

Practical delegation for secure distributed object environments

This article has been downloaded from IOPscience. Please scroll down to see the full text article.

1998 Distrib. Syst. Engng. 5 168

(<http://iopscience.iop.org/0967-1846/5/4/004>)

View [the table of contents for this issue](#), or go to the [journal homepage](#) for more

Download details:

IP Address: 38.107.179.211

The article was downloaded on 20/02/2012 at 07:52

Please note that [terms and conditions apply](#).

Practical delegation for secure distributed object environments

Nataraj Nagaratnam[†] and Doug Lea[‡]

[†] Department of Computer Engineering, Syracuse University, NY 13244, USA

[‡] Department of Computer Science, SUNY Oswego, Oswego, NY 13126, USA

Received 1 July 1998

Abstract. SDM is a secure delegation model for Java-based distributed object environments. SDM extends current Java security features to support secure remote method invocations that may involve chains of delegated calls across distributed objects. The framework supports a control API for application developers to specify mechanisms and security policies surrounding simple or cascaded delegation. Delegation may also be disabled and optionally revoked. These policies may be controlled explicitly in application code, or implicitly via administrative tools.

1. Introduction

Open distributed computing environments must address four symmetrical security issues:

Services need not trust users. For example, a database service may require that only certain users be able to modify records.

Users need not trust services. For example, a person using an unknown word-processor application may not wish it to delete existing files.

Users need not trust users. For example, a system administrator may only transiently allow an ordinary user to access a resource such as a tape drive.

Services need not trust services. For example, a distributed database service may limit rights of different application programs that use it.

This paper describes the delegation-based mechanisms that underlie a proposed framework, the *Secure Delegation Model*. SDM integrates support for these different aspects of security in Java-based distributed systems.

SDM is an architectural framework for structuring remote method invocations (RMI) among distributed components. It does not involve new encryption techniques, authentication protocols or language constructs. SDM instead builds upon existing mechanisms, mainly those already established in the Java JDK1.2 security framework, to establish a practical basis for constructing flexible yet secure components and support infrastructure.

This paper focuses on the way in which delegation is structured and used in SDM to support secure operation when multiple components together provide a given service. Other aspects of the framework are described only briefly. Readers may find further details in [7].

The remainder of this paper is structured as follows. Section 2 defines Java-based security concepts

and terminology surrounding principals, permissions, privileges, roles and security domains. Section 3 introduces the SDM delegation framework. Section 4 describes the details of the resulting protocols, which are extended in section 5 to handle dynamic revocation of delegated privileges. Section 6 briefly compares SDM to other approaches.

2. Concepts and terminology

Principals. All parties associated with secure computation in SDM are known via *principals*: identities (unique names) that can be authenticated. We further restrict attention to *scoped* principals, for example *Syracuse's Nataraj*, where the scope represents an organizational domain (which may in turn be further structured and scoped in any fashion). Principals are most often associated with individual people. However, they may also be associated with entities such as departments (as in *Acme's MarketingDept*), entire companies, or any other authenticatable unit. In SDM, we further categorize principals in terms of the properties and usages as discussed in the remainder of this paper and implemented via the classes and interfaces illustrated in figure 1.

Signing. Java software components may be *signed*. The *CodeSource* associated with a component (i.e. one or more related classes) includes a set of signers recording the principals who developed that piece of code, or those who authorize the validity of the code. In the current Java model, a *CodeSource* encapsulates a set of signers who signed the class files, and the URL representing the location from which those class files are to be downloaded. Access can then be controlled based on such a *CodeSource*.

CodeExecutors. A signed component may be obtained from a software vendor and then executed by a variety

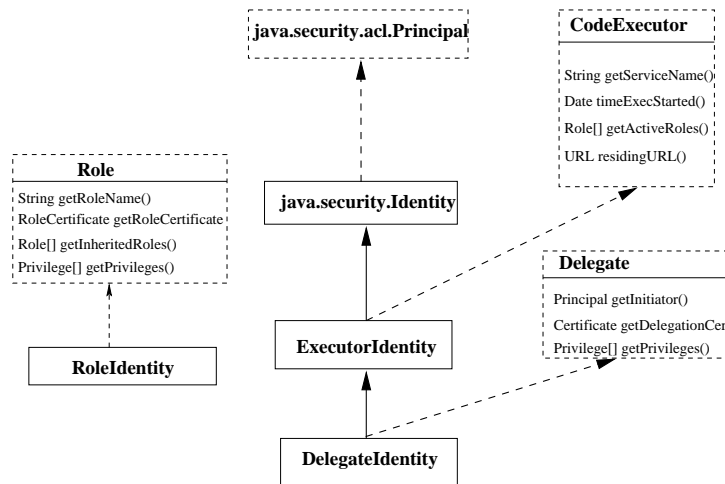


Figure 1. Principals in SDM.

of users. Normally, the principal executing the code is different from the one that signed the classes. To clarify the resulting distinctions, we introduce the concept of a *CodeExecutor* to be the principal invoking a given service, and upon which authentication, delegation or access control can be based.

Permissions. A *permission* is a named value conferring the ability (or formal consent) to perform actions in a system. We focus mainly on permissions based on access control policies, that grant permissions to principals on the basis of security attributes or privileges typically maintained via access control lists (ACLs). In order to make a control decision, access decision functions compare the permissions granted to a principal against the permissions required to perform an operation. For example, permission to read a file `/tmp/foo.txt` can be denoted as

```
FilePermission: read:/tmp/foo.txt.
```

Privileges. A *privilege* is a security attribute which may be shared by possibly many principals. We focus on the kinds of privileges defined in the XGSS and CORBA specifications, that include groups, roles, clearances and capabilities [8]. For example, Bill Clinton might have the privileges:

```
role: President-of-USA
capability: OccupyWhiteHouse
group: AmericanPresidents
accessId: WilliamClinton
```

Note that the permission to occupy the White House may be a capability transiently issued to him, with an expiration at the end of his presidency.

2.1. Roles

A given person or principal need not always have the same set of privileges. Rather than continually change them across different contexts, it is convenient to introduce

the notion of a *role*, a set of actions and responsibilities associated with a particular activity [12] that might be adopted by any principal. A role is normally represented as a set of privilege attributes that a principal or set of principals can exercise within the context of an organization. The notion of a role does not add any power to a security framework, but instead improves manageability by adding an optional level of indirection. Role-based access control provides a higher level of granularity than approaches limited only to individuals. Because roles make transient privilege assignment much easier to administer, they have been widely adopted in security frameworks.

Role certificates. A *role certificate* is an authenticatable device that provides evidence that a given principal possesses the attributes of a given role. In SDM, an executing Identity adopting a role is represented as a *RoleIdentity*. A *RoleIdentity* contains a *RoleCertificate* within it that it can be presented to any server. *RoleCertificates* have associated names and privileges, along with any other role hierarchy information; for example, rules stating that all managers are also employees. When a principal authenticates itself and presents a valid role certificate, the privileges associated with that role become effective for the principal.

Adopting roles. Roles may be used to obtain both extensions and reductions of privileges [1]. Reductions are typically performed in accord with a ‘least privilege’ policy in which principals have only the privileges they need to accomplish a given task. Examples include:

- An administrator may want to have the power of an ordinary user most of the time, except when performing installation or user account creation.
- Users invoking untrusted software might want to reduce their powers before doing so.
- Users may wish to delegate only some of their privileges to others.

A principal *A* may adopt role *R* and act with the identity (**A as R**) when transiently obtaining or reducing powers. The privileges associated with a role work in the same way as those associated with principals. For example, a Manager role might have privileges:

```
group: CEOAnnouncementRecipients
group: companyBudgetReviewers
capability: MakeAppointmentOffer
-grantedBy Company
capability: ChargeCompanyCreditCard
-grantedBy Company
```

A principal plays a role by associating itself with one of its roles for a particular period of time. Thus, these privilege attributes must become associated with the principal. In SDM, this is accomplished by querying the RoleIdentity for its privileges.

Multiple roles. A given principal can play multiple roles at the same time. So long as those selected roles are allowed to co-exist (i.e. they are not mutually disjoint roles), the principal can exercise the roles simultaneously, and thus obtain the union of privileges associated with them. To extend the above example, in a company intranet environment, access to a budget information file might be limited to the group named companyBudgetReviewers. A principal who has been assigned the role of a Manager can access this information, since its privilege contains the group membership. This group membership need not be explicitly assigned to the identity, but can just be associated with a role, in this case Manager. Similarly, the capability to make an offer to a candidate is automatic for a Manager as it contains the capability MakeAppointmentOffer having been granted by the company itself.

2.2. Domains

Protection domains. A *protection domain* is an administrative scoping construct for establishing system and service security policies. The Java 1.2 security architecture provides support for protection domains and domain-based access control. Currently, the creation of domains is based on a CodeSource indicating a URL and code signers. SDM extends this framework to include explicit support for principals.

Principal domains. In SDM, each service is run on behalf of some principal, the CodeExecutor, who takes the responsibility for that service. In particular, given a remote service running on a machine at a port (mapping to a URL), there is an authoritative CodeExecutor responsible for that service. Implementation of SDM requires that the JDK1.2 domain model be extended to include principals, so that each CodeSource will also have a principal associated with it. One domain will be formed for each such $\langle \text{Code-Executor}, \text{CodeSource} \rangle$. Further authentication and access control (and delegation) may then be based on the Code-Executor.

To support PrincipalDomains, the Java runtime system must maintain a mapping from the $\langle \text{CodeSource}, \text{Code-Executor} \rangle$ pair to their protection domains and also the mapping between protection domains and their privileges. This could, for example, be implemented at the execution stack level with the aid of class blocks and the executing environment frame, as illustrated in figure 2. More complete details can be found in [6, 7].

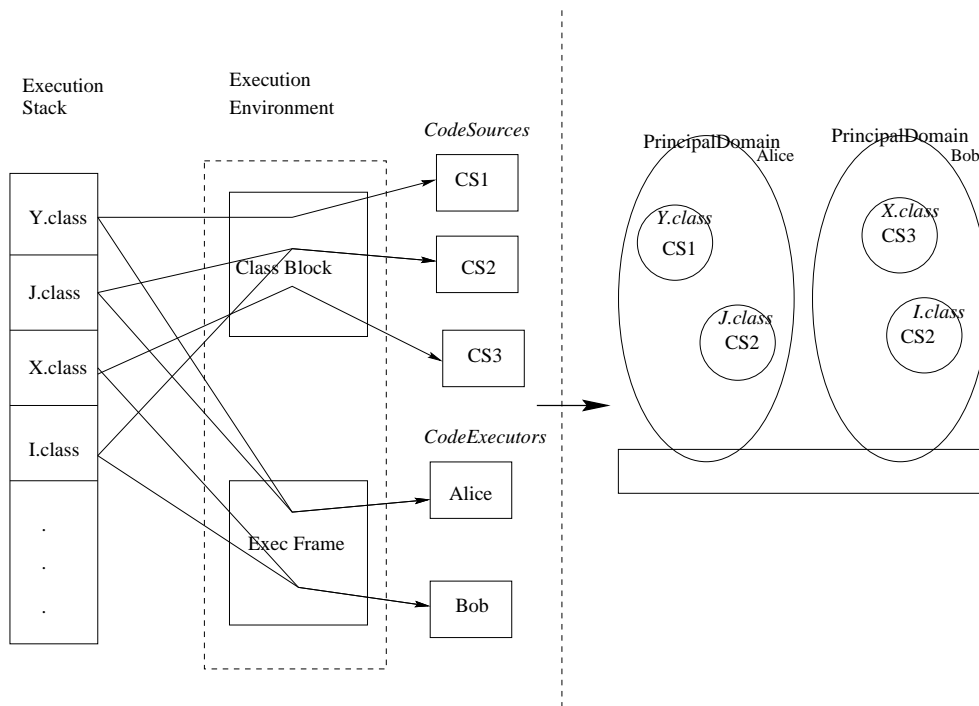


Figure 2. Obtaining domain information from the execution stack.

3. Delegation

Secure delegation occurs when one object (the delegator or initiator) authorizes another object (the delegate) to perform some task using (some of) the rights of the delegator. The authorization lasts until some target object (endpoint) provides the service. The essence of secure delegation is to be able to verify that an object that claims to be acting on another's behalf is indeed authorized to act on its behalf [15].

The problem becomes more complicated in practice when we consider mobile objects, agents and downloadable content being passed around an open network, where the initiator need not have a clue of where all its representative objects are passed around. Additionally, a number of practical issues must be solved: the framework must be scalable in wide area networks, remain efficient under widespread use, and remain secure when dealing with complex trust relationships that can emerge in practice. Toward these ends, SDM provides a multifaceted approach, supporting any of several styles and protocols, including both simple (impersonation) and cascaded (chained) delegation, as well as means to disable and revoke delegation.

3.1. Protection domains and delegation

In Java (as of release 1.2), a protection domain is created for each CodeSource. In SDM, this notion is extended to form *PrincipalDomains* based on CodeExecutors as well. A target (or intermediate) controls access to its methods based on protection domains, i.e. the $\langle PrincipalDomain, ProtectionDomain \rangle$ pair. Access is then controlled via the permission associated with both the CodeExecutor and/or CodeSource.

SDM delegation protocols are based on the notion that when a client delegates its rights to one object in a

domain (i.e. when it enables delegation before invoking on a target object), it effectively delegates its rights to all the objects in that domain. This is implemented via *DelegationCertificates*, that behave analogously to RoleCertificates. In particular, a DelegationCertificate passed to a delegate can only be used by the object it is issued for.

A set of security *requirements* is associated with each object. If an intermediate object needs delegation from an initiator, it *specifies* the delegation mode in its security requirements. Depending on the context (see section 4), a delegation session may be established. If the target does not need to further delegate actions, no DelegationCertificate is generated by the client.

When initiating a delegation session, information about the initiating principal (CodeExecutor) is associated with the context of invocation. This is propagated through the underlying layer to the remote server (target) and is associated (principal and CodeSource pair) with a protection domain. The target may provide access based on the identity of an individual or based on privileges it has (based on its effective role during invocation).

3.2. Modes and chaining

A series of objects may be involved in a given service request. For example, suppose some object *A* (client) invokes a method on another object *B* (target). Object *B* might complete the task on its own or might in turn invoke a method on another object *C*. In this context, object *B* which was earlier the target (for *A*'s invocation) becomes a client for the method invocation on object *C*. Thus objects that are at first targets may later become clients. This effectively forms a *delegation chain* where object *A* is the *initiator*, object *C* is the *final target* and object *B* is an *intermediate*.

There are three different approaches, or *modes*, that may apply to such chains (see figure 3):

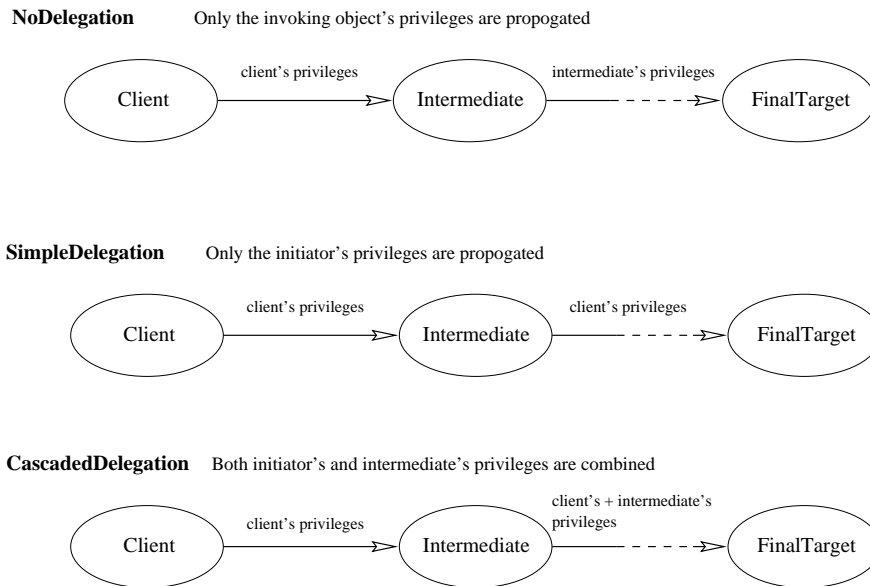


Figure 3. Delegation chaining.

```

public class SystemAdmin {
    :

    private void getSystemInfo() {
        :
        // enable super user Role to obtain information
        AccessController.enablePrivileged(SUPER_USER);
        systemA.getSystemLog();
        AccessController.disablePrivileged();
        :
    }
}

```

Figure 4. Enabling roles.

NoDelegation. The intermediate exercises its own rights for further access.

SimpleDelegation. Impersonation; either restricted or unrestricted.

CascadedDelegation. Combining rights of initiator and delegates.

After obtaining the delegation certificate from a delegator, an intermediate object might invoke a method on another object down the chain. At this point, the intermediate may decide to use only the delegator's privileges or combine them with its own privileges. This decision of either passing delegator's privileges only (impersonation) or combining its privileges too (composite) is based on the delegation mode specified for the intermediate object. Mode specification may be explicit through the application, or may be implicitly set by the administrator of that object service.

3.3. Controlling delegation

Objects can explicitly enable delegation at the application level. This is accomplished by using an `AccessController` object. The `AccessController` method `enablePrivileged()` permits delegation. The method `enablePrivileged(RoleType)` is similar, except that when a role type is passed, the available privileges for that session are extended or restricted to the privileges associated with that enabled role. This functionality is not restricted to delegation. It can also be used whenever access to local methods and resources needs special control. For example, consider a system administrator who logged in as a normal user but would like to exercise super-user privileges for an account creation. In this case, the method (which has authenticated the user to have super-user privileges) would invoke `enablePrivileged(superUser)` to enable super-user privileges, as illustrated in figure 4.

Either implicit or explicit enabling can be used to specify control in cases of cascaded delegation where the intermediary objects are *unaware* of secure delegation. If the intermediate is unaware, then the underlying security layer must effectively carry out either simple delegation or a special delegation mode set by an administrator. In SDM, explicitly specified modes are settable at the application level and may override the default mode set

by the administrator. Either way, delegation requirements become attached to an intermediate object's reference. This set of requirements is made available to any client holding a reference to this remote (intermediate object) reference.

In contrast, a delegation-aware intermediate might explicitly enable delegation for a method call. In SDM, this explicit delegation may be performed at the application level. If delegation is enabled, the client may generate a delegation certificate and pass it on to the intermediate object. Otherwise, no delegation certificate is generated and the intermediate provides service using only its privileges and none of the delegator's (in which case `NoDelegation` is the delegation mode).

An intermediate may also explicitly enable delegation using the `AccessController` methods `enableSimpleDelegation()` and `enableCascadedDelegation()`. The specified delegation mode is taken into account when privileges of the intermediate need to be presented to consecutive objects in the method invocation chain. Whether the intermediate's privileges are combined with the delegator's is based on the mode of delegation. The system can obtain the security requirements attached to any remote reference. The delegation, if required by the specified requirements (and the target object is thus willing to act as a delegate), is activated appropriately from the context. Using the context of invocation, a delegator's `AccessController` determines the `CodeExecutor` who is executing the client's code. This `CodeExecutor` becomes the `Signer` of a delegation certificate, and thus, effectively, the initiator of a delegation.

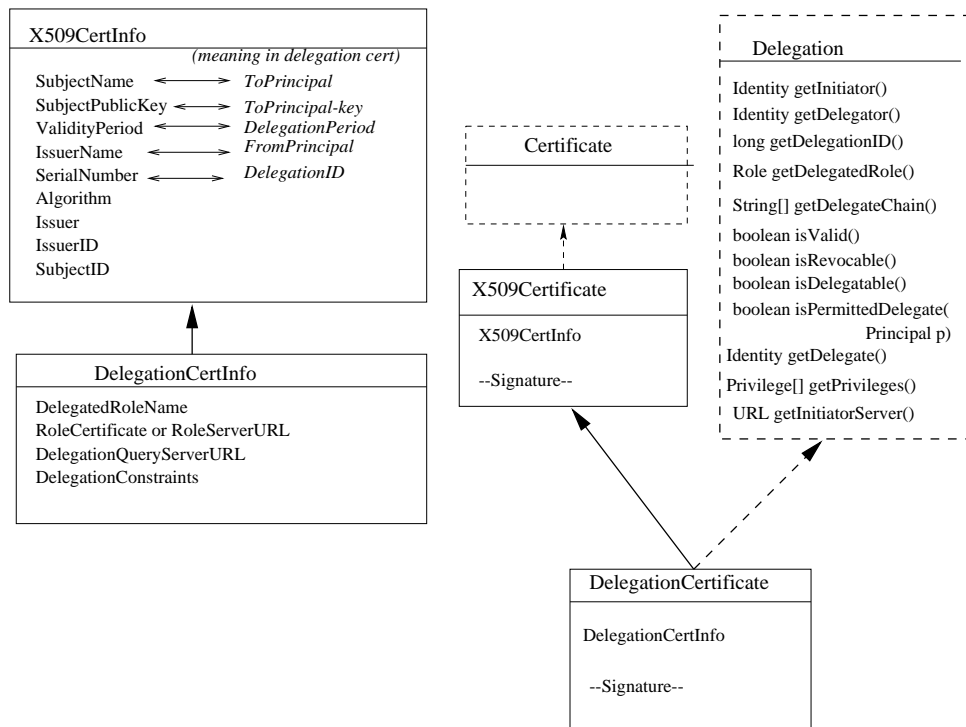
An example of application-level control is shown in the code segment in figure 5. This code could be used to handle situations in which a client object invokes method `obtainInsuranceInfo()` on a servlet object, `InsuranceDBServlet`. The `InsuranceDBServlet` object might in turn invoke methods on some enterprise service in the insurance company, say, an `insuranceServer` object. In the sample code, the `InsuranceDBServlet` explicitly enables delegation before further invocation on `insuranceServer`.

3.4. Delegation certificates

When an object decides to delegate a task to another object (effectively to the `CodeExecutor` of that object), it

```

public class InsuranceDBServlet {
    :
    // if the posted request is to obtain information, the servlet
    // invokes this method
    private void obtainInsuranceInfo(int insuranceId) {
        :
        // impersonate the requestor in making a request to insuranceServer
        AccessController.enableSimpleDelegation();
        insuranceServer.getInsuranceInfo(insuranceId);
        AccessController.disableDelegation();
        :
    }
}
    
```

Figure 5. Sample usage.

Figure 6. X.509 and delegation certificates.

creates a delegation certificate. This certificate specifies the initiator, role it is delegating, any constraints that are bound to the delegation, a nonce, validity period and its DelegationServer name for handling queries regarding delegation revocation. A role certificate is associated with the role being delegated, which might contain a set of privileges associated with it.

A delegation certificate is generated using the CodeExecutor as the FromPrincipal and the CodeExecutor of the remoteAdmin object as the ToPrincipal. Implementations could be based on public key cryptography using X.509 certificates, as illustrated in figure 6. The associated role (and hence, set of privileges) is specified in the certificate.

A delegation certificate is issued for every delegation session unless an earlier delegation has been set to remain valid for consecutive sessions. The type of the delegation

certificate (SimpleDelegationCert or CascadedDelegationCert) reflects the kind of delegation that is activated for this session. If the delegation is revocable, the endpoint makes sure that the delegation certificate is not revoked before it provides access.

Selection of consecutive delegates is made by an intermediate. The selected principal (CodeExecutor of the selected object for further delegation) is verified to be a permitted delegate by invoking the isPermittedDelegate(Principal) method on the certificate (DelegationCertificates must implement the Delegation interface shown in figure 6). This method will scan through the list of exempted delegates (if any) and accordingly will return a boolean value, indicating whether or not the principal is a valid delegate.

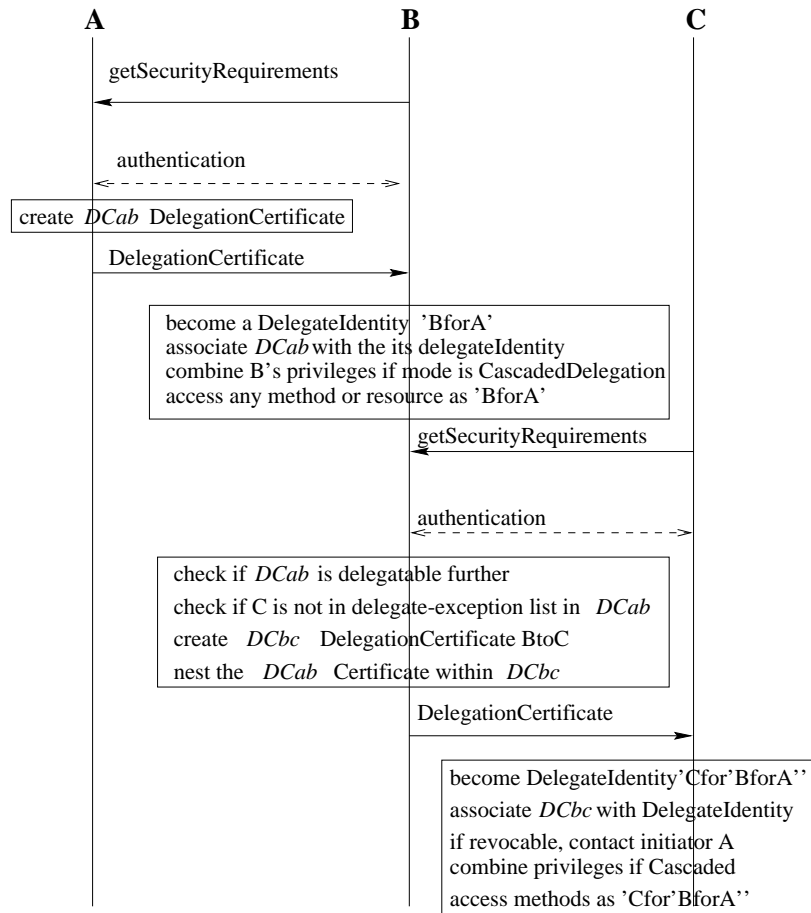


Figure 7. Main delegation protocol in SDM.

4. Delegation protocols

SDM employs a set of basic protocols that underly the usages described in section 3. SDM delegation protocols specify what information gets exchanged when an object *A* invokes a method on object *B*. The underlying layer must determine the delegation mode to be enabled from the context and security requirements attached to the target (remote reference *B*). Thus, the security policy for an intermediate object governs which privileges and delegation mode to apply at any given context (see figure 7).

Different rules apply for each of the combinations of required and specified modes that can occur in a sequence of invocations from object *A* to object *B* to *C* (i.e. $A \rightarrow B \rightarrow C$):

A does not enable Delegation, B specifies NoDelegation. Delegation is disabled for this session. No delegation certificates are generated. Methods on object *B* are invoked as if by object *A*, and methods invoked by object *B* on the next object in the delegation chain are invoked with object *B*'s privileges and so on. Any object that is invoked by *B* will not get any information that reflects that *A* has delegated to *B* to complete the task.

A does not enable Delegation, B specifies Simple or Cascaded Delegation. Delegation is disabled for this

session even though *B* requires it. When the operation that requires delegation from *A* to *B* is attempted, an exception is thrown and the operation is not carried out.

A enables Delegation, B requires NoDelegation. If the security requirements attached to *B* specify that delegation be not enabled for this session, then no delegation certificate is generated. The method on object *B* is invoked as if by object *A*, and the method invoked by object *B* on the next object in the delegation chain is invoked with object *B*'s privileges and so on.

A enables Delegation, B requires Delegation. If *B* requires delegation, *A* must generate a delegation certificate, DC_{ab} , to *B*. This delegation certificate is available to *B* for any further invocation. Consider when *B* needs to invoke a method on another object *C*. Such invocation on object *C* is carried out by *B* as a **DelegateIdentity (B for A)** (i.e. *B* is a delegate and *A* is the initiator).

A enables Delegation, B specifies Simple Delegation. Further invocations (and delegations, if any) made by *B* are made as **(B for A)** with only the privileges of *A* being used. In other words, *B* impersonates *A* (**B as A**). Any target that receives a request from *B* will authenticate *B* and obtain the delegation certificate DC_{ab} . Further control

will be based on privileges of A and B 's capacity to act as a delegate.

A enables Delegation, B specifies Cascaded Delegation. Further invocations and delegations are made by B by combining both the privileges of A (using delegation certificate DC_{ab}) and B (by providing necessary role certificates or identity certificates). In other words, B represents A by combining the privileges of both A and B . Any target receiving a request from B will base its access decision on A being an initiator and B being a delegate, with the combined privileges of both A and B .

4.1. Chained invocations

Once the intermediate B has obtained the delegation certificate DC_{ab} from A , it has the authority to speak for A . To complete the service, B might have to invoke methods on other objects. When B selects C to be the next target, B represents an entity (**B for A**) and requires access to method invocation. At this point B exercises the type `DelegateIdentity` and during the process of becoming a `DelegateIdentity`, the delegation mode is considered to calculate the privileges of the delegate (here, B). In this case of B being a `DelegateIdentity`, B can authenticate for itself. Also, it provides the delegation certificate DC_{ab} to prove that A has indeed delegated the task to B . C authenticates that it is actually B it is talking to, through normal authentication procedures. It verifies the delegation certificate to be signed by A by verifying the digital signature of A that is engraved in the certificate DC_{ab} . These provide proof of the fact that ' B speaks for A ', abbreviated as (**B for A**) [1]. The delegate's (B 's) `getPrivileges()` method returns the privileges associated with B , which is either only the privileges gained through delegation (A 's privileges only), or also includes the privileges of the delegate identity itself (both B 's privileges and A 's privileges). Thus the set of privileges returned by the delegate reflects the privilege of the identity (**B for A**) during that context.

Based on access control policies on the target C , the method invocation and any related resource accesses are controlled. These access control policies may be based on only the initiator (A) or might depend on the delegates as well.

If C requires delegation from its requester and B is a delegate for A possessing a delegate certificate DC_{ab} , i.e. (**B for A**), then:

- (i) B first checks if this delegation is forwardable, i.e. delegatable to further intermediaries.
- (ii) If it is forwardable, B checks whether C is present in an exception list provided with the delegation certificate DC_{ab} (i.e. whether A disapproves any further delegation to C).
- (iii) If delegation to C is not prohibited, B generates a delegation certificate DC_{bc} and passes on DC_{ab} with it to C . The delegation certificate DC_{bc} may contain: (a) A 's privileges only, (b) B 's privileges only or (c) a combination of both, depending on the delegation mode specified in the delegate identity B . Once B delegates

to C , the delegation chain becomes $A \rightarrow B \rightarrow C$, i.e. (**C for (B for A)**).

In contrast, if B is not a delegate for A , that is if B had not specified delegation in its security requirements, then A would not have generated (and passed on) the delegation certificate DC_{ab} to B . In this case, when a request is issued to C , it is not possible for B to establish A as the original initiator due to the lack of a delegation certificate. So C must treat it as if the request originated from B and handle it accordingly, without having any idea about the involvement of A in the complete invocation chain. Extending this chain to one more principal, we get $A \rightarrow B \rightarrow C \rightarrow D$. If B does not require delegation and C does, then when the request reaches D , D will treat the request as having initiated from B and delegated through C .

Thus, at any given time, control is based only on currently available information on the delegation chain and the specified modes and policies. SDM does not support any means of tracing back calls through intermediaries to obtain predecessor delegation certificates.

4.2. An example

Consider an example of a requestor (insurance agent) using the services of a `InsuranceDBServlet` object to obtain insurance information for a customer. This is illustrated in figure 5. Let a servlet object provide services related to insurance information for a company, using other enterprise services. It might in turn need to make use of the services of `InsuranceServer`. The requestor obtains the reference of the servlet and invokes the `obtainInsuranceInfo` method on it by passing the id of the customer for whom the information is obtained. The servlet might specify, attached with its object reference, a set of security requirements. Let the security requirements specify that *Delegation* is required. In SDM, our system will analyse this security requirement attached to an intermediate object (in this case, the servlet) and whether the requestor is willing to delegate (known from requestor's security specification attached to its object reference). The underlying system generates a delegation certificate and passes it on to the servlet.

Let the servlet contact the `InsuranceServer` object to obtain the insurance information by invoking the `getInsuranceInfo` method. (We assume that the servlet has already authenticated and established connection with the `InsuranceServer`.) The servlet provides the delegation certificate issued by the requestor. The `InsuranceDBServlet` object acts as a delegate, acting on behalf of the requestor (impersonating the requestor), and makes a request to the `InsuranceServer`. For this request, the servlet uses the privileges of the initiator (as the insurance server might make use of requestor's privilege of being an insurance agent). Thus the servlet makes use of the `SimpleDelegation` facility provided by SDM while invoking the `getInsuranceInfo` method on the `InsuranceServer` object.

5. Revocation

Sometimes users and services need to *revoke* privilege assignments. Users change their minds, people leave

groups, services change functionality, and so on. Even though it adds complexity, any practical delegation protocol must support revocation.

In SDM, revocability is an *optional* attribute of delegation. If performance is an issue, or revocation is somehow known to never be necessary, the delegation can be made non-revocable. This facility to explicitly enable or disable revocation is again carried out using the AccessController object. The changed revocation status remains valid, until it is changed again. The AccessController method `setRevocableDelegation(true)` enables delegation to be revocable until it is set otherwise.

If delegation is revocable, then the endpoint (but not necessarily any of the intermediate delegates) of a chain must be able to find out. In SDM, the DelegationID and delegation server (URL) associated with certificates define the uniqueness of a delegation certificate. If the endpoint has not seen the delegation certificate earlier, it must contact the DelegationServer of the initiator and verify its validity. And if it is not a one-shot delegation (a delegation that is valid for one access request only), the endpoint registers itself as a DelegationRevocationListener with the initiator.

When an endpoint receives a service request from a principal, its AccessController checks if the service has been delegated through the invoking principal and, if so, whether the delegation is revocable. If the delegation is not revocable, it goes ahead to provide/deny access according to the delegate privileges.

But if the delegation is revocable:

- The AccessController first checks if the delegation certificate is in a local `<delegationCertificate, status>` table.
- If the certificate is not present in the table, then this must be the first time this delegation certificate has been obtained. It contacts the DelegationServer of the initiator querying the status of the delegation.
- If the delegation is not one-shot, the user setting is analysed to see if the change-of-status notification is *periodic* or *aperiodic*.
- In the default aperiodic case, the endpoint registers itself as a DelegationStatusListener with the delegation server for future updates only upon status changes.
- In the periodic case, the endpoint registers itself as a DelegationStatusListener with the delegation server for future periodic updates at a given update interval.
- If the delegation is revoked (i.e. the delegation is no longer valid), the access is denied. Otherwise, the AccessController then provides or denies access according to resulting status and delegate privileges.

5.1. Revocation notifications

In SDM, revocation is possible even when the initiator does not know the endpoint *a priori*. When an endpoint (final target) receives a revocable delegation request, it registers with the initiator as being interested in receiving revocation notifications. Thus each of such endpoints register themselves as DelegationStatusListeners to the initiator. The initiator in turn maintains a list of endpoints to whom its delegation has propagated. These endpoints

will implement the NotificationHandler interface to handle any event notification.

If the endpoint contacts the initiator every time before servicing a delegated request, then the endpoint is considered to follow a *pure pull* mechanism to obtain the status information from the initiator. A common alternative is *pure push* mechanisms, in which the initiator continually broadcasts out revocation information. Analyses of similar protocols using Broadcast Disks [3] show that pure pull provides extremely fast response time for a lightly loaded server, but as the server becomes loaded, its performance degrades, until it ultimately stabilizes. The performance of pure push is independent of the number of clients listening to the broadcast. But if the number of interested clients (endpoints) is large, then it is a waste of resources to send irrelevant data. A more serious problem is that the servers might not deliver the specific data needed by clients in a timely fashion. One solution suggested by Zdonik [3] is to allow the clients to provide a profile of their interests to the servers.

In SDM, the clients are the endpoints who are interested in the revocation status of certain delegations (serviced by those endpoints). When an endpoint receives a delegated request, the first time around it pulls information from the initiator about revocation status and at the same time registers itself to receive delegation-related events. The profiles of those endpoints of interest in SDM are the details on whether they require periodic push or aperiodic push. An *aperiodic push* is event driven—a data transmission is triggered by an event such as data update (in SDM, it is a change delegation status). As a result, endpoints (and hence, the NotificationHandlers) are notified of any change in delegation or its privileges by the initiator (which might use a helper object that implements EventGenerator interface). A *periodic push* is performed according to some pre-arranged schedule. The endpoint, when it registers itself with the initiator, will specify the time interval of periodic updates (pushes). Hence, the initiator will push delegation details at specified time intervals to the registered endpoints. This leaves it to the endpoint to specify whether it needs aperiodic or periodic (if so, the necessary time interval) pushes. Thus the endpoint need not pull information after its initial ‘pull’ as the initiator will ‘push’ (revocation) data to registered listeners (endpoints). Either periodic or aperiodic, this *pull-once-push-many* approach supports revocation where an endpoint receives revocation notifications from an initiator.

An endpoint will decide to specify its interest in periodic or aperiodic pushes from the initiator based on how critical the revocation affects its service and its resources. For example, if the endpoint is a TelephoneDirectory service then providing information to a requesting delegate (a secretary object) about a revoked number is not very crucial, as the delegate might not misuse the telephone number. In this case, periodic pushes from the service department are not necessary and the endpoint might settle for aperiodic pushes only. On the other hand, if the requested service is providing classified information, then the endpoint needs to know the revocation of the delegate (for example, a secretary object) immediately. In this case,

short-interval periodic pushes from the service department may be selected. The load on the server to keep pushing revocation status of its delegates becomes worth its cost when compared to the risk involved in providing classified information to any revoked delegate.

6. Status and future work

This paper has focused on the way in which delegation is structured and used in SDM to support secure operation when multiple components together provide a given service. SDM builds upon existing mechanisms, mainly those already established in the Java JDK1.2 security framework, to establish a practical basis for constructing flexible yet secure components and support infrastructure. SDM extends the JDK1.2 framework to include explicit support for principals. We have provided an implementation strategy for SDM to be built over the JDK1.2 framework.

As outlined in section 2.2, implementation of SDM requires that the JDK1.2 domain model be extended to include principals, so that each `CodeSource` will also have a principal associated with it. One domain will be formed for each such `<CodeExecutor, CodeSource>`. Further authentication and access control (and delegation) may then be based on the `CodeExecutor`.

To support `PrincipalDomains`, the Java runtime system must maintain a mapping from the `<CodeSource, CodeExecutor>` pair to their protection domains and also the mapping between protection domains and their privileges. This could, for example, be implemented at the execution stack level with the aid of class blocks and the executing environment frame, as illustrated in figure 2.

In the future, we intend to implement our SDM delegation framework over the JDK1.2 security framework. We have already implemented access control mechanisms [17] based on `CodeSource` information. We plan to extend the mechanism to include the information on principals to further control any access requests.

7. Discussion

SDM provides a realistic security framework for Java-based distributed object systems. It isolates the complexities of the underlying protocols necessary to provide a very wide range of security policies and trust levels. It presents application writers and system administrators with a flexible, uniform API. SDM appears to be the most conservative extension of the Java 1.2 security architecture that simultaneously supports both delegation- and role-based security, along with revocation mechanisms that are often needed in practice.

7.1. Comparison with other work

The design of SDM has also benefited from other work in security architectures, but differs from previous systems in significant ways.

DSSA. Roles are not explicit in *DSSA* [4] and are achieved through their notion of groups, whereas explicit

support for roles is provided in SDM. *DSSA* supports only combined delegation, whereas SDM supports both combined delegation and composite delegation.

Varadharajan et al. The main revocation strategy proposed by *Varadharajan et al* [15] propagates revocations through delegates. These revocations might not take effect due to network problems or other distributed failures. Another solution proposed in [15] assumes a previously known endpoint. This is also supported in SDM. Approaches suggested in their paper require changing the key associated with a principal. This is not effective in public key systems, which are generally more manageable and scalable in distributed systems (and are supported in SDM). They also suggest passing a *read* capability of the delegation token and not the token itself. Our approach is vaguely similar in that the endpoint needs to contact the initiator before servicing. But by using the *pull-once-push-many* approach, SDM does not need to contact its initiator because the initiator will multicast revocation details, if needed.

SESAME. SDM provides both simple and cascaded (composite, combined) delegation with support for constraints whereas *SESAME* [9] supports only simple delegation. Also, unlike *SESAME*, SDM also supports scalable distributed naming schemes.

Kerberos. In *Kerberos* [14], the endpoint contacts the authentication server for every signature authentication as it uses a shared key approach. SDM allows implementation via public keys and hence need not contact an authentication server every time. *Kerberos* does not support roles. Principals can restrict their privileges before delegation. Also, *Kerberos* does not support cascaded delegation. There is no mechanism mentioned for revocation.

Taos. *Taos* [16] has no mechanism for revocation implemented. It supports the notion of a Privileges Server. Every time an access is processed by the endpoint, it contacts the Privileges Server to validate the certificate.

DCE. *DCE* [2] does not provide any facility for revocation. Also, *DCE* uses shared key authentication which is not as scalable in distributed environments.

7.2. Limitations

We are aware of the following limitations of SDM, that reflect some of the engineering trade-offs encountered in its design:

- SDM relies on initiators to enable delegation. If they do not, delegation will never be enabled and hence no delegation certificates will be generated. If delegation is not initially enabled, at a later stage during method execution (through delegation), a target object cannot determine the original initiator of the request. The only way to find out the original initiator would be to use a call back trace mechanism, which is not supported in SDM.

- SDM does not support any means to check whether a principal adopts mutually disjoint roles. SDM cannot ensure that roles adopted by a principal do not conflict (for example simultaneously requiring and prohibiting rights).
- Although the pull-once-push-many approach is an efficient approach, event notification does not carry any real-time guarantees due to possible network latency. Before a revocation event notification is delivered to a listener, the listener might have already allowed the revoked delegation. Hence, the event notification across distributed systems in SDM is not atomic.

References

- [1] Abadi M *et al* 1993 A calculus for access control in distributed systems *ACM Trans. Programming Lang. Syst.* 706–34
- [2] Erdos M and Pato J 1993 Extending the OSF DCE authorization system to support practical delegation *PSRG Workshop on Network and Distributed System Security*
- [3] Franklin M and Zdonik S 1997 A framework for scalable dissemination-based systems *Proc. OOPSLA '97*
- [4] Gasser M and McDermott E 1990 An architecture for practical delegation in a distributed system *Proc. 1990 IEEE Symp. on Security and Privacy* (Los Alamitos, CA: IEEE Computer Society Press) pp 20–30
- [5] Gong L Java Security Architecture (JDK1.2) *Draft Document* Revision 0.5, available from Java Developer Connection <http://java.sun.com>
- [6] Nagaratnam N and Lea D 1998 Secure delegation for distributed object environments *Proc. USENIX Conf. on Object Oriented Technologies and Systems '98*
- [7] Nagaratnam N 1997 Practical delegation for secure distributed object environments *PhD Dissertation* Syracuse University
- [8] Object Management Group *CORBA: Security Service Specification*
- [9] Parker T and Pinkas D 1995 *Sesame V4—Overview* Issue 1
- [10] Sandhu R *et al* Role-based access control models *IEEE Comput.*
- [11] Sandhu R *et al* 1994 Role-based access control *Proc 10th Ann. Computer Security Applications Conf.* pp 54–62
- [12] Sandhu R and Samarati P 1994 Access control: principles and practice *IEEE Comput.* 40–8
- [13] Sollins K 1988 Cascaded authentication *Proc. IEEE Symp. on Security and Privacy* (Los Alamitos, CA: IEEE Computer Society Press) pp 156–63
- [14] Steiner J *et al* 1988 Kerberos: an authentication service for open network systems *Proc. Usenix Winter Conf.* pp 191–202
- [15] Varadharajan *et al* 1991 An analysis of the proxy problem in distributed systems *Proc. IEEE Symp. on Security and Privacy* (Los Alamitos, CA: IEEE Computer Society Press) pp 255–75
- [16] Wobber, Abadi, Burrows and Lampson 1994 Authentication in Taos operating system *ACM Trans. Comput. Syst.* **12** (1)
- [17] Nagaratnam N and Byrne S B 1997 Resource access control for an Internet user agent *Proc. USENIX Conf. on Object Oriented Technologies and Systems '97*
- [18] Wollrath A, Riggs R and Waldo J 1996 A distributed object model for the Java system *Proc. USENIX Conf. on Object-Oriented Technology and Systems '96*