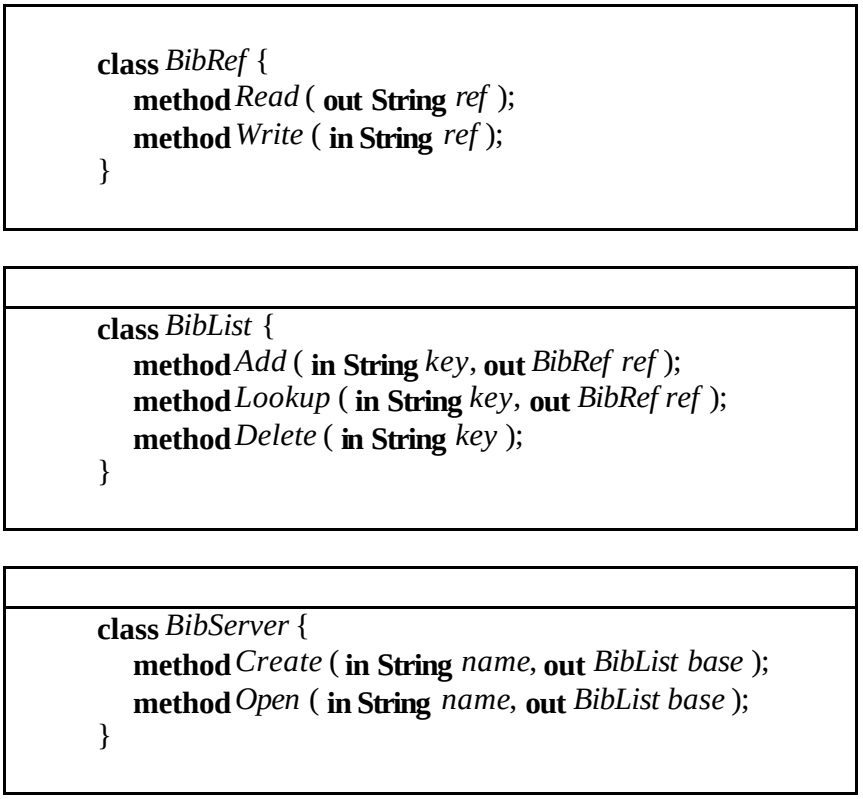
2.4. Examples of Protected Applications

In this section, we consider two applications. The first one illustrates the use of multiple protected interfaces associated with the same class. The second shows how mutually suspicious servers may be protected.

2.4.1. Use of Different Protected Interfaces

We provide here an example of a server that exports capabilities with different protected interfaces. This example has been implemented on top of our prototype and used to validate the protection model.

The application is a server that manages bibliographic references. The entry point of the server is the *BibServer* class whose methods *Create* and *Open* respectively allow to create or open a bibliographic base. A bibliographic base is an instance of class *BibList* which implements a list of references. The interface of *BibList* allows references to be added (*Add*), retrieved (*Lookup*) or deleted (*Delete*). A bibliographic reference (of class *BibRef*) contains two variables which are the key (*key*) used to quote the reference and the text of the reference (*ref*). Methods *Read* and *Write* allow the *ref* field to be respectively read and written. The interfaces of these classes are the following:



and below the protected interfaces we defined for those classes:

```
protected interface BibRef_reader {
    method Read ( out String ref );
}
protected interface BibRef_writer {
    method Read ( out String ref );
    method Write ( in String ref );
}
```

```
protected interface BibList_reader {
    method Lookup ( in String key, out BibRef_reader BibRef ref );
}
protected interface BibList_contributor {
    method Add ( in String key, out BibRef_writer BibRef ref );
    method Lookup ( in String key, out BibRef_reader BibRef ref );
}
protected interface BibList_owner {
    method Add ( in String key, out BibRef_writer  BibRef ref );
    method Lookup ( in String key, out BibRef_writer  BibRef ref );
    method Delete ( in String key );
}
```

```
protected interface BibServer_user1 {
    method Create ( in String name, out BibList_owner BibList base );
    method Open ( in String name, out BibList_owner BibList base );
}
protected interface BibServer_user2 {
    method Create ( in String name, out BibList_owner BibList base );
    method Open ( in String name, out BibList_contributor BibList base );
}
protected interface BibServer_user3 {
    method Create ( in String name, out BibList_owner BibList base );
    method Open ( in String name, out BibList_reader BibList base );
}
```

Using these protected interfaces, the administrator of the server may provide capabilities to three classes of users:

- Fully trusted users (*BibServer_user1*): when a bibliographic base gets opened or created, a *BibList_owner* capability is returned, allowing users to modify the base as they wish.
- Half trusted users (*BibServer_user2*): when a base is opened, a *BibList_contributor* capability is returned, only allowing users to add new references to the base.
- Untrusted users (*BibServer_user3*): users only get a *BibList_reader* capability when the base is opened.

This is an example of protected interfaces which may be associated with this set of classes. However, without any code modification, the same set of classes could be used with a different protection policy.

Notice also that the client application which uses the protected server may obtain a *BibList_owner* capability and transfer it to another protection domain with a *BibList_contributor* protected interface (this would be described in the protected interface used to transfer the capability to the other protection domain).

2.4.2. Mutually Suspicious Servers

Let's consider now an example in which mutually suspicious servers interact.

This example is a refinement of the printing server described in section 2.2. The printing server receives a reference to the file object to be printed out, but also a reference to a signal object that must be invoked when the printing completes. The printing server also returns a reference to a status object that may be invoked by the client in order to get the status of the printing.

The interface of the printing class is the following:

```
class Printer {
    method Print ( in File f,
                   in Signal done,
                   out Status state );
}
```

We don't describe here the interfaces of the classes *File*, *Signal* and *Status*. They all define the methods *Read* and *Write* on file, signal or status objects (writing a signal means here signalling the associated event). We also don't describe the protected interfaces for those classes, which are obviously the *Reader* and *Writer* protected interfaces as defined for the *File* class in section 2.3.

Then, the protected interface of the *Printer* class is:

```
protected interface Printer {
   method Print ( in File Reader f,
                  in Signal Writer done,
                  out Status Reader state );
}
```

When the server administrator defines this protected interface to be associated with the capability he exports to clients, he specifies his requirements regarding object reference parameters. The administrator expresses that the server needs read right on the file, write right on the signal object and that read right on the status object will be returned after invocation.

When a capability with this protected interface is installed in the client protection domain, the client verifies that this capability is not a Trojan horse in his protection domain. For each parameter in each method from the protected interface, the client must check the following:

- for "out" parameters, the client must verify that the server doesn't try to introduce a Trojan horse capability associated with the returned parameter. Recursivity is present here. In our example, the client will do the same verification for the *Reader* protected interface (associated with the *state* parameter) than for the *Printer* protected interface.
- for "in" parameters, the client must verify that the protected interfaces associated with the parameters are not giving too much rights to the server. For instance, the protected interface associated with the *f* parameter should not be *Writer*; it must be *Reader*. Also, the client must verify the protected interfaces associated with parameters in the methods from the *Reader* protected interface. Indeed, the protected interface *Reader* may have been defined by the server. Recursivity is also present here.

2.5. Extensions to the IDL

As explained in the section 2.3, we felt that it was not necessary for a client to provide a protected interface (that specifies his protection requirements) to be matched with the server's one at capability installation time. The client can check the interface proposed by the server; the capability is installed only if the client agrees on it.

However, in some cases, it may be interesting for the client to specialize the protected interface of an installed capability. We describe in this section two examples of protected interface specialization. These extensions are not integrated in the prototype described in section 3, but we illustrate here the perspectives of our protection model.

Administration of capabilities

The first extension is related to protection domains administration.

In order to simplify the administration of capabilities in protection domains, we propose to manage lists of capabilities. A symbolic name is associated with a list of capabilities and a list may be included in a list, allowing the domain administrator to organize access rights in a domain as a hierarchy like directory trees in the Unix system.

Thus, in order to facilitate the administration of capabilities, the IDL can be extended to allow the management of capabilities in this environment. When a protected interface specifies the transfer of a capability to a protection domain, it is then possible to specify the list where the capability should be installed in that protection domain.

The administrator of a server can specify in the protected interfaces he defines, where server-entering capabilities must be installed. Symetrically, a client that accepts the capability provided by the server may specialize the protected interface in order to define where client-entering capabilities must be installed. This is done by copying the capability and overloading the protected interface of the capability. Naturally, the system layer in which protected interfaces and capabilities are managed must not allow the client to modify the installation clauses of the server.

In the following example, a server that implements the management of a library has defined the protected interface *Lib_user*. When a document is registered in the library, the capability on the document provided by the client is stored in the *bib_doc_list* list. When a document is searched for, a capability with read right on the document is returned.

```
class Library {
   method Register ( in File  book );
   method Lookup ( in String  name,
                   out File book );
}
```

```
protected interface Lib_user {
   method Register ( in File Reader book
                     install bib_doc_list );
   method Lookup ( in String  name,
                   out File Reader book );
}
```

Using an *install* clause, the server's administrator may associate different lists with different users.

On the other side, the client may specialize the capability provided by the server, for installing the returned capability in the *book_list* list. The protected interface overloaded by the client is:

```
protected interface Lib_user {
   method Register ( in File Reader book
                        install bib_doc_list );
   method Lookup ( in String  name,
                   out File Reader book
                        install book_list );
}
```

Restriction on leaving capabilities

The IDL can also be extended in order to allow the administrator to place restrictions on capabilities leaving the domain both on the client and the server side. For instance, the library server described above can specify a capability list (*restricted_list*) only from which returned capabilities can be taken. The corresponding protected interface is the folllowing:

```
protected interface Lib_user {
   method Register ( in File Reader book
                        install bib_doc_list );
   method Lookup ( in String  name,
                   out File Reader book
                        from restricted_list );
                   }
```

Using a *from* clause, the server's administrator may specify different lookup environments associated with different classes of users.

Of course, such a restriction could be included in the server's code. *Hidden Software Capabilities* allow the protection policy to be defined in protected interfaces independently from the application code.

We have shown in this section the perspectives of programming protection at the IDL level. The next section is devoted to the implementation of the protection model.

3. Integration of Protection in the Orbix System

In this section, we describe the integration of our protection model within the Orbix system [Iona 94] which is a CORBA-compliant runtime system developed by Iona Ltd.

3.1. Hypotheses

The main requirement we had for the implementation is that servers must not be able to bypass access control. Moreover, since we are considering mutually suspicious servers, we didn't want only to protect servers against their clients. Therefore, we are looking at possible attacks, both from the clients and from the servers.

In this implementation, we are assuming that each protection domain is able to authenticate the sender of any message it receives. The technique used to implement this authentication service is out of the scope of our experiment. In our prototype, the sender just includes in the request its identification and we assume the correctness of the received identification.

3.2. Implementation

In the implementation, the techniques used for access control and capability transfers are very similar for the invocation (outward) and for the return (backward) from the invocation. Therefore, in the following, we mainly describe the implementation in one way.

In order to guarantee the reliability of our protection service, we decided to manage access rights in a centralized protection server (the protected layer of the introduction). This server (called *PS* from now on) manages protection domains (sets of capabilities) for all the processes running in the system. *PS* also provides an interface for the management of protected interfaces. A protected interface is identified by a symbolic name. A server cannot modify the association between a name and the protected interface since this protected interface may be consulted by a client and it corresponds to a contract between the server and the client.

Architecture

In order to invoke *PS* at call time, we implemented a post-processor which is used to modify the stubs generated from the IDL specification of a class. The modifications to the stubs are independent from the application protection policy: a call to *PS* is integrated in order to check access rights and trigger capability transfers.

The basic interactions between the client (*C*), the server (*S*) and the protection server (*PS*) are described on figure 3.

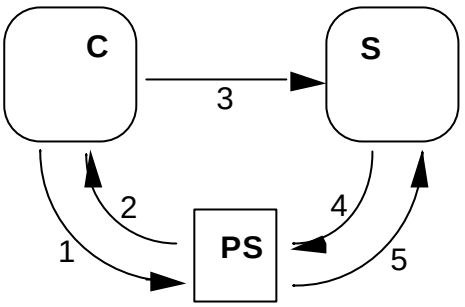


Figure 3. Implementation on Orbix

In figure 3, we show the messages that are exchanged between *C*, *S* and *PS* at invocation time.

The main steps of an invocation are:

- messages 1 and 2: *C* calls *PS* which checks if invocation is permitted and prepares an invocation descriptor with an identifier *id*.
- message 3: effective invocation (*id* is added to the parameters).
- message 4 and 5: *S* calls *PS* (passing *id* as a parameter) which verifies that *id* is valid and completes capabilities transfers.

Then, the invocation in *S* can proceed.

Authentication

As explained in section 31, authentication is out of the scope of this paper. However, we explain how the authentication of the caller (our hypothesis) can be used to prevent the protected service to be bypassed.

When a capability is installed in a protection domain, we register in the capability the identifier of the server that exports the capability and the identifier of the client that receives it.

When *C* calls *PS*, knowing the identifier of *C*, we search for a capability that allows the invocation in the protection domain associated with *C*. This is done in *PS* and cannot be bypassed. If a capability is found, it is registered in the invocation descriptor (*id*).

When *S* receives the invocation from *C* (message 3), it knows the identifier of the caller *C*. *S* calls *PS*, passing *id* and the identifier of the caller *C*. Therefore, *PS* can verify the validity of the invocation, i.e. that the caller is effectively the initiator of the request, preventing a client from using an invocation descriptor prepared by another client.

In the same way, when *PS* receives the call from *S* (message 4), it knows the identifier of *S*. It can therefore verify that *S* is effectively the server which is invoked (the server that exported the

capability associated with *id*), preventing a server from using an invocation descriptor addressed to another server.

Implementation

As mentioned above, the client stub is modified in order to invoke *PS* before the effective invocation (messages 1 and 2 in figure 3). The message sent to *PS* contains the identifier of *C* (for authentication) and the characteristics of the invocation, including the identification of the invoked objects (Orbix provides a function which returns a name, associated with the object and unique in the whole system), the name of the called method and the parameters of the call. Then, *PS* performs the following operations:

- *PS* checks access rights for the invocation. It verifies that the protection domain associated with *C* includes a capability associated with the called object and that the capability's protected interface authorizes the invocation. If it is not the case, then *PS* returns an error to *C*.
- If the invocation can proceed, *PS* prepares the capabilities to be transferred according to the protected interface that was found and also according to the effective parameters of the call. These capabilities are stored in a list.
- Then, *PS* allocates a new invocation identifier (*id*) which corresponds to the invocation in progress. *PS* registers the association between *id*, the capability which grants access to the object and the list of capabilities to be transferred. The identifier *id* and its associated information are used further by the server in order to authenticate the client and continue the invocation.
- *PS* returns *id* to the client *C*.

Then, the effective invocation of the server is performed by the client stub (message 3 in figure 3). The client stub adds to the message sent to the server the identifier of *C* (for authentication) and the identifier *id*.

The server stub is also modified in order to invoke *PS* before the effective call inside the server. The message sent to *PS* contains the identifier of *C* it received from the client (for authentication) and the identifier *id*. *PS* performs the following operations:

- *PS* checks if *id* is valid. More precisely, *PS* has to verify that both *C* (transmitted by the server) and *S* (the server that called *PS*) are those included in the capability associated with *id* (it prevents a client or a server to bypass protection). If it is not the case, then *PS* returns an error to *S*, which may also return an error to *C*.
- *PS* installs in *S* protection domain the capabilities included in the list associated with *id*.
- *PS* returns to the server *S*.

Then, the effective object invocation is performed in the server.

When returning from the invocation, a similar scheme is used. More briefly:

- *S* invokes *PS* in order to prepare returned capabilities (a list is again associated with *id*). *PS* checks that *S* is the right server.
- *S* returns to *C*, passing *id*.
- *C* invokes *PS* in order to complete returned capabilities transfer. *PS* checks that *S* is the right client.
- *C* returns from the invocation.

The protection model has been implemented on top of Orbix on PC 486 machines running Windows 3.1. It has been validated using the application described in section 3.3.1.

3.3. Another Implementation Considered

Even if the purpose of our prototype was validation, an important drawback is the number of messages required for each invocation. Consequently, we have considered another possible implementation that we now survey.

The idea is to forward invocations from *PS* to the server.

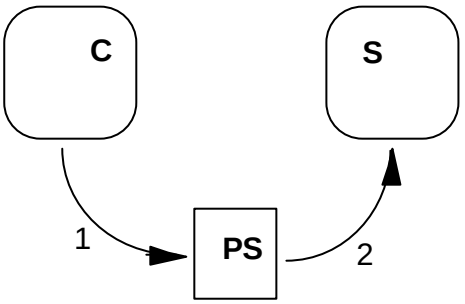


Figure 4. Another possible implementation on Orbix

The principle for authentication is similar but a little simpler. *S* has to verify that it always receives messages from *PS* and *PS* verifies that it receives messages from the right server.

The invocation descriptor identifier *id* is passed to *S* and used again on return.

In order to avoid having to bind (statically) all the client stubs in *PS* statically, we must use in this implementation the dynamic invocation mechanisms through the DII (Dynamic Invocation Interface). The client code creates a request dynamically by calling the DII interface. Therefore, no stubs are necessary in *PS*.

However, we were not able to implement this solution since we used Orbix on Microsoft Windows which doesn't support multithreaded servers. Using a monothread *PS* implies that only one invocation can be in progress in *PS* at a time.

4. Related Work

It is very difficult to relate this work to similar experiments since we are not aware of any implementation of an access control service on a CORBA system[2].

More generally, Hidden Software Capabilities can be compared to capability-based protection schemes that were proposed in some object oriented distributed systems.

The Amoeba system [Tanenbaum 86] is based on the client-server paradigm and its protection relies on software capabilities. However, in Amoeba, capabilities are used explicitly to invoke objects managed by a server. Capabilities are also explicitly exchanged as parameters when an invocation takes place between two servers.

Another system in which protection is based on software capabilities is the Opal system [Chase 94]. Opal is a distributed shared memory system in which memory segments may be shared between protection domains (processes). Capabilities are used in the application code to attach (i.e map) segments in protection domains when they are addressed; a process uses *portals* (cross-domain capabilities) in the same way to enter a new domain, just like an RPC call.

Therefore, in both systems, the definition of protection is wired in the application code. This drawback - managing capabilities in the application code - has already been partially addressed in a few systems. For instance, in Opal, it is possible to force a trap into the system when an object is not yet mapped in the current protection domain, and to have the object automatically mapped if a capability for that object is associated with the protection domain. Furthermore, Opal allows a portal to be hidden from applications behind a proxy object [Shapiro 86]. However, with this approach, the same object cannot, with one protection setting, be called locally and, with another setting, be invoked in another protection domain. Moreover, capability parameters passed to the target protection domain must be specified in the application code. Opal's designers claim that their system can be used as a support for an object based system [Chase 92]. We believe *Hidden Software Capabilities* could be implemented on top of Opal.

The overall goal of CORBA is to manage interoperability between different object oriented languages and runtime systems. Our proposal is well integrated in CORBA since it is language and application code independent.

---

[2] OMG provides a document addressing the security issues in CORBA [OMG 96]. We did not find in this paper any mechanism that may be compared with our proposal.

5. Conclusion and Perspectives

We have presented a novel protection model based on *Hidden Software Capabilities*. In this protection model, object servers are managed in protection domains and may only invoke other servers' objects for which a capability exists in the domain. In order to control capability parameters exchanges between domains, an extended IDL is used to define a protected interface associated with each capability. This protected interface specifies the capabilities that should be passed with object name parameters.

The main advantage of the model is that it separates protection definition from implementation code, thus enhancing modularity and flexibility.

In order to validate our ideas, we implemented this protection model on top of the Orbix system and experimented with simple protected distributed applications. The implementation is based on a protection server in which all the capabilities are safely managed. In terms of message exchanges, the prototype is rather inefficient, but as pointed out in section 3.3, it will be straightforward to improve this prototype using the DII mechanism.

The perspectives of continuation of this work are two-fold. First, we have not yet explored many possible extensions to the IDL for application protection. For example, it will be interesting to experiment with extensions for administrating access rights in protection domains (as proposed in section 2.5). Second, we are currently experimenting with *Hidden Software Capabilities* applied to a Java environment [Hagimont 97]. The idea is basically to allow servers to execute untrusted mobile agents written in Java. Access to objects managed by a server is controlled by capabilities that are invisible to agent programs.

Acknowledgments:

P. Dechamboux, J. Han, T. Jacquin, C. Jensen, A. Knaff, E. Pérez-Cortés, X. Rousset and F. Saunier contributed to the design of the Hidden Software Capability protection model. We also would like to thanks R. Balter and S. Krakowiak.

This work was partially supported by CNET (France Télécom).

Bibliography

[Chase 92] J. S. Chase, H. M. Levy, E. D. Lazowska, M. Baker-Harvey. Lightweight Shared Objects in a 64-Bit Operating System, *Proc. of the 7th ACM Conference on Object-Oriented Programming Systems, Languages and Applications*, 27(10), pp. 397-413, October 1992.

[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.

[Dechamboux 96] P. Dechamboux, D. Hagimont, J. Mossière, X. Rousset de Pina. The Arias Distributed Shared Memory: An Overview, *Lecture Notes in Computer Science*, N° 1175, pp. 56-73, November 1996.

[England 75] D. M. England. Capability Concept, Mechanism and Structure in System 250, *RAIRO-Informatique (AFCET)*, Vol 9, pp. 47-62, September 1975.

[Hagimont 94] D. Hagimont. Protection in the Guide Object-Oriented Distributed System, *Proc. of the 8th European Conference on Object-Oriented Programming*, pp. 280-298 , July 1994.

[Hagimont 96] D. Hagimont, J. Mossière, X. Rousset, F. Saunier. Hidden Software Capabilities, *Proc. of the 16th International Conference on Distributed Computing Systems*, May 1996.

[Hagimont 97] D. Hagimont, L. Ismail. A Protection Scheme for Mobile Agents on Java, *3rd ACM/IEEE International Conference on Mobile Computing and Networking (To Appear)*, September 1997.

[Iona 94] Iona Technologies. Orbix distributed object technology - programmer's guide, Version 1.2, February 1994.

[Kowalski 90] O. C. Kowalski, H. Härtig. Protection in the BirliX Operating System, *Proc. of the 10th International Conference on Distributed Computing Systems*, pp. 160-166, May 1990.

[Levy 84] H. M. Levy. *Capability-Based Computer Systems*, Digital Press, 1984.

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

[OMG 96] OMG. CORBA Security, OMG Document Number 96.08.03-06, Version 1.1, July 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.

[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.

[Wilkes 79] M.V. Wilkes, R. M. Needham. *The Cambridge CAP Computer and its Operating System*, North Holland, 1979.

[Wulf 74] W.A. Wulf, E. Cohen, W. Corwin, A. Jones, R. Levin, C. Pierson, F. Pollack. Hydra: The Kernel of a Multiprocessor Operating System, *Communications of the ACM*, 17(6), June 1974.