Use of DSOM Before/After Metaclass for Enabling Object Access Control

Messaoud Benantar, Bob Blakley and Anthony Nadalin
Object Services Technology Center, IBM Corporation
 11400 Burnet Road, Austin, Texas 78758, USA. email: {benantar,
 blakley, anthony}@vnet.ibm.com

Abstract

We review the IBM's System Object Model (SOM) and Distributed SOM (DSOM). Then, we introduce DSOM's approach to object access control and contrast it with traditional procedural systems. Subsequently, we elaborate on the problem addressed in this paper that seeks to enable the process of object access control within DSOM kernel transparently from application developers. We discuss different approaches to solving the problem and present the adopted method. We provide implementation details of the solution and conclude with remarks on future improvement.

Keywords

Distributed system object model, object access control, before method, principal object, secure metaclass, secure class, access control list, access right.

1 INTRODUCTION

Computation in a distributed application, generally, proceeds in the form of a dialog amongst at least two separately executing programs. Each can be making requests for services, at which time it is called a client, or it can be in the state of servicing one or more requests, and at which time it is called a server. As such, client-server applications cross the boundaries of a single address space and communicate with one another, passing information about service requests and receiving results. The client and the server can be running on two different hardware and software systems, remote from each other, and linked through the communication media and protocols of an underlying network. Although this paradigm of computing seems attractive, developing client/server applications still presents some difficulties and challenges that make developing applications on single systems looks much simpler.

Building client-server applications can be achieved through explicit network programming, and thus requiring an application developer, whether in the role of a client or a server, to understand the details of network communication. The Open Software Foun-

dation (OSF) Distributed Computing Environment, or simply (OSF DCE) * is making an attempt to reduce the complexity of building such distributed applications to a degree that is at least nearly comparable to developing applications to run on single systems. OSF defines the elements under which a distributed environment is based upon; including a unique naming service, a security service and a time service that make up the building block for such a distributed environment, called a *cell*. Furthermore, this environment is capable of scaling by composing cells through intercell communication. As it is described in the OSF DCE Application Development Reference (1993), writing applications for DCE is characterized by advertising server interfaces in the naming service; clients then consult such naming service for the server's interface definition and its binding location. Subsequently, remote procedures are triggered on the server in the same way a traditional application would invoke a local procedure call; while the DCE run-time of an application hides the details of network communications.

With the advent of object oriented programming, distributed object oriented applications have naturally evolved; thus, bringing the requirement for a novel technology that distributes and facilitates communication over a network of distributed objects. The Object Management Group (OMG), a consortium of computer software and hardware vendors, is developing interface standards for distributed object computing known as Common Object Request Broker Architecture (1991), known as CORBA. Its Object Request Broker specification, or simply ORB, defines mechanisms under which objects transparently communicate with each other, issue requests for services and receive responses.

IBM's Distributed System Object Model is developed to comply with the OMG CORBA specification for which there are many aspects. One that predates object oriented programming and is also present in DCE, is the Interface Definition Language (IDL) used to define the interface to a server. A client then invokes the methods defined by the server's IDL definition on objects residing in remote servers in the same manner when they are residing in the address space of the client. The ORB run-time mechanism acts as the bridge between the client's application and the server program and, thus, hides the communication details from application developers and users.

To seek standards for building distributed object solutions, the Object Management Architecture (OMA) specification of OMG defines a complete set of Object Services and common facilities that make it useful for building such solutions. These services, some of which are still being developed, include object lifecycle, naming, event, persistence, and security. Parts of this paper relate to the Object Security Service, or OSS, designed and integrated with IBM's DSOM and developed in light of the OMG security requirements and guidelines for OSS as it is defined in the OMG documents of both the White Paper on Security (1994) and the Object Services Request for Proposals (1994). In Section 2 we introduce the fundamentals of SOM and DSOM. In Section 3 we discuss the process of setting up a secure DSOM client-server association, and in Section 4 we outline the DSOM approach to object access control and discuss different methods to enforcing it. In Section 5 we present the SOMMBeforeAfter metaclass that implements the SOM before/after behavior, and in Section 6 we show how we apply SOMMBeforeAfter to enabling DSOM access control. We close up with some concluding remarks in Section 7.

*OSF DCE is a registered trademark of Open Software Foundation, Inc.

2 SOM AND DSOM OVERVIEW

Besides being a system compliant to the CORBA specification, SOM provides key object oriented programming features, as described in the IBM SOMObject Developer Toolkit Users Guide (1994). These include implementation inheritance and polymorphism. Inheritance, or class derivation, allows for developing new classes from existing ones. The derived class, also called sub-class, inherits the functionality and the data defined by the parent methods, in addition the sub-class, usually, introduces new methods and defines data of its own so as to widen the scope of functionality introduced in the parent class. Polymorphism allows for a derived SOM class to redefine the functionality of a method inherited from a parent class, in order to provide for a specialized functionality. In SOM a class is also an object whose class is called a *metaclass*. A class object then, being, an instance of its metaclass, responds to the methods that its metaclass defines. A metaclass, like any other SOM class, can be sub-classed and its methods be overridden.

A SOM program is organized as a set of directed acyclic graphs of classes and subclasses, and computation consists of invoking methods on the object instances created out of this hierarchy. At the root of which we find two basic SOM run-time classes, SOMObject and SOMClass. The former is the root ancestor for all classes defined in a SOM application, whereas the latter is the root ancestor of all SOM metaclasses. A SOM object is an instance of a SOM class, and all SOM classes defined by an application are ultimately derived from SOMObject, including SOMClass. Similarly, all metaclasses of SOM classes are ultimately derived from SOMClass. Because they are classes, both SOMObject and SOMClass are instances of the root metaclass, SOMClass. As such SOMObject class introduces generic methods that, by inheritance, are applicable to all SOM objects of an application. These methods provide basic SOM functionality such as the dispatch mechanism for executing method invocations. Similarly, all SOM class objects respond to the generic methods defined by SOMClass, including a method for creating instance objects of a class, and methods for creating and modifying a class's instance method table as detailed in the IBM SOMObject Developer Toolkit Reference Guide (1994). Figure 1 shows the object model of class hierarchy found in a SOM application.

By providing for the CORBA ORB functionality, Distributed SOM, or DSOM, allows application programs to access objects across address spaces and/or systems. DSOM Object Manager, a run-time DSOM object, transparently provides functionality for locating, creating, using and destroying remote objects by client programs. The client seamlessly creates and invokes methods on remote objects, through the run-time mechanism of the *proxy objects*. These objects act as the local representatives of the target remote objects. Proxy objects are created by consulting the Interface Repository database that the SOM compiler maintains for the interface to the remote object. The Implementation Repository, on the other hand, is used to retrieve information about the actual implementation of the server. This includes its host, its implementation ID, the class of its server object, and the path for loading its executable code. The server side run-time includes three main objects, the server's implementation definition (ImplementationDef), the SOM Object Adapter (SOMOA), and the application specific server object which is generally an instance of the predefined SOMDServer class or one that is user defined and derived from SOMDServer.

The implementationDef object provides the framework interface to the implementation repository. The SOMOA receives client requests and in cooperation with the SOMDServer

Figure 1 *Class hierarchy of a SOM/DSOM application.*

object it creates, resolves DSOM references to local objects, and dispatches methods on those objects. Figure 2 illustrates the basic elements that define a DSOM client/server interaction.

3 ESTABLISHING A DSOM SECURE CLIENT-SERVER ASSOCIATION

Upon attempting to establish a connection with a server in a separate address space or on a remote host machine, the secure DSOM client ORB transparently communicates the clients *credentials* and *privileges* to the server's ORB. Following the authentication of the client to the server ORB, the server is said to have acquired the client's security context. This defines a client-server security association under which requests for services will be carried in conjunction with an access control policy that the server maintains for its resources, or simply objects. A credential is an entity that can be used to establish the authentication; i.e. proving, the client's identity; e.g., a secret password, a DCE or a Kerberos ticket (current implementation provides for these three options.) While a

Figure 2 Basic DSOM client/server interactions.

privilege represents a special authority in the ‘eyes’ of the server program or it might simply be a list of groups of which the client is an authorized member. This list in turn grants the authority of the groups that it enlists to the client. The credentials serve the purpose of proving the identity of the client to the server, while the privileges are used by the server in the process of enforcing some specific access control policy to its objects. Both the credentials and privileges are anchored through the DSOM Object Security Service (OSS) principal object; i.e. they are part of the principal object’s instance data. We refer to the process of communicating the client’s credentials and privileges to the server by *establishing a shared security context*, or in short the security context. Note that establishing a security context is transparently performed through the cooperation of the client and the server ORBs, and results in the client’s OSS principal object built on the server side.

3.1 Building the principal object

Following an established security context between a client and a server we seek to devise a method by which the client privileges can be made available to the server application. Towards achieving this goal, we take advantage of the OMG definition of the **get_principal** method used by the server to return the ID of the client that issued the request. DSOM implements this method by returning a principal object from class **Principal**. In turn, the principal object can be queued through **get_userName** to retrieve the client’s ID. As such, the principal object that satisfies the CORBA requirement for identifying the client is a ‘light weight’ object with respect to its instance data. The intent of the **get_principal**

method is to allow the server application to retrieve the identity of its caller at any point during its execution. In order to make the client's privileges available to the server application we take advantage of the presence of this principal object and use it to maintain the client's privileges so that we enable and facilitate the process of authorization checking to the server's protected objects. To avoid redundancy we sub-class the CORBA principal object from the OSS principal object representing the client, created as a result of establishing a shared security context. Figure 3 shows the relationship between the OSS and the CORBA principal objects. An invocation of the method `get_principal` will thus, return an object that anchors the client privileges.

Figure 3 *The CORBA principal object inheriting from the OSS principal.*

4 DSOM'S APPROACH TO OBJECT ACCESS CONTROL

Access control paradigm in object oriented systems lends itself well to controlling the method interface to a protected object as in Bryce (1993). This seems to be a natural approach in light of the fact that an object and its instance data can be transformed only through its interface. As such, object access control may fit with the evolving Role Based Access Control, as it is for instance presented by Ferraiolo and Kuhn (1992) and Sandhu and Feinstein (1994). The essence of the RBAC model is that access to resources is based on the transformation that one would effect on the resource as a result of the access. Thus, the transformation is confined to the subject's role in his/her organization. In this respect access control in object systems presents a departure from its counterpart in the traditional sense where controlling access to a resource, generally, implies whether or not data, in its passive form, can be disclosed, altered, or destroyed. Controlling access to an object through its interface level implies that the space of access rights or the set of allowable permissions becomes proportionate to the number of methods a class library introduces. Therefore, we seek to determine a fixed set of usable and manageable access rights that can be used by application developers in securing class libraries. The importance of using such a fixed set of rights is desirable in order to enable application portability and reduce the complexity of administering object access control.

The solution that we devise to address this problem of scale breaks an access right into two components a right type and a value. The goal is to define a small number of family right types; each containing a manageable set of permissions. A right type is

intended to be generic and to encompass a wide scope of semantic; while a right value is more specific to within the semantic of its type. A right type can be thought of as being equivalent to a POSIX semantic of a DCE ACL manager. Figure 4 depicts two family rights OPERATION_RIGHTS and ROLE_RIGHTS.

Rights Type	Right	Intended Interpretation
OPERATION_RIGHTS	R	Read
	W	Write
	X	eXecute
	C	Create
	D	Delete
	A	Append
ROLE_RIGHTS	G	Guest
	U	User
	O	Operator
	M	adMinistrator
	T	audiTor
	S	Super

Figure 4 DSOM OSS family of right types.

In traditional procedural systems, the process of enforcing access control is triggered by the component of the operating system that has control over the resource. Such operating system components and their equivalents in application software are known as *resource managers*. A common example of such resource managers is a file system. Upon a request to access a protected resource, its resource manager intercepts the user's request and computes an access decision based on the requester's privileges, and the security attributes associated with the resource, generally in the form of an Access Control List, or ACL. A resource's ACL enumerates the principal's having access to the resource along with their mode of access; e.g., EXECUTE. In this paradigm access control to resources owned and managed by the application is explicitly enforced by the developer.

4.1 Approaches to enforcing access control

Mapping the paradigm of invoking access control that we discussed in the previous section to a distributed object system such as DSOM implies that prior to dispatching a client request to execute a method on a protected DSOM object, an authorization mechanism needs to be triggered to verify that the client is allowed to invoke the underlying method. This access decision is computed based on the client's privileges acquired during the process of establishing a shared security context and a security policy, generally, managed by the server. Programming this interception mechanism by the application developer is

not a feasible solution as the authorization code has to be inserted into every method procedure that is to be controlled. This approach poses a great deal of inconvenience and results in the authorization code being duplicated. It is characterized by a very poor scalability as the number of methods defined by a protected class library grows.

An alternative solution might be the use of DSOM ORB to intercept the process of dispatching a method before the method itself takes control. This solution can be best implemented by the DSOM object adapter, known as SOMOA. SOMOA, as we have mentioned earlier, is a run-time DSOM object that comes up on the server side of a distributed application and defines the main interface between the server application and the DSOM run-time. In particular, SOMOA handles all inbound requests for services and their outbound results. It receives client requests for method invocations on target objects created within the 'realm' of the server and dispatches the request on SOMDServer object of the server application using the following method

```
// method called by SOMOA to dispatch an incoming request
void somdDispatchMethod( in SOMObject somobj,
                        in somId methodId,
                        in var_list ap,
                        out somToken retValue );
```

where the target object is designated by `somobj`, the requested method is `methodId`, `ap` holds the argument list, and `retValue` contains data that the method returns. Subsequently, SOMDServer invokes the method on the target object. The SOMOA object, therefore, acts in the role of the server agent that the DSOM ORB contacts first on an incoming request. To provide for access control at the SOMOA level, one has to modify the sequence of the SOMOA implementation that invokes method `somdDispatch` above. First we have to determine that the target object is designated to be protected and, then invoke the authorization mechanism that returns a boolean decision. Based on this decision, the method is dispatched or rejected as shown in the following:

```
//
if( protected(somobj) ) {
    principal = get_principal(...);
    if(access_allowed( principal, somobj, methodId )
        somdDispatchMethod( somobj,
                            methodId,
                            ap,
                            retValue );
}
```

The principal object, as we discussed earlier, represents the client and anchors his/her privileges used during authorization.

This solution builds object access control into the DSOM kernel and relieves application developers from having to manage and explicitly invoke access control. However, it requires having a way for the developer to indicate to the SOMOA whether an object requires access control. Additionally, SOMOA will have to find out whether an object is protected on every method invocation even when the server does not make use of any type of access

Figure 5 *Modifying DSOM ORB to perform object access control.*

control. Finally, this solution requires rewriting the SOMOA code, and is applicable only on the server side as SOMOA and SOMDServer are not available on the client side. In the following we show how we make use of the currently available DSOM technology, without requiring changes to it, to seek a solution that is equivalent to this one but addresses the aforementioned concerns.

5 SOMMBEFOREAFTER METACLASS

A *before method* is a behavior that precedes the action and side-effect of some program construct. An *after method* is a behavior exhibited following the effect of some program construct as it is described in Paepcke (1993). It is generally desired that the nature of the behavior be left to the application developer so that it can be tailored to the needs at hand. SOMMBeforeAfter is a SOM run-time predefined metaclass that inherits directly from SOMClass and introduces two methods BeforeMethod and AfterMethod. A SOM class whose metaclass is the SOMMBeforeAfter arranges for the execution of the BeforeMethod and AfterMethod, respectively, before and after each execution of the class' instance method. By default the behavior exhibited by these two methods does not perform any functionality. Application developers are expected to sub-class this metaclass and override the Before/After methods to define their own particular behavior.

The essence of the SOMMBeforeAfter implementation in SOM is the overriding of the method `somDispatch` introduced by `SOMObject`. As a result of designating the `SOMMBeforeAfter` metaclass for a SOM class, stubs are placed in the class' method table, as described by Forman et al. (1994), to call the `somDispatch` override that looks like this

```

somDispatch( self, primaryMethod, ... ) {
    retval = BeforeMethod( class( self ),
                          self,
                          primaryMethod, ... );

    if( retval ) {
        primaryMethod( self, ... );
        AfterMethod( class( self ),
                    self,
                    primaryMethod, ... );
    }
}

```

As shown above, the SOM before/after behavior implementation through `SOMMBeforeAfter` metaclass is, somewhat, particular in that a boolean value is returned as a result of the before behavior. Only when this value is `TRUE` can the primary method (or the target method) be executed along with the after behavior. Furthermore, the `SOMMBeforeAfter` metaclasses have the property of composition whereby the before after behavior of multiple `BeforeAfter` metaclasses that a given metaclass inherits from is executed in a preorder traversal of the metaclass inheritance hierarchy. Whereas the after behavior is achieved in reversal order so that the desired before and after behaviors match one another as it is presented by Forman et al. (1994). In addition, currently the behavior is based on the class granularity level as opposed to a finer level, for instance on method granularity such as in Paepcke (1993).

6 SOMMBEFOREAFTER: ENABLING DSOM OBJECT ACCESS CONTROL

`SOMMBeforeAfter` metaclass defines a point from which to intercept a method invocation right before its execution takes place. Moreover, this interception can be selectively made for a given class and its subclasses, while it can be avoided for another class. This simply falls into the application developer's responsibility by either assigning the target class to the `SOMMBeforeAfter` metaclass or to some other metaclass. The interception point consists of the before behavior that, like in any other SOM class, can be overridden through subclassing in order to provide for the desired behavior. At this juncture it becomes clear that our desired before behavior is to perform method level access control. Thus, instead of modifying the DSOM ORB to perform such task as we discussed earlier, we simply make use of the existing technology within DSOM to accomplish our goal.

We define the *secure metaclass*, denoted `SOM_M_Secure` to be a subclass of `SOMMBeforeAfter` as follows

```
// secure metaclass definition
```

```

interface SOM_M_Secure: SOMMBeforeAfter
{
    // implementation
    dllname = 'security.dll'

    // method modifiers
    sommBeforeMethod: override;
}

```

where `sommBeforeMethod` is overridden to perform authorization checking as defined by the following

```

// override of sommBeforeMethod
boolean SOM_M_Secure_sommBeforeMethod( SOMObject* object,
                                         somId      methodId,
                                         ap          va_list )
{
    principal = get_principal(...);
    return access_allowed( principal, object, methodName ...);
}

```

`access_allowed` method performs authorization checking by comparing the caller's authorization credentials anchored by the principal object against the ACL for the corresponding object, with the desired permission being the target method. The access control decision is returned as a value for the before behavior and, therefore, the target method will be dispatched or not according to this decision.

An application developer that wishes to protect a particular SOM class simply needs to assign `SOM_M_Secure` class as a metaclass for the target class. Due to a SOM limitation, however, that only one metaclass can be specified in an IDL definition of a class and as such the use of `SOM_M_Secure` prohibits defining another metaclass, we create `SOM_Secure` class as in the following

```

// SOM secure class
interface SOM_Secure: SOMObject
{
    ....
    // implementation
    dllname = 'security.dll';
    metaclass = SOM_M_Secure;
    ....
}

```

`SOM_Secure` class is defined to have `SOM_M_Secure` as its metaclass and as a result the class hierarchy descending from it will inherit the before behavior that is defined by `SOM_M_Secure` metaclass. Therefore, a particular class can be designated protected by its developer by simply inheriting directly from `SOM_Secure` or from one of its descendants as shown in the following

```
// protecting myClass
interface myClass: SOM_Secure, myotherClass
{
    ...
}
```

Because DSOM multiple inheritance resolves an inherited method according to a pre-order traversal of the parent classes, it is required in this solution to have `SOM_Secure` as the first parent class; i.e., the leftmost class. Subsequently, as we descend in the hierarchy derived from `SOM_Secure`, we are required to inherit first from the protected class(es) as shown in the following

```
// protecting myNewClass
interface myNewClass: myClass, anotherClass
{
    ...
}
```

where by first inheriting from `myClass` the newly defined class maintains protection.

This ordering also remedies the problem of `SOMMBeforeAfter` composition in which an application developer makes use of multiple `SOMMBeforeAfter` metaclasses (with different before/after behaviors.) In this case it is desired to avoid the observance of any before/after behavior when authorization checking fails. This also provides a control point for turning off access control, by not following the ordering of inheritance described, for a particular sub-class that descends from one or more protected classes.

7 CONCLUSION

We have defined the elements of object access control for DSOM that can be applicable to other object systems. These elements include the granularity of controlling access based on the method level; choosing an access right based on two components: a type and a value so that to accommodate for method level semantic, and a method by which to maintain a client's privileges through the use of the principal object. More importantly, we have devised a novel method for enabling the process of DSOM access control transparently and with minimal developer's intervention. This method is characterized by two factors:

- it makes use of existing DSOM technology by simply using `SOMMBeforeAfter` meta-class and avoids involving the application developer or modifying the DSOM kernel ORB to achieve this purpose.
- it is easy to use and involves a very small effort on the part of the application developer, namely requiring that the protected class inherit from `SOM_Secure` class during the IDL definition phase.

This solution can be improved at two levels. The first is by providing for a before/after behavior that is based on object instance level, and in the second one that is based on the method level. The former allows for objects of the same class to be selectively

protected, while the latter enables access control to be invoked for a particular method and automatically bypassed for another method from the same class. Finally, the proposed method can be adopted by other object systems having support for class inheritance and the before/after behavior or its equivalent.

REFERENCES

- Bryce, C. (1993) Protection in object-oriented software, *Proceedings of 2nd International Conference on Achieving Quality in Software*, Venice, Italy, 97–109.
- Ferraiolo, D. and Kuhn, R. (1992) Role-based access controls, *Proceedings of the 15th National Computer Security Conference*, Baltimore, Maryland, 554–563.
- Forman, I., Danforth, S. and Madduri, H. (1994) Composition of Before/After Metaclasses in SOM, *Proceedings of OOPSLA '94 Conference Proceedings*, 429–437.
- IBM SOMObject Developer Toolkit Users Guide (1994).
- IBM SOMObject Developer Toolkit Reference Guide (1994).
- OMG Common Object Request Broker Architecture and Specification (1991), *Document 91.12.1*, Framingham, Massachusetts.
- OMG Object Services Request For Proposals (1994), *RFP3*, Framingham, Massachusetts.
- OMG White Paper on Security (1994), Object Management Group, *Document 94.4.16*, Framingham, Massachusetts.
- OSF DCE Application Development Reference (1993), *Open Software Foundation*, Prentice Hall, Englewood Cliffs, New Jersey.
- Paepcke, A., ed., (1993) Object-Oriented Programming: The CLOS Perspective, *MIT Press*, Cambridge, Massachusetts.
- Sandhu, R. S. and Feinstein, H. (1994) A three Tier Architecture for Role-Based Access Control, *Proceedings of 17th National Computer Security Conference*, Baltimore, Maryland, 34–45.