3 Standard CORBA Security

Section 3.1 of this chapter gives a general introduction to the CORBA Security Framework. Additional detail is found in [Brose et al., 2001b], [Blakley, 2000], [Hartmann et al., 2001]. Section 3.2 examines the access control model that is defined by the OMG Security Service Specification [OMG, 2001b] , and section 3.3 evaluates the Security Service with respect to the requirements motivated in chapter 2.

3.1 Overview of CORBA Security

The OMG Security Service Specification defines a general security reference model that describes security concepts and terminology and an architecture that outlines how the model should be implemented. The service addresses authentication, access control, auditing, integrity, confidentiality, and non–repudiation. It defines three distinct sets of interfaces: application programming interfaces (API), administrative interfaces for service administrators, and "implementor's interfaces," which guide the implementation of the Security Service. Management services are not defined.

One goal in the design of the reference model was to be "sufficiently general to cover a very wide variety of security policies and application domains" [OMG, 2001b, p. 2–28]. Another design goal of the specification was to support the use of existing security technology rather than requiring the implementation of entirely new security functions. Consequently, the Security Service does not introduce new security mechanisms and its service interfaces generally provide a superset of existing security technology like DCE security [Transarc, 1993], Kerberos [Neumann and Ts'o, 1994], and SESAME [Parker and Pinkas, 1995]. The only additional protocols defined by the specification serve interoperability purposes and do not add core security functionality. The last major design goal is supporting both security–unaware applications, for which security is enforced transparently, and security–aware applications that need to control security functionality themselves. The Security Service distinguishes between these levels of functionality and calls transparent functionality *security level 1*, whereas the functionality used by security–aware applications is called *security level 2*.

The remainder of this section examines the different components of the model. The non–repudiation service mentioned below is an optional package, so implementations of the Security Service need not include it.

3.1.1 Principals and Authentication

Principals are defined in the Security Service as follows (p. 2–3):

> An active entity [...] must either be a principal, or a client acting on behalf of a principal.
>
> A principal is a human user or system entity that is registered in and authentic to the system. Initiating principals are the ones that initiate activities. An initiating principal may be authenticated in a number of ways, the most common of which for human users is a password. For systems entities, the authentication information such as its long-term key, needs to be associated with the object.
>
> An initiating principal has at least one, and possibly several identities (represented in the system by attributes) [...]. There may be several forms of identity used for different purposes.

The defining feature of a principal is that it is "authentic to the system," much like the definition of a principal as an authenticatable entity in [ISO/IEC, 1996a]. It is unclear, however, whether "the system" is a global or a local notion here. Initiating principals are the sources of activity. Principals have potentially many identities that have different uses in other parts of the security model, such as denoting the initiator of a request for access control. Principals can also be system entities, but the term is left undefined. Unfortunately, the implicit equation between a system entity and an object in the following sentence is misleading: "For systems entities, the authentication information [...] needs to be associated with the object." Since "object" cannot refer to CORBA objects — as explained in the following paragraph — the only valid interpretation of this statement is that authentication information needs to be associated with the entity.

In fact, authenticating CORBA objects as active entities is not meaningful — and not even possible in most cases. By "active entity" we understand an entity that can autonomously cause system activity, which implies that it has an independent thread of control. By definition, CORBA objects are abstract entities identified by object references. This abstraction is implemented in server processes that answer requests to objects and host object implementations called "servants," which they map to abstract objects using Object Adapters. CORBA objects are not defined to have their own thread of control and thus cannot independently initiate requests without being invoked themselves. Rather, requests are initiated by *processes* which may or may not host any objects, so objects are not identifiable as the sources of request in the general case. Technically, it is only possible to identify an object as the source of a request at one particular stage: when the request originates from within the processing of another request. A this stage, the target of the original request is known, but authenticating this object as the request source does not appear to be meaningful. Moreover, CORBA does not define any standard object adapter interfaces to associate objects with authentication information. The source of the activity in this case is not the object itself but some principal, either the one on whose behalf the object is invoked or the one on whose behalf the process is running, if any.

Unlike in [Lampson et al., 1992], authentication is not the identification of the source of a request by a reference monitor at the target. Rather, authentication means establishing a set of security attributes (the principal "identities") in the form of *credentials* at the client–side and is performed by a component called PrincipalAuthenticator. Depending on the underlying security mechanisms, it may be necessary to verify these credentials again when they are used, e.g., during the establishing of security associations or for performing access control.

Figure 3.1 depicts a credentials object, which contains unauthenticated attributes and *privilege attributes* in addition to the aforementioned identity attributes. Privilege attributes are those properties of a principal that are used for access control and include an access identity, roles, groups, clearance levels, capabilities, and other arbitrary privilege information.

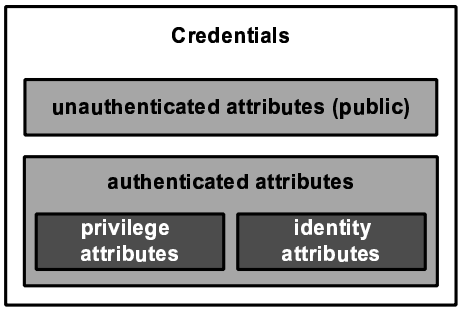

Figure 3.1: Credentials.

The internal format of credentials, how they are exchanged, used, and whether or not they are accepted in particular circumstances is defined by the individual model components that rely on credentials. This lack of a standardized format gave rise to interoperability problems if, e.g., a reference monitor wants to check credentials provided by a client relying on a different vendor's security service implementation. These problem will be treated in more detail in section 3.1.3.

3.1.2 Secure Invocation

Since all activities in a distributed, object–oriented system are based on invocations, the main notion of the Security Service is that of a *secure invocation*, i.e., the application of security functions to ensure that security requirements related to invocations are met. Figure 3.2 illustrates how a client invokes a request that is mediated by the ORB. In general, this involves setting up security associations between the client program and the target before additional security functionality, depending on the effective security policies, is applied for securing the invocation.

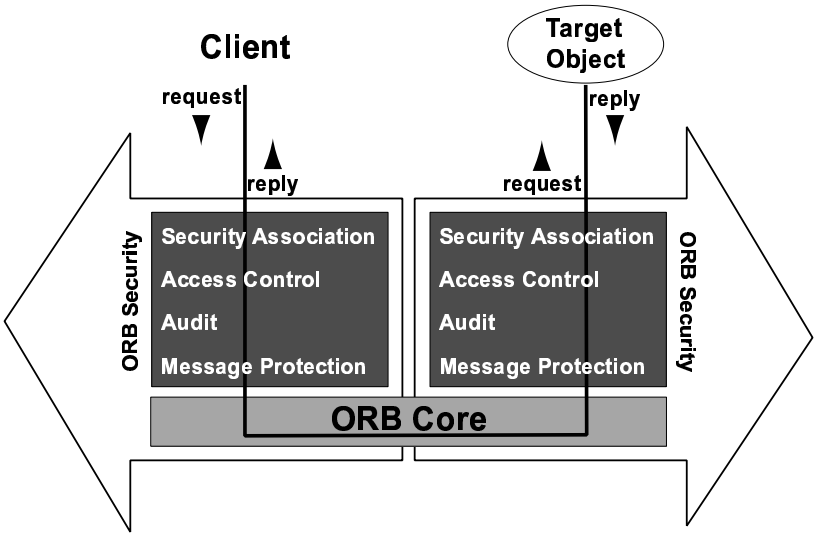

Figure 3.2: Secure Invocation.

Establishing Security Associations

Establishing a security association between a client program and a target object involves establishing trust in the identities of the communication partners, which may involve mutual authentication, e.g., the target could authenticate the client's identity attributes and vice versa. Again, note that authenticating the target does not mean authenticating the target *object*, but the credentials of the target process, if any. This step depends on the security policies that govern the client and the target object, and on the security mechanisms available on both sides. Furthermore, the client needs to make its credentials available for use by other security mechanisms, e.g., access control. Finally, a security context can be established that will be used for protecting messages. This last step may involve exchange of cryptographic parameters and again depends on policies that specify the required quality of protection.

Access Control

The overall authorization model used in the Security Service is that of [ISO/IEC, 1996b] and is depicted in figure 3.3. An access decision function receives different kinds of access control information as input. Three types of access control information are distinguished. The request and context information describes the access request and its context and includes information such as the operation name, parameter data, and the time of day. Initiator–bound information refers to a principal's privilege attributes, such as a clearance–level or access identity, and target–bound information or control attributes refer to the representation of the access policies effective for the target object.

Access control can be performed at two different levels according to different policies. The

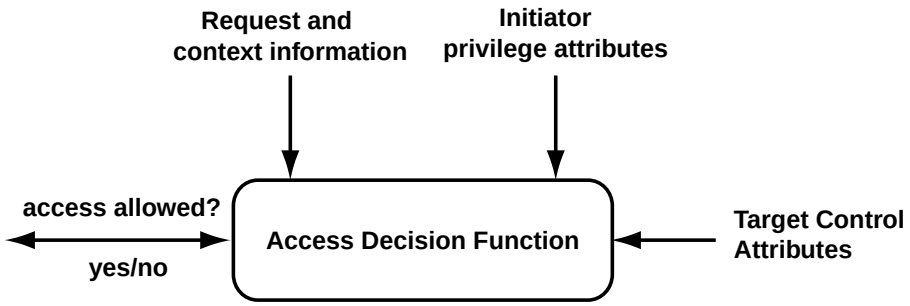

Figure 3.3: ISO Access Control Model.

object invocation access policy is enforced by decision functions in the security service layer and is transparent to applications. The *application object access policy* is enforced by decision functions within the application itself and can be used to control accesses to finer–grained internal resources that are not objects.

Access decision functions can be allocated both on the client and the target side. Client–side access control can be used to prevent access requests from being sent at all, e.g., to prevent information from being transmitted, which is similar to *refrain policies* in [Damianou et al., 2001]. It can also be seen as an optimization to avoid unnecessary traffic in the first place, if the client runtime knows which accesses would be rejected anyway. Access policies are examined in more detail in section 3.2.

Auditing

Auditing means logging information about specific events in the system and potentially reacting by raising alarms or other actions. Which events are audited under which circumstances is controlled by audit policies. As with access control, there are two different layers at which auditing can be performed. *System audit policies* control which system events cause audit reactions and *application audit policies* control reactions to application–level events.

Message Protection

Messages, i.e., requests and responses, may require protection for integrity and/or confidentiality. Integrity protection ensures that any potential modifications of messages in transit, the insertion or deletion of messages during transmission, and message reordering are detected. Confidentiality protection means encrypting messages to prevent their contents from being observed by a third party. The desired quality of protection, e.g., the type of encryption algorithm and the key length, is specified in a security policy.

3.1.3 Delegation and Secure Interoperability

A well–known problem in distributed systems is controlling the delegation of access rights. In its original form, delegation means that a grantor of rights explicitly passes on rights he possesses to a grantee, who may or may not need to further delegate these rights. Depending on the actual system, the original grantor may have the right to revoke rights from grantees, which may result in cascading revocations of rights if the grantee has passed on granted rights to other grantees. The analogy in the authentication logic of [Lampson et al., 1992] is the *hand–off*, whereby a principal A states that another principal B "speaks for" him, so that B's statements (requests) can be taken as if made by A.

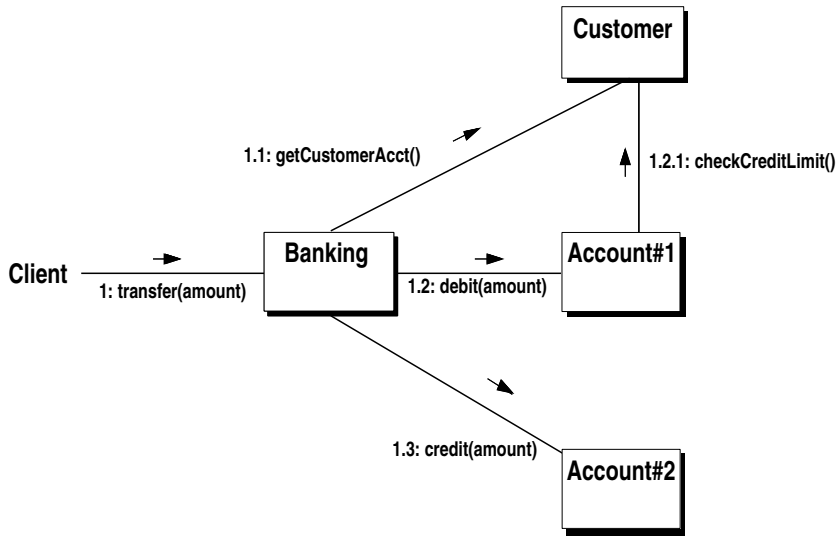

Figure 3.4: Delegation of Invocations.

In a distributed object–oriented system, the need for delegation often arises implicitly because the invocation itself is delegated. Figure 3.4 illustrates such a situation, where the primary target of an invocation has to rely on other objects to complete the execution and becomes an intermediate. If access control is performed at the final targets, access control information has to be delegated with the invocation to enable attribution of the request to the initiating principal. Because the elements that might need to use delegated access rights are generally not known in advance, delegation should ideally happen transparently and on demand. At the same time, however, fine–grained control over the delegation of access rights is necessary because the intermediate elements in a large–scale distributed system cannot be trusted a priori. Different kinds of control might be necessary, e.g., control on the set of grantees that may receive rights, on the target objects and operations that these rights might be used to access, and on whether these rights may be further delegated or not.

In order to be able to specify this kind of control, advance knowledge about the intermediates involved in the processing of a particular request would be needed, but the intermediates

are determined both by the implementation of the operation in the first target and the configuration of the system as a whole. Both these factors are deliberately hidden from the client by the general principle of data abstraction in object–oriented programming. For the initiator of the call — or some component in its runtime system — to perform an appropriate delegation of rights before the actual invocation, it would be necessary to determine the complete set of required access rights for a particular invocation in advance.

This is not possible for two reasons. The first reason is a technical one: the set of required access rights would need to be determined dynamically in a complete round–trip visiting all involved intermediates and targets. Thus, a separate protocol for this pre–invocation check would need to be added to the original CORBA invocation protocol, but the performance overhead introduced by this approach renders it impractical. The second reason is a conceptual one: not all intermediates are principals, so they may not be able to receive and later use granted rights. Explicit granting of access rights is thus not generally applicable in the case of delegated invocations.

3.1.3.1 Delegating Credentials

The only technically viable solution in such a setting is based on the delegation of credentials with the invocation. This is the approach taken in the Security Service specification. The Security Service proposes the following terminology for delegation models:

1. *No delegation*: A client does not permit intermediates to use its privileges in delegated calls. This might result in loss of service if access control is enforced at the final target and the intermediate does not have sufficient privileges of its own to complete the call.

2. *Simple delegation*: A client permits an intermediate to assume its rights for both accessing other objects and further delegation. Target objects do not see the intermediate but only the originator.

3. *Composite delegation*: As in simple delegation, but the intermediates' privileges, if any, are also passed on to the final target, which can now check these separately.

4. *Combined privileges delegation*: The initiator's and the intermediate's privileges are combined into a single set of credentials, so the target cannot distinguish which privileges originate from whom. Note that this can also result in an amplification of rights because the combined privileges might permit accesses that neither of the individual privileges might have permitted.

5. *Traced delegation*: like composite delegation with several intermediates, so the target receives a chain of credentials containing the initiator's and all intermediates' credentials.

The effective delegation scheme to be used is selected by an administrator. Security–aware applications can also set the delegation scheme according to application policy. The security

service mentions, but does not specify interfaces for achieving *controlled* delegation by limiting the validity period of delegated credentials, the specific target that may receive credentials, or the number of invocations they may be used for. Restrictions on the target objects are also envisioned, but not currently defined.

Controlled delegation of credentials in a heterogeneous environment without pre-established trust relations is a difficult technical problem. How can the initiator of a request enforce that an untrusted intermediate uses a particular delegation model chosen by the initiator? How can the final target object successfully authenticate credentials that have been forwarded by multiple intermediates? In particular, how can this be achieved if multiple security technology domains are crossed, e.g. , if an initiator uses SSL as the security mechanisms to communicate with an intermediate, but the only way for the intermediate to reach the final target is by using DCE Security?

3.1.3.2 Common Secure Interoperability

To be able to communicate securely at all, each client and target must be able to use at least one common security mechanism. The *Common Secure Interoperability* (CSI) package of the Security Service distinguishes between three levels of interoperability that implementations of the service can provide. These levels support different delegation models and are determined by the security mechanisms that can be used to establish security associations between initiator and target object:

1. *CSI level 0* means that no delegation is possible and security policies can only rely on a single identity attribute.[1] No other privilege attributes are transmitted. This level of interoperability is achieved by using, e.g., SSL as the security mechanism. A second mechanism that provides CSI level 0 only is the CORBA definition of the Simple Public Key Management API (SPKM) [Linn, 1993] in conjunction with the SECIOP protocol defined by the Security Service.

2. *CSI level 1* also transmits principal identity only, but here delegation is possible. However, delegation cannot be controlled, so any intermediate may assume the initiator's identity for interacting with other intermediates or the target. This level of interoperability is achieved when GSS Kerberos [Linn, 1996] is used in conjunction with SECIOP. This mechanism can also be used to provide only CSI level 0 interoperability.

3. *CSI level 2* finally supports the transmission of arbitrary privilege attributes, which are required by more flexible security policies. It also supports controlled delegation. The only security mechanism providing this level of interoperability is SESAME [ECMA, 1996], [Parker and Pinkas, 1995].

   Technically, delegation of credentials, which are represented as *privilege attribute certificates* (PAC) can be controlled by passing a *control value* with the PAC. Targets will

[1] For public key based mechanisms, the value of the identity attribute is an X.500 distinguished name, for Kerberos–based mechanisms it is a Kerberos name qualified with a realm name.

only accept PACs if the client also proves knowledge of the control value, so not transmitting this (encrypted) value with the PAC will prevent the receiver of the PAC from using it. Effectively, this value corresponds to SQL's grant option. More fine–grained restrictions on the use of PACs are also possible.

The OMG has recognized a number of shortcomings in the CSI section of the Security Service specification and therefore issued a *Request for Proposals* (RFP) [OMG, 1999b] that calls for a companion specification. The main problems are that the format of credentials is dependent on the individual security mechanisms so that credentials have no uniform interpretation and must be mapped between different formats whenever a security technology domain is crossed. Moreover, they are tightly coupled with transport layer authentication.

The RFP explicitly calls for a solution that separates client authentication (during association establishment), message protection, and authorization into three distinct layers. The proposed specification for CSI version 2 [OMG, 2001a], which is currently in its finalization stage at the OMG, achieves this separation through the definition of a Security Attribute Service (SAS) protocol. SAS protocol messages are transmitted in a header field of standard CORBA GIOP request messages and are thus independent of the transport layer security protocols, e.g., SSL or SECIOP, which are used for client authentication and message protection.

CSI version 2 can rely on client identities that were authenticated by a secure transport layer, but the SAS protocol also supports an additional client authentication function, so that it is possible to leave out transport security if communication links are not considered unsafe in the target environment. However, target authentication is not fully supported in the SAS protocol, so if mutual authentication is required, the SAS protocol alone is not sufficient. Authorization information ("privilege attributes") in protocol messages is transmitted encoded as privilege attributes in X.509 certificates.

Additionally, the protocol supports "identity assertion" as a means of delegation. Identity assertion means that the calling client, which may act as an intermediate in a chain of invocations, explicitly asserts another identity by including it in a particular field in a protocol message. The client may or may not be authenticated. In a typical situation, this identity will be the subject of one or more privilege attribute certificates transmitted in the authorization information of the same protocol message. The target has to decide whether it accepts the client as "speaking for" the asserted identity and thus also accepts the attributes as applying to the client. CSI version 2 defines three cases in which the target accepts the identity assertion. The first is called "presumed trust", in which the target blindly accepts because it is assumed that only trusted entities can use identity assertion. In the second case the target simply trusts the authenticated client. Finally, the target may accept the client's asserted identity because the attribute certificates contain delegation rules that permit the client to assert the identity to which the certificates originally apply, i.e., the certificate subject. Such a rule corresponds to the *hand–off* of [Lampson et al., 1992].

The SAS protocol defined in the CSI version 2 specification thus supports a form of controlled delegation of privilege attributes. It does not, however, provide means to dynamically delegate privileges on demand: Delegation rules or hand–offs have to be provided in advance

by the authorities defining the privilege attributes. There is no finer–grained control over delegated privileges such as restricting their use to specific targets or even specific operations on specific targets. Also, the identity that is being asserted has no control over the identity assertion, only the authority that issued attribute certificates. The specification also does not address the problem of revoking delegated privileges. Since there are no certificate revocation lists or additional verification steps in the protocol definition, it appears that the propagation of these privileges can only be controlled via certificate life times.

3.1.4 Non–Repudiation

Non–repudiation is defined as an optional service in the specification and only mentioned here for completeness. It is not used automatically but only provided to security–aware applications that request this service. It is used to generate irrefutable evidence that an event has occurred, which can later be used to protect against false denials. For example, the sender of a message might want to prove that it did send a specific message at a given time, so the service is used to provide proof of origin. Alternatively, the service can be used to deliver a message and generate proof of delivery to protect against a false denial of message reception. This kind of service relies on one or more trusted third parties. The types of events that cause evidence to be generated and the kind of evidence that is generated are controlled by non–repudiation policies.

3.1.5 Security Policy Domains

For scalability reasons, the Security Service requires management of security policies to be based on *domains* of objects rather than on individual objects. The canonical definition of a domain in the CORBA specification is that of “a distinct scope within which certain common characteristics are exhibited and common rules are observed” [OMG, 2000a, p. 13–2]. The same document, defines a *policy domain* as “a set of objects to which the policies associated with that domain apply. These objects are the domain members.”, (p. 4–44). A *security policy domain*, finally, is “the scope over which a security policy is enforced” [OMG, 2001b]. Objects are domain members and policies represent rules to make the domain secure. A security policy domain is also associated with an administrative authority, which is ultimately responsible both for the management of the domain, its policies and members, and for the enforcement of policies in the domain. Thus, domains “provide a mechanism for delimiting the scope of administrators’ authorities.”

To protect a given set of objects with a particular access policy, these objects are grouped into a domain and the policy is attached to the domain. At runtime, an object’s domains need to be determined to retrieve the necessary policy and to obtain the required authorization information. The CORBA specification defines an operation on object references to retrieve an object’s DomainManager objects. These represent the object’s enclosing domains and provide access to the policy objects that apply in these domains. The specification requires that every domain is only associated with a single policy of a given type so that individual domains have

no conflicts between policies of the same type.

Since domain managers are objects themselves, they can be made members of other domains. In this case, the domain they represent becomes a subdomain of the domain that now contains the domain manager. The outer domain's policies now apply also to the domain manager object. The precise meaning of hierarchically composed *policies* is, however, not specified. The main purpose of forming hierarchies of policy domains is to reflect organizational hierarchies, so subdomains could have more specific security policies than enclosing domains. The specification does not state, however, whether the policies that apply to a domain manager object transitively apply to the member objects of that domain or not. If so, it is not clear how potential conflicts between policies of the same type would be resolved.

Another way of composing policies is by forming federations of interoperating domains, where each domain has full authority but has some amount of trust in the other and agrees to give certain rights to the other side. As with domain hierarchies, the question of how this is done and what the semantics of such a federation would be is left unspecified.

The specification also requires that every object is a member of at least one domain, which means that an object must initially be added to a domain on object creation and can later become a member of other domains as well. There are no standardized interfaces, however, that would allow administrators to configure this initial domain. An object may be a member in multiple domains, but the specifications again define no interfaces to manage object domain membership at all. Also, there are no interfaces for managing the life cycle of domain objects, i.e., to create or destroy domains and to set up hierarchical relations between them.[2]

The OMG recognized these open issues in a Request for Proposals for Security Domain Membership Management [OMG, 1998b]. In addition, the RFP asks for approaches of associating object groups of different granularities with domains, e.g., all objects in a process, POA, or type extension. Initial and even revised submissions to the RFP exist, but the published drafts [Concept Five Technologies and Hitachi, 2000] are still instable and immature and therefore not discussed here.

3.2 The default CORBA access model

In principle, arbitrary access control models may be defined within the general reference model of CORBA security. The Security Service defines one access model as the default model, which it calls DomainAccessPolicy. Individual policies are represented by DomainAccessPolicy objects, but the type DomainAccessPolicy defines the class of policies that can be expressed, i.e., the access model. The default access model is based on the access control matrix [Lampson, 1974], which represents principals as rows, objects as columns, and permissions or access rights in matrix entries. For a detailed description of the access matrix semantics of the

[2] The OMG Relationship Service could be used as an implementation of relationships, but the semantics in terms of the involved policies would still be undefined. The same argument holds for the so–called Life Cycle Service, which is not actually an implementable service but only defines the interfaces for life cycle operations.

default CORBA access control model see [Karjoth, 1998].

As motivated in chapter 1, any component of a CORBA infrastructure must be able to scale up with the applications and the same holds true for a general CORBA access control model. The smallest unit of access control in object–oriented systems is a single operation on a single object, but in many cases this level of granularity would induce too much overhead in operation as well as in management, so a number of aggregation constructs are provided to reduce the number of entities that must be considered.

3.2.1 Principals and Attributes

As described in section 3.1, the Security Service refers to both human users and system entities as principals, which have one or more identities. The default access model regards identity simply as one out of potentially many security attributes of a calling principal, so access rights are assigned to combinations of security attributes rather than to individual principals. Thus, an access decision is generally made in terms of a second, more abstract principal concept, which is not given a name of its own, however. Using security attributes, it is possible to define role–based or label–based policies. Security attributes thus support the aggregation of principals for policy management purposes based on common attributes.

3.2.2 Rights

Access permissions are represented as *rights*. The security service defines individual rights in *rights families*. The default rights family is corba and contains four generic rights: g, s, m, u for *get*, *set*, *manage* and *use*. The definition of new rights families, albeit possible, is discouraged to keep policies simple. There is no notion of negative rights or denials.

Permissions are checked per individual operation. The default access model defines no explicit grouping construct for operations that should be treated alike. Operations are grouped implicitly by the access rights they require, however, because access will be granted or denied for all of these operations alike. The *required rights* for a particular operation are intended to be defined by interface developers and specified per object type, not per object. Thus, the specification of required rights defines a mapping from generic access rights to actual operations. Using *combinators*, required rights are defined as a combination of rights by stating whether the given rights are required in conjunction (“all"–combinator), or whether the presence of one of the listed rights is sufficient to be granted access (“any"–combinator). Required rights are stored in a global table and not associated with any policy domains, so they are in fact a policy–independent concept to categorize operations.

Using a domain’s DomainAccessPolicy object, a particular access policy is defined by granting a set of *effective rights* to some combination of security attributes. One implicit security attribute that is set by the ORB for every access is the *delegation state*. This attribute identifies for every request whether the principal is the initiator of the request or an intermediate in a call chain. It is thus possible to grant a different set of effective rights depending

on the delegation state of the caller. Effective rights are registered in policy–local tables and compared to the required rights for an operation upon access. If a caller's effective rights match the specified combination of required rights for the target object's type, the access is allowed.

3.2.3 An example policy

This section sketches an example access policy for naming context objects using the Security Service's default access model. Naming contexts have the interface CosNaming::NamingContext from the OMG's name service specification [OMG, 1997]. The IDL definition for this interface is given in figure 3.5.

```
interface NamingContext {
    void bind(in Name n, in Object obj);
    void rebind(in Name n, in Object obj);
    void bind_context(in Name n, in NamingContext nc);
    void rebind_context(in Name n, in NamingContext nc);
    Object resolve (in Name n);
    void unbind(in Name n);
    NamingContext new_context();
    NamingContext bind_new_context(in Name n);
    void destroy();
    void list (in unsigned long how_many,
            out BindingList bl, out BindingIterator bi);
};
```

Figure 3.5: Name server interface.

The example policy distinguishes between different uses of naming contexts. Figure 3.6 shows an extended use case diagram that illustrates six different use cases for the NamingContext interface, the respective operations, and the relationships between these use cases.

Basically, there are three kinds of uses of the interface: resolving names, binding names, and managing the context. Name resolution can be subdivided into the two use cases *Resolving* and *Traversing*. The *Resolving* use case is the simplest and only requires the right to invoke resolve(). It is used frequently by CORBA client programs that retrieve server references in a given naming context using a name. These client programs usually do not need to traverse a graph of naming contexts or bind names to references. A second use case that includes the previous one is *Traversing*, which needs list() access to naming contexts. This operation returns a list of all names bound in the context. If one of the returned names is bound to a nested context, the client can resolve that name and obtain a reference to the nested context. Using list() and resolve(), the entire naming context graph can be traversed, any binding in the entire graph may be inspected and any name in any reachable context can be resolved.

Name binding can be refined into *Binding*, *Rebinding*, and *Extending*. *Binding* is analog to *Resolving* and only requires bind() access to a single context. This is used by typical CORBA

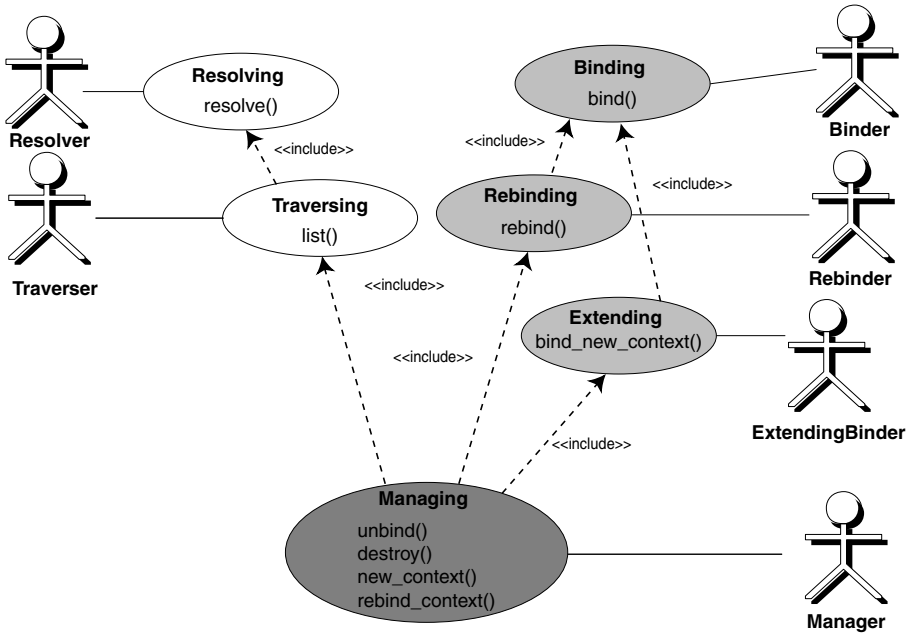

Figure 3.6: Name Service use cases.

services to publish references to objects. Since the bind() operation fails if the name is already bound in that context and there is no unbind() access, only new names can be bound in this use case. An extension of this use case is *Rebinding*, which includes rebind() access, i.e., a bound name can be unbound and rebound to another reference. A second extension of the *Binding* use case is *Extending*, which permits to work around already bound names by creating and binding new subcontexts, in which the name can then be bound. This use case does not need resolve() access because the new context is directly returned by the bind_new_context() operation. The *Extending* use case does not unbind or rebind any names, nor does it extend the naming graph with externally created naming contexts. Thus, only simple extensions of the naming context graph are possible, modifications of existing contexts are not permitted.

The *Managing* use case finally has complete access to all operations on NamingContext objects, in particular the life cycle operations destroy() and new_context(), but also the unbind() operation to remove bindings completely, which also includes bindings of names to subcontexts.

3.2.3.1 Defining Required Rights

Before this policy can be expressed in the default CORBA access model and effective rights can be granted to the appropriate security attributes, *required rights* for each operation of the NamingContext interface have to be defined. The rights combination for each operation can be specified using

1. rights from the default corba rights family exclusively.
2. rights from another family, perhaps one that was introduced for exclusive use with the type NamingContext.
3. rights from a combination of different rights families.

To keep management of rights simple, it would be desirable to refrain from defining new rights families and select the first of these options. However, it is not always straightforward how to model the intended semantics of authorization using the generic rights “get", “set", “manage" and “use". Note that these rights do not just describe distinct operation types (g,s,u) but also the level of sensitivity for operations (the m right) [Blakley, 1998].

A more serious limitation with using only the small set of rights from the corba family is that no more than 16 different combinations of rights can be distinguished. Thus, it is likely that a number of operations will require the same set of rights, especially if the domain contains objects of a number of different types, and if required rights are defined independently by individual interface designers and not by security administrators. If required rights are not unique, however, ensuring the principle of least privilege is very difficult if not impossible with this access model. Because rights are generic, callers can invoke any operation on any object of any interface in the domain that happens to match the effective rights combination granted to them.

As an example, imagine that the operation list() on naming contexts and the operation create() on some object of type JPEGFactory were to require corba:g and corba:u. Any caller that was granted this combination of required rights in a domain where objects of these types exist would be allowed both operations and possibly others as well. Determining which operations will be permitted with a given set of rights implies searching the whole *required rights* table because grouping operations by their required rights can only be done implicitly.

The second of the options outlined above, using rights from a newly defined family, would avoid this problem of interfering rights altogether. Also, defining one new rights family per IDL interface would allow policy designers to use access rights that directly correspond to the intended usages of these interfaces. However, introducing new rights families not only complicates management, it is also very cumbersome in practice as there are neither language support nor management interfaces for this task.

As a compromise, the basic corba rights could be used in conjunction with a minimal set of newly introduced, type–specific rights that help ensuring uniqueness of required rights. For our example, we introduce a new rights family naming with two rights n and m for *naming* and *manage*. The right naming:n is used to make this required rights combination type–specific, the second right is necessary to distinguish between the last two authorization types. These are AND–combined with the default corba rights to specify the required rights as in table 3.1.

Note that the rights required for, e.g., the operation list() include those required for resolve(), or that those for unbind() include all other rights. This models the policy feature that whoever is authorized to invoke list() may also invoke resolve(). Also note that to model this

required rights	combinator	operation
corba :g--- naming:n-	all	resolve
corba :g-u- naming:n-	all	list
corba :-s-- naming:n-	all	bind
corba :-su- naming:n-	all	rebind
corba :-sum naming:n-	all	bind_new_context
corba :gsum naming:nm	all	unbind, destroy new_context bind_context rebind_context

Table 3.1: Required rights.

property of the policy, the approach of using the four corba rights as a consistent classification of operation types had to be given up: The operation unbind() had to require `corba:gsum` although it does not "read" the state of a naming context, as the presence of the right `g` would suggest. The example shows that it is not at all straightforward to reconcile the intended semantics with a descriptive and readable specification using this access model.

Another problem with the above policy is that required rights are specified per type, so all naming context objects in the policy domain are treated alike. This means that it is not possible to give principals full but exclusive access to their own naming context object. This is also the reason why the unbind() operation may only be invoked by administrators in the example. Otherwise, a principal could remove any binding in the entire naming context graph. Also, there is no concept of discretionary access control in CORBA at all.

The definition of required rights is not, in CORBA terminology, the definition of a policy, which is represented by effective rights that are granted to principals. This step is omitted here because the more interesting, application–specific aspects are expressed by the required rights defined above.

3.3 Evaluation of Requirements

The access control–related sections of the OMG Security Service do not qualify as a manageable access control infrastructure as described in chapter 2. The main requirements that it fails to meet are support for management and a scalable access control model that is suitable as

the basis for an expressive policy language. Additionally, the CORBA notion of principals is not immediately applicable to the actor–role concept described in chapter 2, although it can be mapped to it.

3.3.1 Principals and Roles

To see whether the notion of principals as actors or roles can be mapped to CORBA principals, it is useful to first compare CORBA principals with the notion defined in [Lampson et al., 1992].

In that model, principals are the sources of request statements, i.e., the sources of *data* flow. Principals are also the units of access control in that access is granted or denied on the basis of the principal that attempted it. The most basic notion of a principal is a channel over which statements arrive. Channels may be speaking for other principals. If a channel speaking for another principal makes a statement, this does not imply that this principal caused any activity that made the channel make the statement. Rather, it means that this principal can be *concluded* to have made that statement also. This situation is illustrated in figure 3.7. This notion of principals can model basically any caller concept that may be considered useful for writing access policies and thus directly supports abstract policy concepts such as roles.

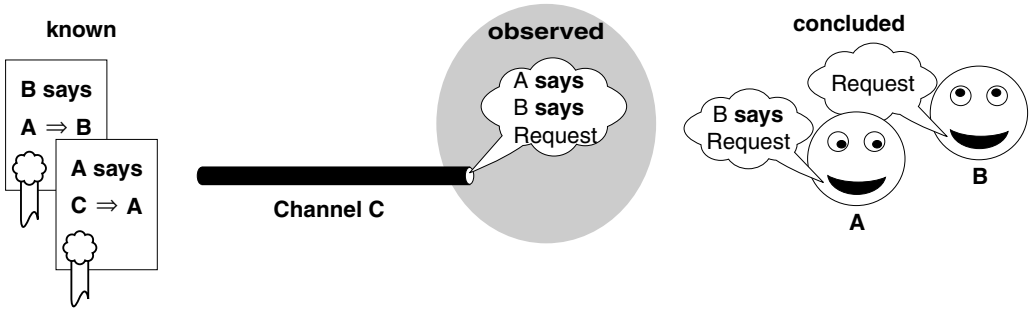

Figure 3.7: Principals make statements.

In CORBA, initiating principals are the sources of activity, i.e., of *control* flow. The prime examples of principals are humans or processes. A client that issues a request need not be a principal, but it must be acting on behalf of one. To associate a principal with a request, it is necessary to track the control flow over which the request arrived back to its origin. Principals are thus the ultimate causes of requests, but not necessarily the sources. In figure 3.8, principal B never invoked `request()`, for example. Also, note that B does not address A in any way but only knows about the object `o1`. In the standard access model, principals are also not the units of access decisions. Effective access rights are not assigned to principal identities but to a combination of security attributes, which can be considered a second, more abstract principal concept for access control purposes, whereas the base principal concept can be seen as a way of implementing it. Since role membership can be modeled through the use of appropriate security attributes, the Security Service can in fact support the role–based policies required in

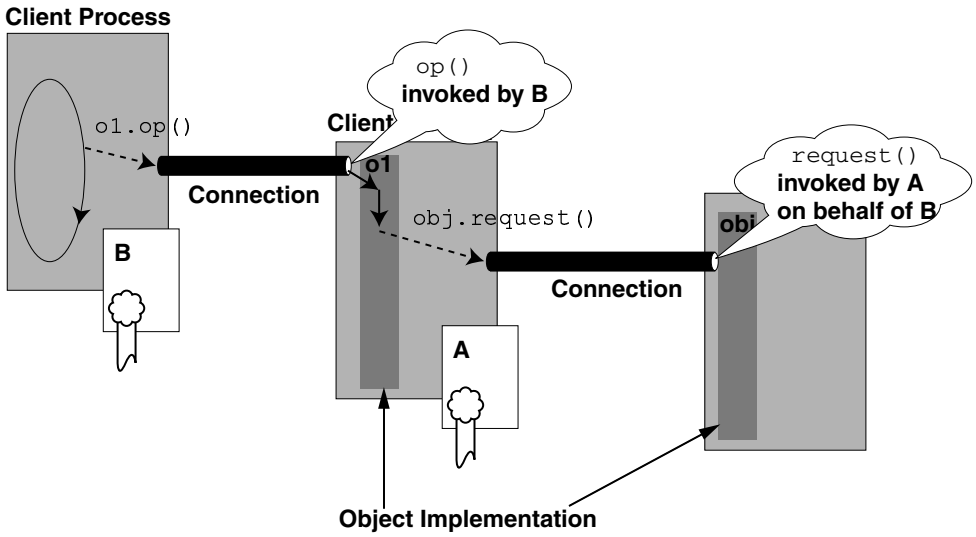

Figure 3.8: Principals invoke operations.

chapter 2.

3.3.2 Management

With the exception of the policy domain concept, the CORBA Security Service does not explicitly address security management issues at all. There is no comprehensive model of management responsibilities and the Security Service does not specify administrative interfaces for credentials management or domain management.

The separation of required rights and effective rights can be regarded as an attempt towards designing an interface between application developers and security managers. Application developers specify required rights based on the sensitivity of operations in terms of access to underlying resources and Security Managers specify access policies in terms of effective rights. However, the precise semantics of these rights and how semantically richer operations have to be mapped to them is left undefined.

3.3.3 Restrictions of the access model

The default access model exhibits a number of restrictions that make it unsuitable as the basis of a policy language for application–level policies. First, the default domain access policy does not provide adequate support for very fine–grained policies. It is not possible to specify different *required rights* for individual object instances, nor is it possible to differentiate between domains by specifying different required rights. If a policy were to require that individual instances of a type be treated differently, different domains would have to be set up for each

object. This would lead to a proliferation of potentially very small domains when fine–grained access control is necessary, however, and thus complicate management.

Second, the model is not scalable up to systems with large numbers of objects and interfaces because the authorization requirements expressable using the CORBA rights concept will frequently overlap. Because rights are used as *generic* access types and can practically not be tied to particular IDL types, enforcing the principle of least privilege is not easily possible within domains. By granting a set of effective rights, principals will almost invariably gain more authorizations than they actually require if only the small set of rights defined in the corba rights family are used — many operations will happen to require the same rights combination.

Third, the default access model does not define any structural relations for its access control concepts, e.g., hierarchies of groups, rights, or domains, nor does it support the specification of constraints such as mutual exclusion of rights. Also note that the management interfaces to the default domain access control model are at a low level of abstraction and do not offer any language support for the sensitive and error–prone tasks of specifying and managing access rights. Other than domains, no abstractions are defined that would help structuring large specifications.