A Design for Inter-Operable Secure Object Stores (ISOS)

Carl Lagoze
Digital Library Research Group
Cornell University

Robert McGrath
Computing and Communications Group
NCSA

Ed Overly
CNRI

Nancy Yeager
Computing and Communications Group
NCSA

November 7, 1995

Abstract

We describe a distributed object-based design for repositories in a digital library infrastructure. This design for Inter-operable Secure Object Stores, ISOS, defines the interfaces to secure repositories that inter-operate with each other, clients, and other services in the infrastructure. We define the interfaces to ISOS as class definitions in a distributed object system. We also define an extension to CORBA security that is used by repositories to secure access to themselves and their contained objects.

1 Introduction

This paper describes a fundamental component of digital libraries, repositories for information in digital form. The success of any such design will depend greatly on its fit with the existing social, economic, and legal context. We define this context as follows.

- The legal framework for protecting intellectual property is strongly established in our existing information infrastructure. The digital analogue will succeed only if it fits within this existing framework.
- The existing infrastructure consists of a large number of diverse and competitive parties. The digital infrastructure will fail if it tries to impose centralized control or standardization on these parties, other than that necessary for inter-operation with other services and protection of intellectual property.
- The use and protection of derivative works is a fundamental notion. The shift to the digital domain opens up many new opportunities for deriving new content from existing content. The infrastructure must exploit this capability, yet protect all parties from abuse of the technology.

The approach presented in this paper, a design for Inter-operating Secure Object Stores (ISOS), is based on a framework for distributed digital information systems first articulated by Robert Kahn and Robert Wilensky [19], as part of the CS-TR project[1], and further investigated in [22]. Kahn/Wilensky broadly defines the components of an open system for storage, access, dissemination, and management of information in the form of *digital objects*. A digital object is a content-independent package that includes the content of a work, a unique identifier for the digital object (its *handle*), and other data about the object, such as the terms and conditions for use of the object. *Repositories* logically store digital objects and are responsible for protecting resident objects according to their respective terms and conditions. An access request on a digital object produces a *dissemination*, which contains the results of the access request (determined by the parameters in the request) and additional data specifying the origin of the dissemination and the specific terms and conditions governing its use. It is important to note that Kahn/Wilensky does not restrict a dissemination to have the same data as the source digital object. Nor does it specify that the dissemination is necessarily some subset of the digital object's data (*e.g.*, a dissemination that is the result of an access request for a single page of a book stored as a digital object). For example, a digital object may be an executable program and disseminations may be produced by running the program using the parameters in the access request as input. Finally, Kahn/Wilensky present the basic components of a protocol, the *Repository Access Protocol* (RAP), which includes services for deposit of digital objects and access to digital objects.

Kahn/Wilensky makes no assumptions about implementation details. This paper describes one possible design approach to the Kahn/Wilensky framework; based on the distributed object model. We have chosen this model because it allows us to define ISOS interfaces without linking them to specific transport or session layer protocols. Within this object-oriented framework, ISOS makes two contributions. First, it provides class definitions (instance variables and methods) for the objects in Kahn/Wilensky: `digital_object`, `dissemination`, `repository`, `data`, and `terms_and_conditions`. The methods are semantically equivalent to Kahn/Wilensky RAP. These class definitions are the basis for interoperability among individual ISOS repositories and between these repositories and other digital library services. Second, ISOS defines a uniform and extensible method for securing access to repositories, to digital objects, and to operations on digital objects. Because CORBA has a relatively well-defined security architecture, we use it as the basis for defining security in ISOS. We define a new object class, `terms_and_conditions`, that is an encapsulation of the stated terms and conditions that apply to access to repositories, digital objects, and disseminations. This `terms_and_conditions` class interacts with the CORBA security architecture allowing a repository to provide protection for the objects that it contains.

The remainder of this paper is structured as follows. We summarize the broader societal framework into which ISOS fits. We then describe the primitive classes that are defined by ISOS and their context within a distributed digital information system. We follow this with a description of the ISOS security architecture and its enhancements to the CORBA security model. Finally, we list planned future work based on this design.

2 The Social, Economic, and Legal Frameworks

The social, economic, and legal frameworks for dissemination of information on networks are still emerging. Current information services, such as those currently available on the Internet, provide essentially no control over the use of on-line information and are therefore most suitable for public access to open information. This is an important class of information, but, more systematic control is a prerequisite for the dissemination of items that are not in the public domain and for on-line commerce.

Following Kahn/Wilensky, ISOS makes no assumptions about what particular economic and legal frameworks will eventually emerge. It provides a structure that can be used to manage a wide range of terms and conditions associated with digital objects. To achieve this end, terms and conditions are associated with each digital object, dissemination, and repository in the design. The repository stores these terms and conditions and enforces them as a precondition to accessing stored digital objects. Repositories will necessarily need to be consistent with the current and emerging legal framework.

The creator of a digital object and the party that stores the object in a repository are able to assign terms and conditions to the object. These terms and conditions associated with digital objects may take many forms. They might include, for example, access restrictions, claims of copyright, or payment terms. For example, access to an object might be free to members of a certain community (e.g., university students) but might require payment from others. Some objects might require a certain security clearance to be disseminated. There are many permutations of different types of terms and conditions and the possibility exists for very complex formulations. Frequently, however, most parties are interested in terms and conditions that are simple to formulate and understand, including the important case of public access.

Repositories must enforce access controls to the stored objects according to their respective terms and conditions. They may keep a log that records fulfillment of terms and conditions and may be called upon to produce records for certain transactions. The design of the repository provides a secure interface, which ensures that these terms and conditions are observed in all interactions with the repository.

Clients must satisfy repositories that they meet the stated terms and conditions for the dissemination of a particular digital object. This requires a means to request, present, and negotiate terms and conditions via information networks. This includes specification of a method for codifying terms and conditions that is portable and extensible, and which allows them to be inter-operable among different platforms and between different clients and repositories. It also includes the procedures by which a client (which might be another repository) can inquire about terms and conditions from a repository and obtain disseminations of the digital objects.

3 ISOS Base Classes

ISOS, by itself, is not a digital library. It is the component of the infrastructure that provides secure storage for the items in a digital library collection. A digital library infrastructure will include many other services. ISOS is directly dependent on some of these such as naming services (in this paper and in our planned implementation we use the CNRI handle service [12]), authentication services [26] [21], and payment services [3] [25] [9]. Some higher level services that may use the services provided by ISOS are indexing and search services [16] [4], browsing services [13], annotation services [27] [10], and link services [20].

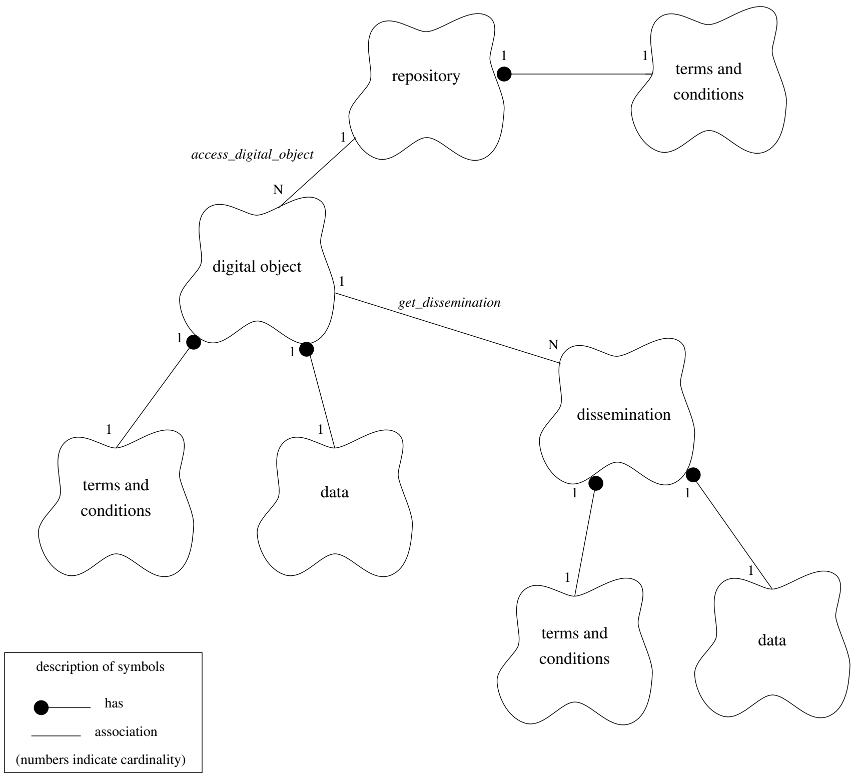

Figure 1: Relationship of ISOS base classes.

In this section we describe the components of the infrastructure that ISOS provides. We do this by describing[1] the five base classes in the ISOS class hierarchy. This class structure could be implemented in any distributed object architecture; *e.g.* CORBA or OLE [7].

An overview of the class hierarchy is as follows (figure 1 - notation from Grady Booch [6]). The class `repository` contains the method that allows clients to reference instances of class `digital_object` and subsequently execute methods on those instances. The class `digital_object` is a content-independent package for digital material, a unique identifier for the material, and additional items that enforce and record access to the material. Instances of `digital_object` are (repository) server resident and are only accessible to clients as remote references (i.e., the digital object itself is not instantiated on client). The class `dissemination` is a content-independent package for digital material and related data (including conditions of

[1]The included class descriptions are not complete. We describe the subset of instance variables and methods that are essential for understanding the ISOS design.

use) that is instantiated at the client when the client invokes a digital object's *get_dissemination* method. The class `data`, which is a package for content, is contained within each instance of the classes `digital_object` and `dissemination`. . Finally, the class `terms_and_conditions`, which encapsulates access rules that secure access, is contained within each instance of the classes `repository`, `digital_object`, and `dissemination`. The description of the class `terms_and_conditions` is left to section 4, which describes the ISOS security architecture in detail. These are the publically visible classes in ISOS, an implementation will define other classes that are used internally.

We use the term *invocation* throughout the class descriptions. Distributed object-oriented applications, like their non-distributed counterparts, execute by invoking methods on objects, which are instances of defined classes. The objects may reside on servers, rather than being in the same memory space as the caller of the method. Clients refer to an object on a server using an *object reference*, the result of an *invocation*. The distributed object system routes the method request to the server resident code for the object instance and returns results to the client.

`DIGITAL_OBJECT` A `digital_object` is a content-independent container for instances of class `data`. In this sense, any class hierarchy that descends from `digital_object` will be orthogonal to that for class `data`. Expressed informally, there will not be different types of digital objects for video content, text, agents, etc. In addition to `data`, the class `digital_object` contains a `handle`[2], its unique identifier, and `access_rules`, which are an instance of the class `terms_and_conditions`. These `access_rules` control invocation of the `digital_object` and access to its methods.

The important method for class `digital_object` is *get_dissemination* which returns to the client an instance of the class `dissemination`.

`REPOSITORY` The class `repository` has two notable instance variables. First, the `handle` of the repository is the unique, persistent identifier for that repository. This identifier is registered with the name service. Second, `access_rules`, which are an instance of the class `terms_and_conditions`, are used to control invocation of an instance of `repository` by a client. We assume that certain repositories might need this level of security, separate from that provided for each digital object.

Clients use the repository method *access_digital_object* to obtain invocations of instances of class `digital_object` that are "contained" in the respective repository. The client supplies the handle of the digital object as the method parameter. The repository method *deposit_digital_object* is called by a client to instantiate an instance of `digital_object` in the respective repository.

`DISSEMINATION` Instances of the class `dissemination` are created on a client as the result of a call on the *get_dissemination* method for a `digital_object`. Each `dissemination` instance contains an instance of the class `data`. There is no requirement that the `data` in a dissemination is identical to or contained within the `data` in the source digital object. In fact, it may be a completely different sub-class of the class `data` than that in the source digital object. Some examples illustrate the scope of this relationship. There may be a digital object which contains the fixed PostScript encoding of a computer science technical report. A dissemination of this digital object may contain that same PostScript encoding, or a portion thereof. There may also be a digital object that contains an program that is recording the current video image of a Senate session. A dissemination of this digital object might be the MPEG clip of the ten minutes of that session on September 1, 1995. Finally, there may be a digital object that contains an program, a dissemination of which might contain another program that interacts with the user, external services, and the source digital object in the repository.

A `dissemination` also contains `access_rules`, an instance of the class `terms_and_conditions`. These control access to the dissemination, and are derived from the rules in the source `digital_object`. Note that securing access to the dissemination (*e.g.,* enforcing payment per access) is complex since the dissemination is not contained in repository. We are exploring methods for doing this, such as encrypting the data in the dissemination and including in the access rules the location of a network-accessible "applet" that would decrypt the data only after enforcement of the access rules (e.g. interaction with a payment service).

[2] Note that *handles*, which are a general naming scheme outside the distributed object domain, are distinct from CORBA-type *object references*. This requires mapping from the handle to the object reference throughout this design.

`DATA` Each instance of the classes `digital_object` and `dissemination` contain an instance of the class `data`. The primitive class `data` simply packages a bit stream, with a basic *get* method to access that bit stream. Real use of the design, however, will depend on extensive sub-classing of this class to allow for new content types. This makes client design difficult since new content types may be introduced that are not recognized by clients. Dynamic object facilities such as the CORBA Interface Repository[29] and Dynamic Invocation Interface[8] will facilitate this sub-classing. Some sub-class examples of `data` are `PostScript_data`, with a *get* method allowing specification of a page range, and `MPEG_data` with a *get* method allowing specfication of a time-slice. We define, as in Kahn/Wilensky, two sub-classes of data: `contained_digital_object`, which contains an instance of `digital_object` (thereby allowing a digital object to package another digital object), and `contained_handle`, which contains an instance of a `handle` (thereby allowing a digital object to reference another digital object). Finally, we do not restrict each instance of `data` to be a singleton. For example, a `digital_object` might contain `data` that is a `set` of `contained_handle`, effectively allowing a digital object to reference a group of digital objects (e.g., a digital object for a computer science technical report that contains the handles of the digital objects containing that report in different formats.

A Demonstration of ISOS interaction

A sample interaction of these ISOS objects with other objects in a digital library infrastructure is demonstrated by the following brief example. The security issues of the example are discussed more completely in 4.

Lucy is a computer science graduate student who wants to find research papers on "functional programming". She knows that CS Research Associates (CSRA) has done an excellent job indexing computer science research. CSRA has done this by getting permission (perhaps through licensing) from many computer science research repositories to browse their collections and index them. Using her browser, Lucy searches the CSRA index and sees a set of search "hits" on her screen. She chooses one of the hits, a paper in *ACM TOPLAS*, and the following set of steps occur:

1. **Resolve the handle.** Using the name service, the browser resolves the handle to one or more repository handles. The browser may then select one repository based on some user profile (cost based, location based) or other decision process.

2. **Resolve the repository handle.** Again using the name service, the browser resolves the repository handle to the object identifier of the repository.

3. **Create a client invocation of the repository object.** This invocation is, in effect, the initiation of a session with the repository, and is the binding that the client can use for further method calls on the repository object. This step allows the enforcement of any `access_rules` associated with the repository. This might involve negotiations between the repository object and client and interaction with payment and authentication services. The invocation has an associated security context.

4. **Access the digital object.** The client invokes the repository's *access_digital_object* method with the handle of the digital object as an argument. This returns to the client an invocation of the digital object that it can use for further method calls on the object. The `access_rules` associated with the object may require interaction with outside services. This establishes an object-specific security context.

5. **Check the type of the data of the object.** The client invokes the digital object's *get_data* method to get a binding to the digital object data. This is again subject to `access_rules` and associated processing. Using this binding the client can check the type of the data and determine whether it should present the user with access choices (e.g. which pages do you want?). If the type is unrecognized by the client, this may require client interaction with a dynamic type service.

6. **Get a dissemination.** The browser invokes the *get_dissemination* method for the digital object. The parameters to this may be a data specific method call (e.g. *get_page(2)*). This returns an instance (**not** a binding) of the dissemination on Lucy's workstation. At this point the interaction with the repository is complete, since the dissemination is not contained in the repository.

7. **View the document.** Lucy can now use a view program, which may be the same as the browser, to view the contents of the dissemination. If the dissemination is protected by `access_rules` and encryption, this may require further interaction with other services (e.g. authentication service, type service).

4 The ISOS Security Architecture

The enforcement of a digital object's terms and conditions requires a range of security services: authentication, access control and secure associations. ISOS relies on the CORBA security framework[28] for these standard security services. In Section 4.1 we summarize the existing security framework defined by CORBA. Then, in section 4.2 we describe the motivation for `terms_and_condition` class, which we use as the uniform device for enforcing secure access to digital objects, repositories, and disseminations and their respective methods. We then proceed to explain proposed extensions to the CORBA security framework that permit interaction between its facilities and instances of the `terms_and_conditions` class. Some of the specifications in the latter two sections are the basis for long-term research, described in section 5.

4.1 The CORBA Security Architecture

The CORBA Security model is an extension to the standard CORBA framework [14, 28]. The CORBA security model defines a *secure object invocation*, which includes:

- establishing a secure association between the client and the object. Authentication is typically part of this process.
- enforcing the access control for each operation
- logging and auditing
- secure transmission

The key elements of the CORBA protection model are illustrated in Figure 2.

Objects in the CORBA object model and users who act on those objects each have attributes which define their secure interaction. The client has *privilege attributes (pa)* and the protected object on the server has *control attributes (ca)*. The standard CORBA Object Request Broker (ORB) implements security services through *interceptor* objects. The *interceptor* objects are called by the ORB on each secure invocation. For example, interceptors are responsible for setting up a secure association between the client and server and for mediating authentication and access control services. Interceptors implement these basic security services by creating security objects: *the credential object* (CO), the *security context object* (SCO), and the *access decision object* (ADO).

Below we outline how security objects and attributes are supported in the CORBA object model.

Client-side attributes and security objects. The client's *privilege attributes (pa)*, or *credentials,* define the principal's access rights within the CORBA protection model. By definition the principal (client) has an identity or role within the system, which is represented by his credential within the system. The principal authenticates to establish his role, and his credentials define his rights to objects within the system. A Kerberos [21] token is an example of a specific type of credential. A user may have several roles and identities, which are represented by a set of credentials. A CORBA *credential object (CO)* encapsulates (stores) the user's credentials.

Server-side attributes and security objects. The server side stores and protects the target object. The server side's attributes and security objects are as follows.

The protected object's *control attributes* (ca) define the level of protection required to access the target object. Control attributes are set by the creator of the object through CORBA interfaces. For example, a document's creator may specify that a Kerberos token belonging to the group "team1" must be presented to read the protected resource. "kerberos-token-team1 read" are that object's control attributes.

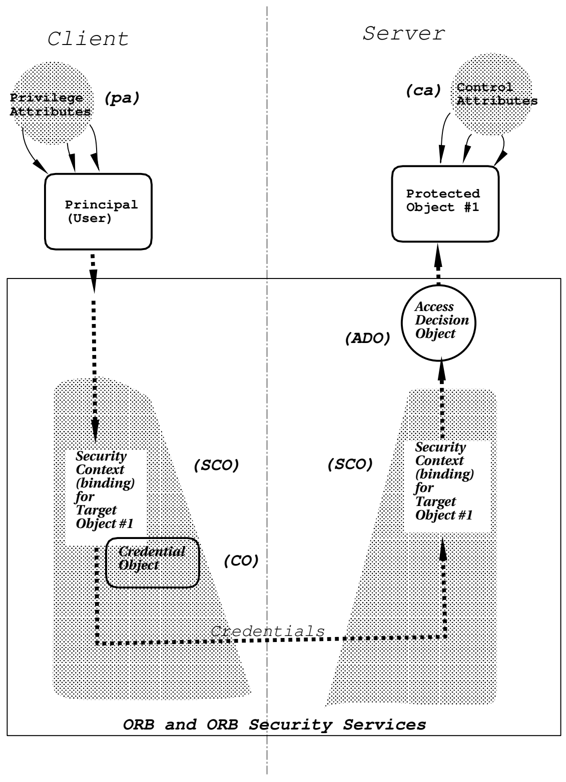

Figure 2: A sketch of the CORBA security architecture

The *access decision object* (ADO) is a security monitor that protects the object and enforces its defined control attributes. At the time of a method invocation, the ADO is responsible for granting or denying access to an object. The ADO is a generic representation of an access control system. It encapsulates the management and enforcement of access control policies. In order for the principal to gain access to an object, the principal must present credentials to the ADO to satisfy the control attributes of the object. The ADO's decision is based on the security context of the invocation: the privilege attributes of the principal making the request and the operation to be performed are compared to the control attributes specified for the object. An application may implement its own ADO, which may be substituted for the default ADO. The application ADO may enforce any access control policies required by the application.

Shared security context. The client and the object on the server communicate securely by setting up a secure association. When a client attempts to access a protected object, the ORB and interceptors establish a security context: a *security context object* (SCO) is created. This process typically requires authentication and may produce additional credentials. The shared security context is represented by a pair of security context objects, one at the client and one at the server. The security context is the vehicle by which the principal's credentials, stored in the credential object, are transferred and presented to the ADO. The ADO uses these credentials to make an access control decision with respect to the object.

4.2 CORBA Security and ISOS

Our design is to use the CORBA security model to provide security services for repositories and digital objects based upon the terms and conditions defined for them. Support for terms and conditions will require facilities beyond current conventional access control mechanisms: mechanisms to express the terms and conditions, mechanisms to convert them into the appropriate control attributes, and mechanisms to "explain" them to clients. Also, when a `dissemination` of a `digital_object` is created, there must be a mechanism by which the terms and conditions of the `dissemination` are created. The `terms_and_conditions` object performs these functions.

Every `digital object` in the repository has a `terms_and_conditions` object. The `terms_and_conditions` object represents the stated terms and conditions. The rules in the `terms_and_conditions` object are translated into the appropriate control attributes. This translation could occur when the `terms_and_conditions` object is created or modified, or at the time the object is accessed. This translation will often be very simple. In the simplest case the terms and conditions are essentially the object's control attributes, or Boolean combinations of control attributes expressed as conventional credential schemes. A terms and conditions for an object, expressed in natural language may be:

> Lucy owns "object1" and only she can read, modify, and delete it. Lucy will prove her identity with a Kerberos token.

The `terms_and_conditions` object might translate this statement into the following control attributes:

> "Kerberos-token-lucy read, modify, delete".

A fundamental requirement for the ISOS security model is that ISOS must support multiple types of credentials and access control mechanisms. This is necessary because the stated terms and conditions may involve credentials of several kinds for a given access. For example, a user may need to have a Kerberos token for "team1" and a RSA certificate for a member of the "gold card club" to access "object 1".

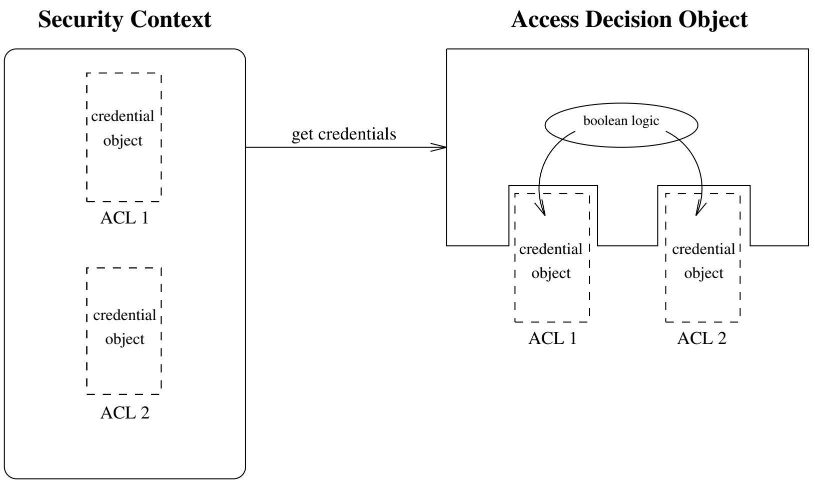

Figure 3: CORBA supports multiple types of credentials and access control schemes.

Figure 3 illustrates how the CORBA security framework is designed to support both multiple types of credentials and multiple types of access control mechanisms [28]. The CORBA model has a generic *credential object* , which may be sub-typed to implement a variety of credential schemes. These credentials are transferred to the *access decision* object (ADO) *via* the security context. The ADO is a generic interface, which may be sub-typed to implement a variety of access control schemes, such as access control list #1 (ACL 1) and access control list #2 (ACL 2). Each access control mechanism would have its own way for expressing rights associated with an object. ACL #1, for instance, could express and understand "Kerberos token for team1" while ACL #2 could express and understand "RSA certificate for a member of the gold card club".

The ADO would have to have a flexible language to express multiple types of credentials, multiple schemes for expressing rights for an object, and combinations of types of rights for an object. For example, the ADO would have to contain some form of *boolean logic* that could distinguish and enforce "Kerberos tokens *AND* RSA certificates".

The architecture and interfaces in Figure 3 are still under consideration for inclusion in the CORBA standard, and the exact mechanisms for supporting multiple access control mechanisms is still under discussion. To our knowledge, there is not yet an implementation of an ADO that would support multiple *types* of access control mechanisms. The CORBA security model specifically allows applications to enforce their own access control policies by implementing their own ADO. So, until a CORBA ADO that supports the necessary access control mechanisms is available, the ISOS repository will replace the ORB's ADO with it's own (application) ADO.

Just as the ISOS server may have to accept multiple types of credentials, the ISOS client may have to supply multiple types of credentials. The security context object (SCO) is the vehicle by which credentials are presented to the ADO. The SCO may have to link to external authentication services, which generate the credential necessary to satisfy the conditions for instantiating a protected object.

For this case, the CORBA security architecture recommends that Generic Security Services (GSS) [23] be used as a generic interface through which these authentication services are accessed. An authentication module, or library, which conforms to the GSS-API is installed on the client. When the security context object requires a credential the client invokes the correct routine in the authentication module *via* the GSS-API. The authentication service supplies the correct credential or negotiates with external agents for credentials on behalf of the client. For example, a Kerberos library on the client would negotiate with the third party Kerberos authentication service to supply and hand back the Kerberos token to GSS-API and consequently to the client's SCO. Similarly, the GSS-API may invoke routines which talk to an agent of an external payment protocol to negotiate payment tokens on behalf of the client.

4.3 How the `terms_and_conditions` Class is Used to Enforce Access Control

Stated terms and conditions may go beyond conventional credential schemes, which generally support individual and group membership. For instance, there may be a requirement for human intervention, either by the user or some third party, such as a certificate authority or a payment service. There may also be complicated algorithms required, such as arranging payment and acquiring multiple signatures. This may involve a negotiation between the user or her client-side software, the repository, and/or other parties. There may also be negative credentials of various kinds, such as, "the user must not be a felon".

These relatively exotic terms and conditions require a more complex mechanism to express them, to convert them into appropriate control attributes, and to "explain" them to clients. The **`terms_and_conditions`** object is intended to meet these requirements for these more complex stated terms and conditions.

The **`terms_and_conditions`** object encapsulates the terms and conditions as suggested above, and other behavior necessary to use multiple types of credentials and multiple access control mechanisms. As an example, suppose the stated terms and conditions specify that the document is freely available, but the user must acknowledge and abide by the conditions in the copyright statement. This is to be done by displaying the copyright statement and requesting that the user agree to the terms by pressing "OK". (This is a level of protection similar to commonly used "shrinkwrap" licenses around software distributions.) The *get_dissemination* method (step 6 in Section 3) requires that this procedure be satisfied before the dissemination is created and returned to the user.

Assume a secure association has been created (steps 3 and 4 of section 3). Figure 4 shows how the object would be protected using the **`terms_and_conditions`** object. The numbers in the figure correspond to the steps in the description below.

1. The client initiates a *get_dissemination* request. (Step 6 of Section 3).

2. The request is passed through the client-side interface, across the ORB, to the server.

3. The Interceptor traps the request and calls *access_query()* method of the ADO. In this case, the decision is made by an application ADO, that is, the ADO of the repository.

4. The ADO gets the current credentials from the *security context* to determine the principal's **`privilege attributes`**, which are compared to the *control attributes* of the object. In this case, the ADO realizes that the *control attributes* are a special type which indicate that the ADO should invoke a method on **`terms_and_conditions`** object of the stored digital object to acquire the necessary credentials.

5. The **`terms_and_conditions`** object uses the facilities of the *security context* (SCO) to request the "credentials" it needs. That is, it requests a "the user has seen and agreed" credential, and passes the required copyright message to the SCO.

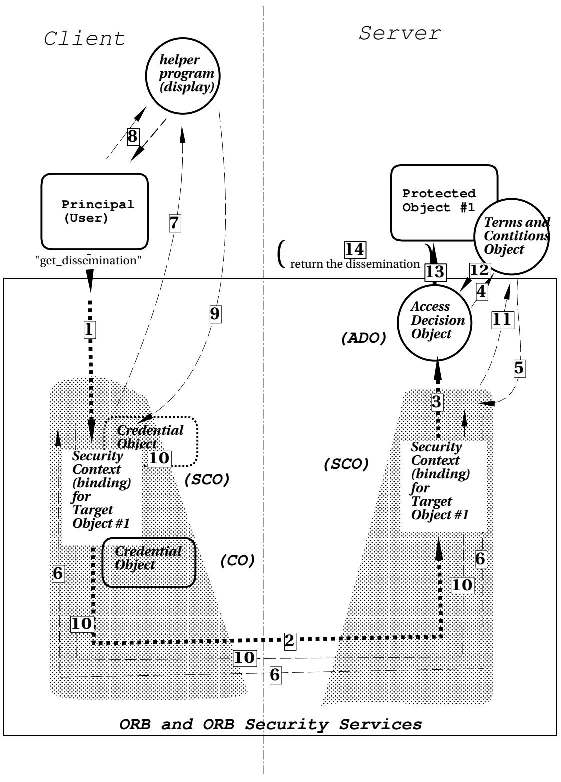

Figure 4: The **Terms_and_Conditions** Object implements complex sets of credentials

6. The SCO, using the GSS API, (securely) transmits the request for credentials, along with the accompanying copyright message, to the client.
7. The client-side GSS invokes a module or helper program (possibly downloaded in a language such as Java[24] or Python[2]) to display the copyright statement and collect the response.
8. The helper program displays the message and collects the user's assent.
9. The helper program returns the user's reply ("OK"), and synthesizes an appropriate "credential".
10. GSS returns the "credential" to the server-side.
11. The server-side SCO returns the "credential" to the `terms_and_conditions` object.
12. The `terms_and_conditions` object returns "allowed" (or "disallowed") to the ADO.
13. Access is granted (or denied) by the ADO, and the method request proceeds (or fails).
14. The dissemination is created and returned to the user. (Not shown in the figure.)

5 Future Work

The completion of this paper represents the end of the initial design phase of ISOS. Over the next few months we plan to prototype the design in an initial implementation that will permit simple terms and conditions. Following this initial prototype, we plan to explore a number of issues.

Of primary concern is how well the design inter-operates with other digital library services, both object-based and those based on other paradigms. Coordination with work being done by the six Digital Library Initiative sites will be especially important. Furthermore, we will examine how well ISOS accommodates complex and new data types. Finally, there are a number of issues related to the scalability of the design that need to be examined; for example replication of digital objects and propagation of changes to digital objects across replications.

In the longer term, there are a number of research issues related to security that we will explore. ISOS requires that the stated terms and conditions for every object be translated from natural language into a machine readable representation. The repository may be required to "explain" the terms and conditions to other parties, including client software, user agents, and other cooperating repositories. This requires a protocol for establishing a mutually intelligible "language", and other suitable languages. This is an active area of research. For instance, the Stanford DLI project [15], several ARPA projects (the KQML project [11], the III project [17], the NIII project [5]) and the CADR group at Illinois [18, 30] all address aspects of this problem.

It is also desirable that the terms and conditions be maintained in a way that is understandable by humans. It should always be possible for authorized parties to obtain a human readable version of the terms and conditions in effect. This version should be an accurate reflection of the actual rules that are enforced.

A third desirable goal is for terms and conditions to be relatively permanent and portable. While the actual enforcement mechanisms depend on the specific platform and architecture, the stated terms and conditions should be portable to new platforms, and should carry forward as time passes.

References

[1] http://www.cnri.reston.va.us/home/cstr.html.

[2] http://www.python.org.

[3] Digicash brochure. http://www.digicash.com/publish/digibro.html.

[4] Lycos home page.

[5] NIIIP: National industrial information infrastructure. http://www.niiip.org.

[6] Grady Booch. *Object-Oriented Analysis and Design.* The Benjamin/Cummings Publishing Company, Inc., 1994.

[7] Kraig Brockschmidt. *Inside OLE 2, Second Edition.* Microsoft Press, 1995.

[8] NEC Corporation. NEC dynamic invocation interface example. Technical Report 93-1-2, Object Management Group, 1993.

[9] Steve B. Cousins, Steven P Ketchpel, Andreas Paepcke, et al. Interpay: Managing multiple payment mechanisms in digital libraries. Technical report, Stanford University, March 1995. to appear in Digital Libraries '95.

[10] Jim Davis and Dan Huttenlocher. CoNote (annotation) homepage. http://dri.cornell.edu/pub/davis/annotation.html.

[11] Tim Finin, Rich Fritsson, and Don McKay. A language and protocol to support intelligent agent interoperability. In *Proceedings of the CE & CALS Washington '92 Conference*, June 1992.

[12] Corporation for National Research Initiatives. Handles and the handle system. http://www.cnri.reston.va.us/home/cstr/handle-intro.html.

[13] Luis Gravano, Hector Garcia-Molina, and Anthony Tomasic. The effectiveness of GLOSS for the text-database discovery problem. In *SIGMOD '94*. 1994.

[14] Object Management Group. The Common Object Request Broker: Architecture and specification. ftp://omg.org/pub/CORBA, December 1993.

[15] Stanford Digital Libraries Group. The Stanford Digital Libraries Project. *Communications of the ACM*, April 1995.

[16] Darren R. Hardy and Michael F. Schwartz. Harvest user's manual. Technical Report CU-CS-743-94, University of Colorado at Boulder, October 1994.

[17] Richard Hull and Richard King. Reference architecture for the intelligent integration of information. Program on Intelligent Integration of Information, Advanced Research Projects Agency, March 1995. http://www.isse.gmu.edu/I3_Arch/index.html.

[18] E. Jones, N. Ching, and M. Winslett. Credentials for privacy and interoperation. In *Proceedings of the New Security Paradigms '95 Workshop*. August 1995.

[19] Robert Kahn and Robert Wilensky. A framework for distributed digital object services. http://www.cnri.reston.va.us/home/cstr/arch/k-w.html, May 1995. also accessible as cnri.dlib/tn95-01.

[20] Frank Kappe. Hyper-G: A distributed hypermedia system. In *Proceedings of INET '93*. 1993.

[21] J. Kohl, B. Clifford Neuman, and J. Steiner. The Kerberos network authentication service. Technical report, MIT Project Athena, November 1989. Version 5, Draft 2.

[22] Carl Lagoze and David Ely. Implementation issues in an open architectural framework for digital object services. Technical Report TR95-1540, Cornell University, Department of Computer Science, 1995. also accessible as CORNELL.CS/CORNELLCS:TR95-1540.

[23] J. Linn. Generic security service application program interface. RFC 1508, 1993. http://ds.internic.net/rf/rfc1508.txt.

[24] Sun Microsystems Computer Company. The Java language environment. White Paper, May 1995.

[25] B. C. Neuman and G. Medvinsky. Requirements for network payment: The netcheque perspective. In *Proceedings of IEEE COMPCON'95*, March 1995.

[26] R. L. Rivest, A. Shamir, and L. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21(2):120–126, February 1978.

[27] M. Roscheisen, C. Mogensen, and T Winograd. Beyond browsing: Shared comments, SOAPs, trails and on-line communities. In *Proceedings of WWW'95*. 1995.

[28] AT&T Global Information Solutions, Digital Equipment Corporation, et al. CORBA security. Technical Report 95-3-3, Object Management Group, March 1995.

[29] Andrew Watson. Object Request Broker 2.0 extensions interface repository rfp. Technical Report 93-9-16, Object Management Group, 1993.

[30] M. Winslett, K. Smith, and K. Qian. Formal query languages for secure relational databases. *ACM Transactions on Database Systems*, 19(4), December 1994.

Acknowledgments

The authors wish to thank Jim Davis from the Design Research Institute, Xerox Corporation, Cornell University, and Bill Arms and David Ely from CNRI for their important contributions to this work. We also wish to thank our reviewers for their help. Finally, we wish to thank Robert Kahn and Robert Wilensky, whose initial work on the digital object framework has enabled this research. The findings discussed in this paper were supported in part by the Advanced Research Projects Agency under Grant No. MDA972-92-J-1029 with the Corporation for National Research Initiatives (CNRI), and in part by funding from the National Science Foundation, the State of Illinois, and NASA Cooperative Agreement Notice (CAN) "Public Use of Earth and Space Science Over the Internet". Its content does not necessarily reflect the position or policy of any of the sponsoring parties, and no official endorsement should be inferred.