An Access Control Scheme for Distributed Object Systems

Vincent Nicomette Yves Deswarte

LAAS-CNRS & INRIA
7, Avenue du Colonel Roche
31077 Toulouse Cedex - France
Telephone: +33/61 33 62 88 - Fax: +33/61 33 64 11
(nicomett@laas.fr, deswarte@laas.fr)

Abstract

This paper describes a new access control scheme for distributed object-oriented systems. The scheme we propose consists in separating two levels of access control: a global authorization server responsible for managing persistent access rights for persistent entities of the system and a local security kernel responsible for checking all local accesses. In the first part of the paper we describe the role and the structure of the authorization server as well as the behavior of the local security kernels. In the second part of the paper, by means of an example, we show how such an access control scheme can be used in distributed object systems.

Technical area: Fault Tolerance, Availability and Security.

1 Authorization in distributed object systems

In an object-oriented system, each entity is represented as an object. Each object is made up of private state information and a set of operations which are the object interface. The state of the object is represented as a set of fields or *attributes*. The values of these fields

at any point during system execution determine the state of the object [1]. The operations defined on objects are called *methods* and are the only way to modify the state of the object or to get information on this state. The invocation of a method leads to the execution of the corresponding operation. The communication between objects consists in sending messages. An object O calls a method of an other object O' by sending O' a message. This message consists of a method selector and a list of arguments. Objects sharing the same structural and operational characteristics are classified into classes. A *class* is a way of describing the behavior of a family of objects. Each object has a class it belongs to. Inheritance is a mechanism for deriving new classes from existing classes by a process of refinement. A derived class inherits the data representation and operations of its parent class but may selectively add new operations, extend the data representation or override the implementation of inherited operations. In this paper, we will call *descendant* of a class X a class which derives directly or indirectly from X.

A distributed object system is a collection of cooperating objects whose locations and communications are transparent for the user. All these objects collaborate to achieve the various user requests but they may be quite different: some objects are persistent, coarse-grained objects of the system and are directly known by the users; some objects are transient objects which may exist just a few seconds. As a matter of fact, in order to perform a task, an object O called by a user S may have to invoke methods on other persistent objects but may also create some objects to delegate them a piece of the task (their lifetime is the duration of the action they realize). Furthermore, all these objects have to communicate (possibly through the network) and do not trust each other.

The access control management is usually based on the concept of active entities (subjects) making accesses to passive entities (objects). In this usual view, all the accesses from subjects to objects must be controlled by a central *reference monitor* [2] which checks if each access is to be authorized or denied. In distributed systems, such centralized access control management must be reconsidered: the number of subjects and thus the number of accesses can be very large and the reference monitor becomes a bottleneck from the performance point of view. Furthermore, this reference monitor is a *single point of failure* since the security of the whole system relies on this unique element.

On the other hand, in the object programming model, the notion of object covers both

active and passive entities. Furthermore, all the accesses to objects are usually made by sending messages which realize information flows. Thus, the access controls management in object systems is obviously different from the access controls management in usual centralized systems. While many authorization schemes have been proposed for the protection of information in operating systems and in data base management systems, the application of these models in object-oriented systems is not straightforward. The particular characteristics of object-oriented systems such as inheritance or composite objects introduce new protection requirements [3]. Several papers [4, 5, 6, 7, 8, 9] have presented some authorization schemes for object systems but these papers mainly address authorization in object-oriented database systems. Our point is to present an authorization scheme which takes into account the characteristics of the object model (i.e., encapsulation, cooperation through method invocation, dynamic aspects) as well as the distribution of the objects.

Our authorization scheme is based on a global protection by an authorization server and on a local protection on each site of the system. The authorization server allows to manage in a consistent way the access rights to persistent objects of a protection domain. Its role encompasses some of the object services defined in the CORBA architecture such as the Naming service for example [10]. From another point of view, the authorization server may be considered as the entity responsible for providing access right management in a domain. The Ticket Granting Server of Kerberos v5 [11] and the Privilege Attribute Service of SESAME [12] are examples of such authorization servers but with a very limited authorization scheme. In Kerberos as well as in SESAME, most of the access controls are realized by the application servers. This solution is well adapted to a client-server scheme but is not flexible enough to manage efficiently access rights for cooperating objects, independently of their locations. The authorization server presented in this paper aims at controlling consistently all relations between persistent objects. This management is in fact very simple because it only takes into account high level actions that can be realized among persistent entities of the system; it corresponds to a global view of the system as a set of coarse-grained, persistent objects. On each site of the system, a security kernel is responsible for controlling accesses to all objects (persistent or transient) located on its site. Each security kernel also manages the access rights for local transient objects.

The purpose of this paper is first to expose the relations between the two levels of

protection (authorization server and security kernel). In Section 2, we describe the role and the structure of the authorization server responsible for enforcing the global protection of the system. We also describe the role of the security kernels which enforce the local protection and we finally present an example of such an authorization scheme in Section 2.6[1].

2 Discretionary access controls

2.1 Definitions and principles

2.1.1 Subject and activity

First let us explain the notion of *subject* and *activity* that we use in the rest of the paper. We call *subject* each persistent entity of the system which has rights on persistent objects of the system. A subject may be a user but may also be a server (a file server for example). An *activity* consists in a succession of method executions through different objects of the system. This set of method executions are functionally dependent and collaborate to achieve a high-level action in the system (e.g. printing a file on a printer). This notion can be viewed as a generalization of the activity described in [13]. In the figure 1, an activity is represented. This activity realizes a high-level action: recording a scene. This activity consists in:

1. Starting the execution of the method *Record-Scene* of the object *Client*,
2. Making a request to the method *Record* of the object *Recorder*,
3. Starting the execution of the method *Record* of the object *Recorder*,
4. Making a request to the method *Film* of the objet *Movie-Camera*,
5. Executing the method *Film* of the object *Movie-Camera*,
6. Returning to the method *Record*,

[1] In this simple example, the distribution is limited to a protection domain. In the conclusion, we will explain how we can extend this scheme to wide-area networks.

7. Returning to the method *Record-Scene*.

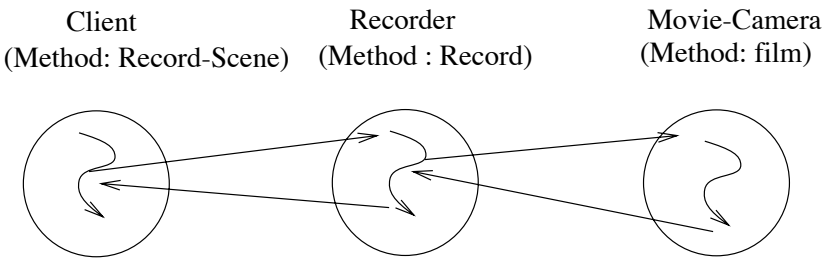

Figure 1: An activity

This notion of activity is obviously a recursive notion. An activity may fork other activities and may have many ramifications.

2.1.2 Request and capability

Each method call is realized by means of a *request*. The object which calls the method is the sender of the request. The object whose method is invoked is the receiver of the request. Each request must carry a *capability* which is a proof that the sender of the request is authorized to access this method.

2.1.3 Method right and symbolic right

In the rest of the paper, we will regularly use the notions of *method right*, and *symbolic right*. A *method right* is a right to invoke a particular method of a particular objet. A *symbolic right* is a high-level right which is used in order to authorize the realization of a high-level operation in the system (e.g., the right to print file f on any printer of the system is a symbolic right). Several symbolic rights are in general needed in order to authorize the realization of a high-level action.

2.2 Role of the authorization server

The authorization server (AS) handles two categories of requests: requests that correspond to the invocation of a particular method of a persistent object of the system and requests that correspond to high-level actions which involve the collaboration of several objects of

the system (such a request corresponds to the execution of a new activity as defined in the previous section).

The AS is given the responsibility of checking that these accesses are authorized. In the first case, the AS checks that the caller has the right to invoke the particular method (the corresponding method right). In the second case the AS checks that the caller has a set of symbolic rights which allows it to realize the task. If the access is finally authorized, the AS must give the caller a capability which allows it to access the method in the first case and to start the action in the second case.

2.3 Authorization server structure

The AS manages an access matrix and a set of *symbolic right rules* and *capability creation rules*. All this information is stored by means of a *directory server*.

2.3.1 Access matrix

The AS access matrix is analogous to Lampson's matrix [14]. It is used by the AS to store the method rights and the symbolic rights. In the matrix, the rows represent subjects' identifiers or *roles* and columns objects (i.e., instances) or classes.

A *role* represents a set of subjects having the same duty (e.g., Administrators, Laser-Printers). In fact, a role is a subject attribute and a subject may have several roles corresponding to different duties. Roles are used as matrix entries instead of subjects' identifiers to reduce the size of the access matrix and to simplify it. Indeed, access rights to an object or a class are often defined for groups of subjects (i.e., roles) rather than for every subject of the system. Roles are a flexible way to gather and to manage subjects. In the same way, the notion of class allow to define access rights to a set of objects (all the instances of the class) and allows to simplify the management of the access matrix.

The access rights that a subject (or a role) holds for an object (or a class) can be found at the intersection of the subject row and the object column (holding a right for a class signifies holding a right for any instance of the class). The two right families (method rights and symbolic rights) are expressed in two different ways:

- The method rights that a subject S holds for an object O are expressed by the list

of the corresponding methods of O. For instance, if $M_{S,O} = (f, g)$ then the subject S has the right to invoke methods f and g of O and is not authorized to invoke any other method of O.

- The symbolic rights are expressed by: *action(list)*. *Action* refers to a high level operation in the system and *list* is a set of objects or classes. The symbol *this* in the list represents the object or the class associated with the column where the symbolic right can be found in the matrix. These symbolic rights are interpreted according to the symbolic right rules. An example of such a symbolic right is given hereafter in Section 2.3.2.

2.3.2 Symbolic rights rules

Each symbolic right rule describe the set of symbolic rights that must be found in the access matrix in order to authorize an access corresponding to a high-level action. Let us give an example of such a symbolic right rule:

$$\begin{aligned} recordscene(recorder, camera) :- (\exists\, Class\ X,\ recordscene(recorder, X) \wedge \\ Class(camera) \in\ SubClass(X))\ \wedge \\ (\exists\, Class\ Y,\ recordscene(Y, camera) \wedge \\ Class(recorder) \in\ SubClass(Y)) \end{aligned}$$

With $SubClass(X) = \{X\} \cup \{Z,\ Z\ is\ descendant\ of\ X\}$.

$recordscene(recorder, camera)$ is a boolean value. If it is true, the action "recording a scene from the movie-camera *camera* with the recorder *recorder*" is authorized; if it is false, the action is not authorized. This boolean is true for a subject S only if both the following symbolic rights can be found in the access matrix for S:

- the right to record a scene with the recorder *recorder* from a set of movie-cameras which includes *camera*.

- the right to record a scene from the movie-camera *camera* with a set of recorders which includes *recorder*.

An example of such symbolic rights in the access matrix is presented below ("*" stands for any object which the symbolic right applies on).

	recorder	camera	...
S	*recordscene(this,*)*	*recordscene(*,this)*	...
recorder		film	...
...	...	...	...

In this representation, symbolic rights are written in italics while method rights are in plain text.

2.3.3 Capability creation rules

When a subject is authorized to realize a particular operation (either accessing one method of one object or executing an action involving several objects), it must be given a capability which allows it to make this operation. The *capability creation rules* specifies how to build such capabilities. These capabilities are to be checked by the security kernel located on the invoked object site (see Section 2.6). These capabilities are similar to those defined in [15, 16]. A capability for an object holds at least a reference and a list of rights for that object.

In order to make all the communications secure, the system enforces controls which ensure that a capability cannot be forged, cannot be replayed and that a capability can only be used by the subject the capability is created for. We assume in the rest of paper that all these properties are implemented by the capability creation and verification algorithms.

2.3.4 Directory server

An authorization server is actually composed of two parts: the actual authorization server which checks the rights and creates the capabilities, and the directory server which stores all the necessary information related to objects and subjects.

Each entry of the directory possesses a set of attributes necessary to perform the authorization controls. If the entry describes a subject, the attributes include at least:

- his name (e.g., {Name=Tom}),
- his role(s) (e.g., {Role=Professor,Administrator}).

If the entry describes an object such as a file or a printer, attributes include:

- its name (e.g., {Name=Laser1}),
- its class, (e.g., {Class=LaserPrinter}),
- its methods, (e.g. {methods=Print}),
- for each method, a capability template (e.g., {Captemp=($R_{print,LaserPrinter}$, $Laser1$)}),
- an access-control list (e.g., {ACL=(Tom, printfile(*, this)), (Administrator, printfile(Manuals, this))}),
- for each symbolic right applying to the object, a symbolic right rule and a capability creation rule.

Other attributes may exist according to the class of the object. For instance a printer entry has an attribute which is the name of its print server (e.g., PrintServer="ps1").

The access matrix is actually stored by the directory server as ACLs (access-control lists) attached to object entries. An ACL corresponds to a column of the access matrix and defines the access rights of subjects on the object. In the example above, *Tom* can print any file on *Laser1* while any subject with the *Administrator* role can only print manuals on *Laser1*. For each subject or role allowed to perform an operation on an object, there is an entry (name, access rights) in the object ACL. If the right is a symbolic right, the corresponding symbolic right rule and capability creation rule are stored in other object attributes. In this case, the same symbolic right rule and capability creation rule appear in each object entry that participates in the same high level action. Symbolic right and capability creation rules are thus replicated in different places of the directory server. However, this replication is limited to the number of objects involved in a rule and so the cost is not really high.

2.3.5 Proxies

A *proxy* is a right that a subject S gives to another subject S' so that S' can realize a task in place of S. This notion is quite usual and is also used in Kerberos v5 [11] as well as in SESAME [12]. A proxy is composed of a right and a subject identifier. This means that the proxy represents a right which can be used by the subject whose identifier is inserted in the

proxy. In our authorization scheme, a proxy may be inserted in a capability. The subject who receives such a capability will transmit the proxy to the subject whose identifier is inserted in the proxy. The subject who receives a proxy will be authorized to realize the operation corresponding to the right inserted in the proxy but will execute this operation with its own identity. This solution does not present the drawback of the SUID bit in Unix which allows a subject to become another subject while executing an operation, which means that the subject is granted all the access rights of another subject to realize an action which does not necessarily require all these rights. Such an excess of privilege can be exploited by a malicious user, and this is a major security flaw on Unix.

2.3.6 Summary of typical scenarios

When a subject makes a request to the AS, there are two possible scenarios:

- If the request corresponds to a method right in the access matrix then the AS gives the caller the capability corresponding to the invoked method.
- If the request corresponds to a high-level action on the system, then the AS must realize the following operations:
  - The AS must check that either the access matrix holds the symbolic rights that authorize this action according to the symbolic right rules or the request holds a valid proxy (i.e., a proxy that has been created for the subject who presents it, a proxy that corresponds to the operation the subject wants to realize and that this proxy has been created by the AS).
  - If the action is authorized, the AS must return to the subject which method of which object it has to invoke to perform the action (to start a new activity), with the corresponding capability[2]. This information is obtained by the AS through the use of the capability creation rule and other attributes stored in the directory server. A detailed example is given in section 2.6.

[2]A more efficient implementation would consist in the AS invoking directly this method of this object on behalf of the caller. Such an implementation would enforce the reference monitor role of the AS, but would be a little more complex.

2.4 Management and security of the authorization server

Regarding the insertion and management of access rights and rules in the AS, we can think about different security policies: as in Unix for example, the creator of an object may autonomously manage the authorizations on his own objects and thus may insert or delete access rights in the access matrix. Conversely, we can imagine granting and revocation of authorizations to be a task of some privileged users, called administrators. The management of the AS is a problem that is beyond the scope of this paper. We wish to focus on the mechanisms that we use (in the AS and in the security kernels) to realize access controls in distributed object systems. The management of the granting and revocation of authorizations is closely linked to the way such a system is used.

A centralized AS would be a point of failure in our distributed system. But it is possible to implement the AS in such a way that an accidental fault or an intrusion would not impede its security. A particular fault tolerance technique, called Fragmentation-Redundancy-Scattering has been successfully experimented to implement such a security server [17]. The approach consists of gathering most of the security functions of the distributed system into a set of specialized sites, the security sites, responsible for subject authentication and authorization. These security sites constitute a distributed security server which can be globally trusted, even if no individual site is trusted: an intrusion into a minority of the security sites is tolerated because it has no consequence on the confidentiality or integrity of the security management data and no consequence on the availability of the overall security service.

2.5 Local protection

A local security kernel (SK) located on each site of the system is responsible for checking all accesses to each local object whether persistent or dynamic. When a local object is accessed, the check is realized as follows: the request must carry a capability which authorizes this access; this capability must correspond to the method and the object which are invoked and must be valid for the caller (the identifier of the caller must be present in the capability).

Each security kernel is also responsible for building capabilities for local transient ob-

jects. For instance, when an object O creates locally a transient object O' in order to make it realize a particular task, O receives from the local security kernel (SK_i) a particular capability which is the *owner* capability. By using this capability, O is authorized to access all the methods of O'[3]. Furthermore, O can ask SK_i to build capabilities to grant another object O'' rights to access some methods of O'.

This notion of security kernel is quite usual [18]. The TMach security kernel for instance enforces rules to control all accesses to local objects (ports and tasks) it manages [19].

2.6 Example

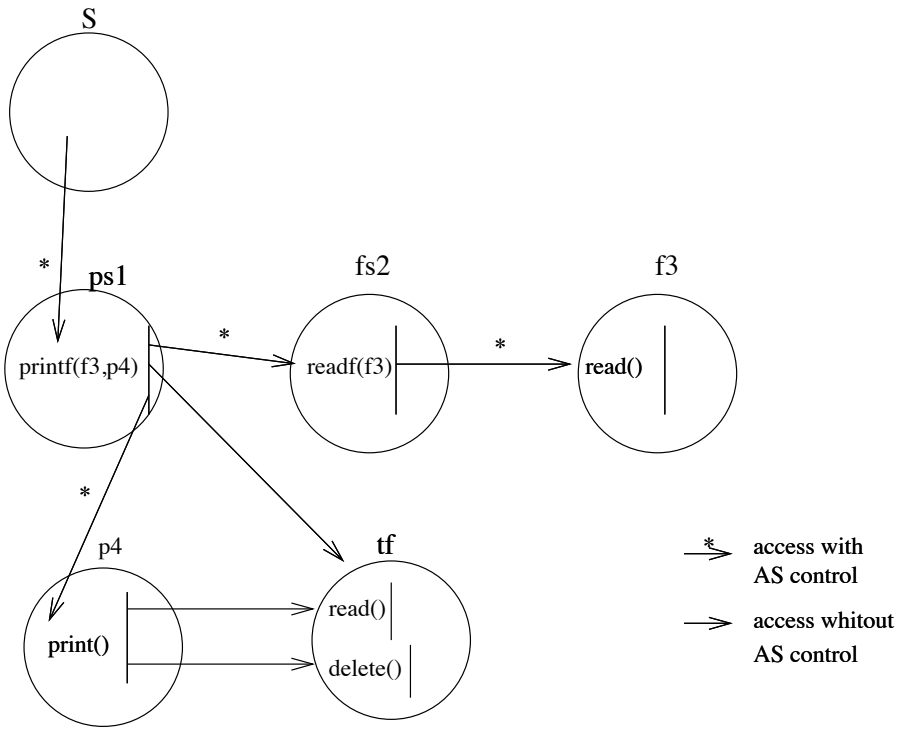

Figure 2: Objects cooperation

In the following example, we consider that a user S wants to print file f_3 on printer p_4. The objects which take part in the execution of this action are represented in Figure 2. S is a subject of the system, ps_1 a print server of class *PrintServer*, fs_2 a file server of class

[3]By using the *owner* capability, O can also delete O'. 'Delete' is considered as a particular method of the object.

FileServer, f_3 a file of class *File*, p_4 a printer of class *Printer* and tf a transient file located on the site of ps_1.
The different components of the AS are presented below: the access matrix, the symbolic right rules and the capability creation rules. Then, we present how this information is stored by the directory server.

2.6.1 Access matrix

	ps1	fs2	f3	p4
S			*readfile(this),printfile(this,*)*	*printfile(*,this)*
ps1				print
fs2			read, write	

Definition of the symbolic rights:

$readfile(f) : (Class(f) \in SubClass(File))$;
$printfile(f, p) : (Class(f) \in SubClass(File)) \wedge (Class(p) \in SubClass(Printer))$;
With $SubClass(X) = \{X\} \cup \{Z,\ Z\ is\ descendant\ of\ X\}$.

The first row of the access matrix represents symbolic rights that are granted to subject S in order to access persistent objects of the system. The symbolic right *printfile(this,*)* in column f_3 means that S is authorized to print file f_3 on any printer of the system. The symbol * refers to all the printers of the system, but we could have indicated a more restrictive set of printers, by means of a list of printer names or subclasses of class *Printer*, e.g., *printfile(this,Class=LaserPrinter)*. Thus, *printfile(this,*)* is equivalent to *printfile(this,Class=Printer)*. In the same way, *printfile(*,this)* in column p_4 means that S is authorized to print any file of the system on printer p_4 and is equivalent to *printfile(Class=File,this)*. Finally, the right *readfile(this)* in the column f_3 means that S is authorized to read file f_3.

The last rows of the access matrix represent method rights that ps_1 and fs_2 have respectively on p_4 and f_3: ps_1 is authorized to access method *print* of p_4 and fs_2 is authorized to access methods *read* and *write* of f_3.

The three lines following the description of the matrix indicate that the symbolic right $printfile$ is defined to apply to particular classes: the first parameter must be an instance of class *File* or an instance of a descendant of class *File*. The second parameter must be an instance of the class *Printer* or an instance of a descendant of class *Printer*. In the same way, $readfile(f)$ is a symbolic right applying to an instance of class *File* or of a descendant of *File*.

2.6.2 Symbolic right rules

The following rule must be included in the symbolic right rule set:

(SR1) $printfile(f,p) :- (\exists\, Class\ X,\ printfile(f,X)\ \wedge\ Class(p) \in\ SubClass(X))$
$\wedge\ (\exists\, Class\ Y,\ printfile(Y,p)\ \wedge\ Class(f) \in\ SubClass(Y))$
With $SubClass(X) = \{X\} \cup \{Z,\ Z\ is\ descendant\ of\ X\}$.

This rule means that in order to authorize a subject to realize the high-level action "printing file f on printer p" ($printfile(f,p)$ is true), the AS must check that the subject holds in the access matrix both the symbolic rights:

- *printfile* for file f on a set of printers which includes p.
- *printfile* on printer p for a set of files which includes f.

In the rest of the paper we will often designate a high-level action by its corresponding boolean. We will for instance talk about the high-level action $printfile(f,p)$ or $readfile(f)$.

2.6.3 Capability creation rules

This set of rules includes at least the following rules:

(PR1) $\forall f\ instance\ of\ File,\ \forall p\ instance\ of\ Printer,\ Cap(printfile(f,p)) ::=$
$(Cap(\mathsf{PrintServer}(p).printf(f,p)),\ [\mathsf{PrintServer}(p), readfile(f)])$

(PR2) $\forall f\ instance\ of\ File,$
$Cap(readfile(f)) ::= Cap(\mathsf{FileServer}(f).readf(f))$

(PR3) $\forall ps\ instance\ of\ PrintServer,$
$Cap(ps.printf(f,p)) ::= ((R_{printf,PrintServer}, ps), ref_f, ref_p)$

(PR4) $\forall fs\ instance\ of\ FileServer,$

$Cap(fs.readf(f)) ::= ((R_{readf,FileServer}, fs), ref_f)$

(PR5) $\forall f$ *instance of* $File$,

$Cap(f.read()) ::= (R_{read,File}, f)$

(PR6) $\forall p$ *instance of* $Printer$,

$Cap(p.print()) ::= (R_{print,Printer}, p)$

Each called method of a persistent object managed by the AS is linked to a capability which is to be given to the subject which is authorized to access this method. In our example, $R_{print,Printer}$ in (PR6) is a class capability. It must be associated with the instance of the class *Printer* that the subject wants to invoke. In this way, $(R_{print,Printer}, p_4)$ is the capability which allows to access method *print* of printer p_4. ref_O represents a reference which allows to find the location of object O.

Each high-level action is also linked to a capability to access a method of a persistent object. As a matter of fact, when a subject is authorized to realize a high-level action in the system, it must start this action by invoking a particular method of a particular object. The AS thus gives it the corresponding capability. The first part of Rule (PR1) means that the capability corresponding to the action *printfile(f,p)* is the capability to access method *printf* of the print server of printer p, while rule (PR2) indicates that the capability corresponding to the action *readfile(f)* is the capability to access method *readf* of the file server of file f. The notations PrintServer(p) and FileServer(f) refer to information that is stored in the directory server indicating which printer is the print server of p and which file server is the file server of f.

Furthermore the capability linked to an action may hold another right that has to be delegated to another object of the system. For instance, in the capability linked to the action $printfile(f, p)$, the proxy $[\mathsf{PrintServer}(p), readfile(f)]$ represents a symbolic right valid for the print server of printer p[4]. This proxy is inserted in the capability returned to the subject authorized to realize the action $printfile(f, p)$. This subject will pass the proxy on to the print server of p when he accesses its method $printf$.

In our example, when user S makes his first request, he sends a message to the AS. This

[4]Let us note that this proxy creation does not require that S holds the symbolic right *readfile* on f_3 in the access matrix (this is not checked by the symbolic right rule corresponding to *printfile*).

message is a request for getting the capability corresponding to the action $printfile(f_3, p_4)$. The first row of the access matrix indicates that subject S is authorized to print file f_3 on any printer of the system and that subject S is authorized to print any file on printer p_4. Symbolic right rule SR1 indicate that the access is authorized. A capability is then built according to the rules of the capability creation rules. This capability is constituted of the capability to access method *printf* of ps_1 and of a proxy for ps_1 which corresponds to the action $readfile(f_3)$.

2.6.4 Directory server

The directory server stores in a tree structure all the necessary information related to the objects and the subjects (see figure 3). All the methods of an object are stored with their capability creation rules. Thus, in our directory server, the entry f_3 possesses an attribute representing all the methods of f_3 with their corresponding capability creation rule. This attribute is *Methods*. In the same way, the capability creation rules corresponding to high-level actions are stored in an attribute for each object referred in the symbolic right. This attribute is *CapabilityRule*. The symbolic right rules which refer to the object are also stored in the attribute *SymbolicRule*.

The attributes *PrintServer*="ps_1" and *FileServer*="fs_2" indicate that the print server of printer p_4 is ps_1 and that the file server of file f_3 is fs_2. This information is used by the capability creation rules to build the capability that subject S receives.

2.6.5 Complete scenario

The scenario which enables subject S to print file f_3 is the following one:

- Subject S wants to print file f_3 on printer p_4. The AS first checks that this high level action is authorized: the access matrix holds the symbolic rights $printfile(f_3, *)$ and $printfile(*, p_4)$. Symbolic right rule (SR1) authorizes this access:
  $printfile(f_3, p_4) = printfile(f_3, *) \wedge printfile(*, p_4)$.

- According to capability creation rule (PR1), the AS gives subject S the capability $((R_{printf,PrintServer}, ps_1), ref_{f_3}, ref_{p_4}, [ps_1, readfile(f_3)])$. The AS also indicates sub-

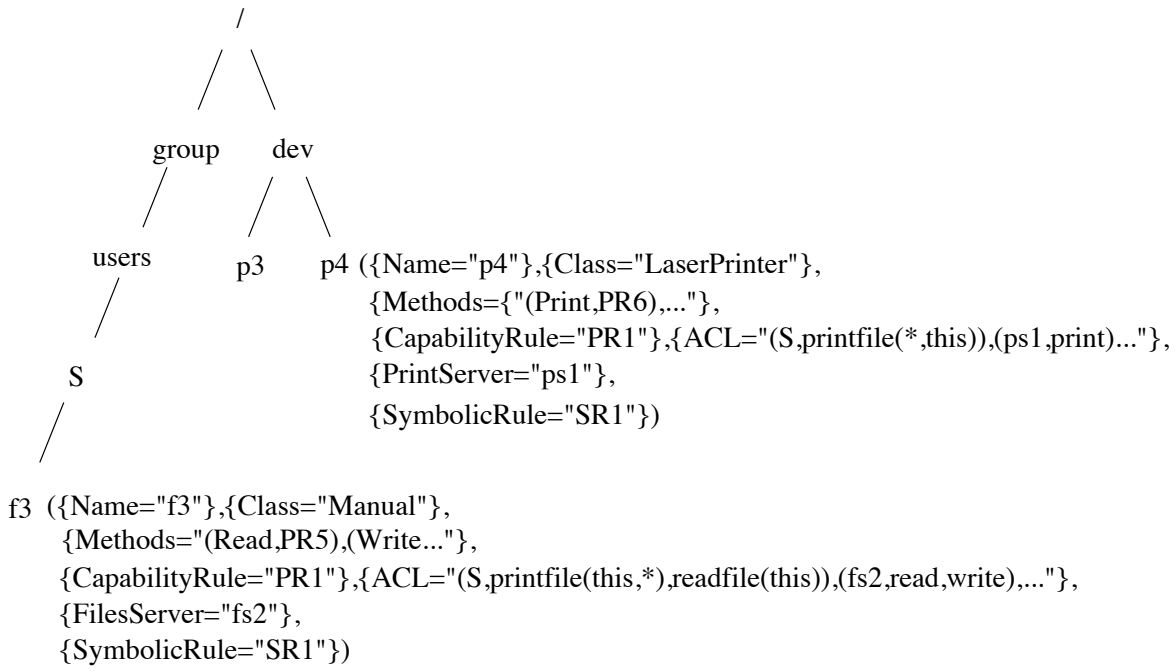

Figure 3: The directory server

ject S the object and the method that he can invoke with this capability (i.e., ps_1 and $printf$)[5].

- S calls method $printf$ of ps_1 and presents the capability $(R_{printf,PrintServer}, ps_1)$. This invocation is controlled by the security kernel SK located on the site of ps_1 which checks that the capability is valid. Then S gives ps_1 the proxy $[ps_1, readfile(f_3)]$.

- ps_1 asks the AS for executing the action $readfile(f_3)$ and presents the proxy received from S.

- The AS first checks that the proxy is valid (i.e., is a right for ps_1 created by the AS). Then the AS identifies fs_2 as the file server for f_3, and finally gives ps_1 the capability corresponding to the action $readfile(f_3)$ which is $((R_{readf,FileServer}, fs_2), ref_{f_3})$ (according to capability creation rule (PR2)). The AS also indicates ps_1 the object and the method it can invoke with this capability (i.e., fs_2 and $readf$).

[5]The AS may itself realize the access to ps_1 instead of giving the capability to S who makes the access. See footnote 2 on page 9.

- ps_1 calls method *readf* of fs_2 and presents $(R_{readf,FileServer}, fs_2)$. The SK located on the site of fs_2 then checks the validity of $(R_{readf,FileServer}, fs_2)$.

- fs_2 asks the AS for a right to access method *read* of file f_3.

- The AS checks that fs_2 is authorized to access method *read* of f_3 and gives fs_2 the capability $(R_{read,File}, f_3)$.

- fs_2 invokes method *read* of file f_3. The SK located on the site of f_3 checks the validity of the capability $(R_{read,File}, f_3)$.

- ps_1 creates a temporary file tf which is to be used by p_4 as a spooler. When creating tf, ps_1 receives the *owner* capability on this file from the local SK. Then ps_1 copies f_3 into tf (the write access to tf is authorized because ps_1 presents the *owner* capability to the security kernel).

- ps_1 asks the AS for a capability to access method *print* of p_4.

- The AS checks that ps_1 is authorized to access method *print* of p_4 and gives ps_1 the capability $(R_{print,Printer}, p_4)$.

- ps_1 calls method *print* of p_4, then asks the security kernel to transmit to p_4 the capabilities to access methods *read* and *delete* of file tf (by presenting the local security kernel the *owner* capability).

- p_4 calls method *read* of tf. This invocation is controlled by the local SK which checks that p_4 holds the capability for method read of tf. Then p_4 prints the file. After printing, p_4 sends a request to delete tf.

The point of this simple example is that the access right management of transient objects such as tf is different from the access right management of persistent objects such as ps_1 for example. In our example, the capabilities that allow to access the methods of tf are built and checked by the local SK. The AS cannot be assigned these tasks because it does not even know that this object exists.

We also want to show in this example that the notion of symbolic right is very useful because it allows to add transparency to the access right management and furthermore is

adapted to the management of all the actions that require the collaboration of different objects (which represent most of the operations realized in a distributed object system). This approach gives more flexibility the administration of access rights.

With respect to the security kernels, one of their most important features is their simplicity. A local SK does not store any capability. It builds and gives capabilities to the subjects who want to create local transient objects and controls all local accesses by checking the validity of the capabilities carried by all the requests. It can thus be easily implemented and the simplicity of the realized checks does not impede system performances too much.

3 Conclusion and future work

In this paper, we have presented an authorization scheme for distributed object systems which is based on two main principles:

- An authorization server manages all the persistent rights for the persistent entities of the system.
- A local security kernel on each site of the system controls all local accesses.

The example we have presented shows how such an authorization scheme can be used in the context of a discretionary security policy. But the application of this authorization scheme is not restricted to such policies and can also be used, for instance, in the context of multilevel security policies which aims at protecting the confidentiality of the information. With such a policy, the different entities of the system (subjects, objects and requests) are assigned security levels and rules enforced by the authorization scheme must guarantee that there is no information flow from a high security level towards a low security level [20]. Our authorization scheme can also be easily adapted to other security policies such as integrity policies. In order to enforce an integrity policy such as the Biba policy [21], we have to assign integrity labels to the objects of the system and establish rules that guarantee the integrity of the system by preventing information flow from low integrity levels to high integrity levels.

In order to gain better performance by reducing the number of accesses to the authorization server, we can use a cache mechanism. The persistent objects can cache the capabilities corresponding to persistent rights that they have in the access matrix. They can thus use these capabilities without having to make requests to the authorization server. This mechanism must obviously not be applied to the proxies delegated by other objects in order to realize a task on behalf of them: these proxies are delegated for a particular action in a particular activity and must not be cached.

Another optimization consists in modifying the proxies that are included in the capabilities associated with high-level action. In our example, the proxies are in fact rights (method rights or symbolic rights) that are to be presented to the authorization server in order to obtain the corresponding capabilities. We could imagine the proxies to directly be the capabilities instead of the rights. This would complicate the capability creation rule associated with each high-level action but this would considerably reduce the accesses to the authorization server. One can pretend that this optimization could lead to new security problems as regards to revocation. This is not the case because in the scheme we have proposed, the authorization server does not check that the proxies it receives actually correspond to rights that are still in the access matrix. The revocation problem would thus be the same if we used capabilities instead of rights in the proxies.

One of the main lines of our future work is the extension of this scheme to wide area networks. For such networks (composed of several thousands of sites), it is obviously impossible to manage all the objects and subjects on a single authorization server. In order to manage them, the whole set of objects and subjects is split into domains. Each domain is then managed by a single authorization server which has to control the access to the objects of its domain. The subjects of a domain are authenticated by the authentication server of this domain. The information about objects and subjects of the domain are stored in the corresponding directory server. A directory server can be easily extended and connected to other directory servers. X500 provides a way to do this through aliases [22]. An alias is a symbolic name of a directory which is located elsewhere: in our case on a different directory server. When accessing an object whose name contains an alias, the search is propagated to the other directory server. Thanks to these main concepts, our authorization scheme can be extended to wide area networks without compromising the

security of the whole system.

References

[1] B. Meyer, *Object-Oriented Software Construction*. Prentice Hall, 1988.

[2] "U.S. Departement of Defense Trusted Computer Security Evaluation Criteria (TCSEC)." 5200.28-STD, December 1985.

[3] E. Bertino and P. Samarati, "Research Issues in Discretionary Authorizations for Object Bases," in *Proc. of the OOPSLA-93 Conference Workshop on Security for Object-Oriented Systems* (B. Thuraisingham, R. Sandhu, and T. Ting, eds.), (Washington DC), pp. 183–199, Springer-Verlag, September 1993.

[4] T. Keefe, W. Tsai, and M. Thuraisingham, "SODA: a Secure Object-oriented Database System," *Computers and Security*, vol. 8, no. 6, pp. 517–533, 1989.

[5] T. Lunt, "Multilevel Security for Object-Oriented Database Systems," in *Proc. IFIP WG 11.3 Workshop on Database Security* (D. Spooner and C. Landwher, eds.), (Monterey, California), pp. 199–209, North-Holland, September 1989.

[6] B. Thuraisingham, "Mandatory Security in Object-Oriented Database System," in *Proc. conf. Object-Oriented Programming Systems Languages and Applications*, (New York), pp. 203–210, 1989.

[7] N. Boulahia-Cuppens, F. Cuppens, A. Gabillon, and K. Yazdanian, "Multilevel Security in Object-Oriented Databases," in *Proc. of the OOPSLA 93 Conference Worshop on Security in Object-Oriented Systems* (B. Thuraisingham, R. Sandhu, and T. Ting, eds.), (Washington DC), pp. 79–89, Springer-Verlag, September 1993.

[8] N. Boulahia-Cuppens, F. Cuppens, A. Gabillon, and K. Yazdanian, "Multiview Model for Object-Oriented Databases," in *Proc. of the ninth Annual Computer Security Applications Conference*, (Orlando, Florida), December 1993.

[9] S. Jajodia and B. Kogan, "Integrating an Object-Oriented Data Model with Multi-Level Security," in *Proc. of the 1990 IEEE Symposium on Security and Privacy*, (Oakland, CA), pp. 48–69, May 1990.

[10] S. L. Chapin, W. Herndon, and L. Notargiacomo, "Security for the Common Object Request Broker Architecture (CORBA)," in *Proc. of Computer Security Applications Conference*, (Orlando, Florida), pp. 21–30, December 1994.

[11] J. Kohl and C. Neuman, "The Kerberos Network Authentification Service (V5)," *RFC 1510*, September 1993.

[12] T. Parker, "A Secure European System for Applications in a Multi-vendor Environment (The SESAME Project)," in *Proc. of the 14th National Computer Security Conference, NCSC and NIST*, (Washington), pp. 505–513, October 1991.

[13] J. Banino, J. Fabre, M. Guillemont, G. Morisset, and M. Rozier, "Some Fault-Tolerant Aspects of the Chorus Distributed System," in *Proc. of 5th International Conference on Distributed Computing Systems*, (Denver, Colorado), pp. 430–437, May 1985.

[14] B. Lampson, "Protection," *ACM Operating Systems Review*, vol. 8, no. 1, pp. 18–24, 1974.

[15] A. S. Tanenbaum and al., "Using Sparse Capabilities in a Distributed Operating Systems," in *Proc. of the 6th International Conference on Distributed Computing Systems*, (Cambridge, MA), pp. 558–563, May 1986.

[16] L. Gong, "A Secure Identity-Based Capablity Systems," in *Proc. of the IEEE Symposium on Security and Privacy*, (Oakland, CA), pp. 56–63, May 1989.

[17] Y. Deswarte, J. Fabre, and L. Blain, "Intrusion Tolerance in Distributed Computing Systems," in *Proc. of Symposium on Research in Security and Privacy, IEEE Computer Society Press*, (Oakland, California(USA)), pp. 110–121, may 1991.

[18] S. Ames, M. Gasser, and R. Schell, "Security Kernel Design and Implementation: an Introduction," *IEEE Computer*, pp. 14–22, July 1983.

[19] M. Branstad, H. Tajalli, and F. Mayer, "Security Issues of the Trusted MACH System," in *Proc. of Fourth Aerospace Computer Security Applications Conference*, (Orlando, Florida), pp. 362–367, December 1988.

[20] V. Nicomette and Y. Deswarte, "Discretionary and Mandatory Access Controls for Distributed Object Systems," Tech. Rep. 95261, LAAS-CNRS, June 1995.

[21] K. Biba, "Integrity Considerations for Secure Computer Systems," Tech. Rep. ESD-TR 76-372, MITRE Co., April 1977.

[22] "The Directory: Overview of Concepts Models and Services." ITU Recommendation X500, November 1993.