

ACCESS CONTROL, REVERSE ACCESS CONTROL AND REPLICATION CONTROL IN A WORLD WIDE DISTRIBUTED SYSTEM

Bogdan C. Popescu
Vrije Universiteit, Amsterdam
bpopescu@cs.vu.nl

Lt. Col. Dr. Chandana Gamage
Sri Lanka Army Headquarters, Colombo, Sri Lanka *
chandag@cse.mrt.ac.lk

Andrew S. Tanenbaum
Vrije Universiteit, Amsterdam
ast@cs.vu.nl

Abstract

In this paper we examine several access control problems that occur in an object-based distributed system that permits objects to be replicated on multiple machines. First, there is the classical access control problem, which relates to which users can execute which methods. Second, we identified a reverse access control problem, which concerns which replicas can execute which methods for authorized users. Finally, there is the issue of how updates are propagated securely from replica to replica. Our solution uses roles and preserves the scalability needed in a world-wide distributed system.

Keywords: Distributed Systems, Replicated Objects, Security, Access Control, Digital Certificates.

*Work completed while at Vrije Universiteit

1. Security in Distributed Systems

Security in distributed systems differs from operating system security by the fact that there is no central, trusted authority that mediates interaction between users and processes. Instead, a distributed system usually runs on top of a large number of loosely coupled autonomous hosts. Those hosts may run different operating systems, and may have different security policies, which can be enforced in different ways by careless, or even malicious administrators.

The popular trend in distributed systems is to encapsulate functionality as objects and provide mechanisms for their location, migration, persistence, as well as for remote method invocation. CORBA [1] [2], DCOM [5], and Legion [6] are examples of distributed systems using this paradigm. Each of them handles security in its own way, and the main objectives are authenticating the communicating parties, protecting network traffic, enforcing access control policies on the object's member functions, delegating rights and respecting site-specific security concerns. There is one feature these systems have in common: all of them support only nonreplicated objects. This makes it easy to implement access control mechanisms for individual objects, since such mechanisms would have to be enforced in only one point, namely at the host where the object resides.

Globe [15], is a wide-area distributed system based on *distributed shared objects* (DSO). The notion of a DSO stresses the property that objects in Globe are not only shared by multiple users, but also physically replicated at possibly thousands of hosts [4].

This paper addresses the security issues arising from the physical replication of objects. In particular, we focus on how access control policies can be implemented in a system where those policies need to be enforced at a number (possibly very large) of distinct locations, with various degrees of trustworthiness.

The security challenges posed by Globe come from the fact that millions of users can invoke methods on any of the possibly thousands of replicas of a highly distributed object. It makes sense to assume that in such an object, there is a trust hierarchy (some of the replicas run on hosts that are more trusted than others), and this translates into what kind of actions the various replicas should be allowed to execute. In nonreplicated systems, the **access control problem** is how to prevent unauthorized users from invoking methods for which they have no rights. With replicated objects, we also need to prevent legitimate users from sending security critical requests to replicas that are not trusted enough to execute them (we call this the **reverse access control problem**). Finally, the **replication control problem** deals with enforcing a security policy on the way replicas exchange state update messages in order to keep the DSO's state consistent.

The rest of the paper is organized as follows: Section 2 gives an overview of the Globe system, the internal structure of a DSO, and how replicas interact to implement the DSO functionality. The following three sections deal with the access control problem, reverse access control problem and replication control problem respectively. Section 6 gives a quick outline on how one would implement the solutions we proposed for these problems, and Section 7 gives an application example. Finally, Section 8 concludes and outlines areas of future research.

2. The Globe System

A central construct in the Globe architecture is the distributed shared object (DSO). A DSO is built from a number of **local objects** that reside in a single address space and communicate with local objects in other address spaces.

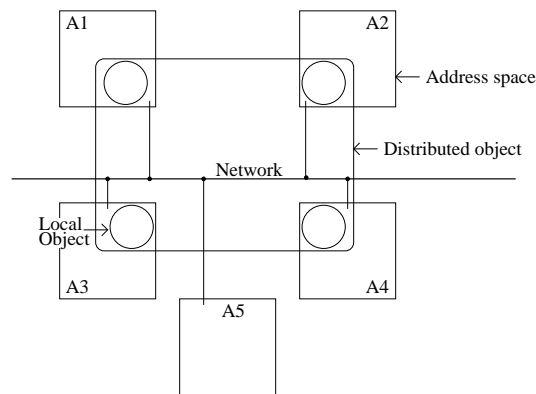


Figure 1. A Globe DSO replicated across four address spaces

Some of the local objects (possibly all of them, depending on the replication strategy) can store all or part of the DSO's state. A local object that stores some part of the DSO's state is called a **replica**.

All the replicas that are part of a DSO work together to implement the functionality of that DSO. Replicas consist of the code for the application, the state they store, and the distribution mechanism. The internal structure of a local object is shown in Figure (2). The distribution mechanism enables transparent distribution and replication of objects, hiding these details from application developers.

The semantics subobject contains the code that implements the functionality of the DSO. This is the only subobject that needs to be written by the application developer.

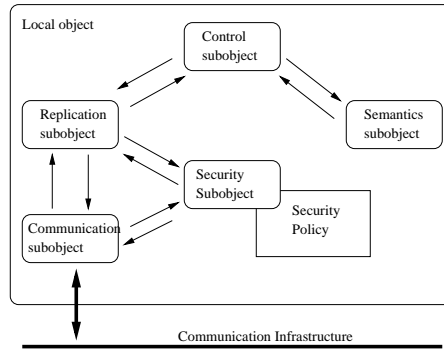


Figure 2. The internal structure of a local object. The arrows indicate the possible interactions between the subobjects

The communication subobject is responsible for the communication between local objects residing in different address spaces. It hides the network communication aspects from all the other subobjects.

The replication subobject is responsible for keeping the state of the local object consistent according to the per-DSO replication strategy. This is accomplished by exchanging state update messages with the replication subobjects of the other replicas that make up the DSO. The replication subobject is also responsible with providing the user with the view of a logical non-replicated object. This is accomplished by transforming method invocations that cannot be handled locally into remote requests, and sending those requests to the replicas that can handle them.

The control subobject is in general system-provided. Its job is to take care of invocations from client processes on the host where the local representative resides and to mediate the interaction between the semantics subobject and the replication subobject.

The security subobject [8] is responsible for enforcing the object's security policy at the local representative level. This is done by having the security subobject mediating the communication flow between the other local subobjects. Prior to passing a protocol message to a lower or upper level in the subobject hierarchy, a subobject passes it to the security subobject. The security subobject can apply security measures to protocol messages (encrypting data before sending it to the network, for example) or can even prohibit the continuation of the protocol if that would be against the DSO's security policy (for example when an un-authorized user tries a remote method invocation on a replica).

In the secure version of Globe, all actions (creating a DSO, running a DSO replica, invoking a method ...) are done by **principals**. A principal can be any entity (human user, group of users, institution ...) that has a public key and a

digital certificate certifying that key. We assume the existence of a Public Key Infrastructure used for the distribution and management of these certificates.

Each Globe DSO must have an **owner**, who is the principal that is in charge of the administration and security policy for that object. The **object's public key** is the public key of the object's owner.

Any DSO replica must be run by some principal. The **replica's public key** is the public key of the principal that runs that replica. The replica's principal may or may not be the same as the object owner.

For a DSO, any principal that calls the object's methods is a **user** for that object. Such a user must have a local representative of the object installed in its address space.

It is important to understand that users and replicas are orthogonal concepts in our architecture. Replicas are the building blocks of the DSO. They store parts of the DSO's state and interact according to a replication protocol in order to implement the DSO's functionality. Users are external world entities that invoke the DSO's methods. They see the DSO as a single logical object through the DSO's local representative installed in their address space (this representative could be a replica, if any part of the DSO's state is stored in it, but that is dependent on the replication policy).

3. The Access Control Problem

As stated before, for a given application, the access control problem is how to restrict an user to execute only those operations allowed under the application's security policy with respect to that user. For Globe, access control policies are defined on a per-DSO basis, so each object can have its own security policy.

The access control problem has been solved in different ways for the various existing distributed systems. One approach is to associate resources with Access Control Lists (ACLs) [10]. Such a list would simply enumerate all the individuals allowed to use that resource. If that resource is an object, this can be fine-grained by specifying which methods are accessible to each individual. Another possible approach is to use capabilities. A capability is a protected identifier that both identifies an object and specifies access rights to be allowed to the accessor that possesses the capability [3].

ACLs are not suited for implementing access control in systems with worldwide replication of objects. For a heavily replicated DSO, the ACL would have to be stored by each of its thousands of replicas. Even in a distributed system where objects are not replicated, the size of an ACL is proportional to the number of users. In a system like Globe, the storage needed for the ACL would also be proportional to the number of replicas for that object. And what makes the situation even worse is the need to have strong consistency among all the ACLs stored by the various replicas worldwide.

Capabilities are also unsuited for implementing access control in a system like Globe because they suffer from the so-called confinement problem: it is hard to prevent a capability from being passed from one user to another without the object's approval. This problem can be solved in centralized systems by not allowing users to directly manipulate their capabilities. Instead, they are stored by the (trusted) underlying system and presented to objects on users' behalf. Since there is no underlying trusted system in Globe, this technique cannot be applied.

Role Based Access Control (RBAC) [11] represents another approach to the access control problem, and has been the focus of intense research in the past few years [12], [9]. The main idea behind RBAC is that permissions are associated with roles instead of users. Roles are abstractions that group entities with equivalent security properties for an application.

In Globe, we identify a role as a subset of all the methods offered by a DSO. If there are N methods, in theory there are $2^N - 1$ potential roles, although most of these are not likely to be useful. This two-level scheme (users to roles and roles to permissions) greatly simplifies the management of access control lists. First of all, we expect many fewer roles than users (since roles group users with the same rights). This will result in much more compact access control lists. Second, remember that one major problem with ACLs in a worldwide system with replicated objects is keeping them consistent, since users can be granted or revoked rights at any time. With RBAC, this is done by assigning that user in a new role without modifying the ACL. In this way, ACLs can remain largely static while user roles are dynamically managed.

A legitimate question to raise is how does a DSO keep track of which roles have been assigned to which users? If this is kept as a list, we encounter the same problems as for ACL's, namely how to make such a list available to all replicas, while keeping it consistent. The solution is to use "role certificates". These certificates bind the user's public key with the role it has been assigned, and are signed by the object's owner.

When designing a DSO security policy, the object's owner first has to identify all the meaningful roles for that object. This is accomplished by careful examination of the application being implemented; some roles are needed to represent the various types of clients (for example a banking application may need to differentiate between regular members, gold card members and platinum card members). Other roles are needed to model the administrative hierarchy for the application - using the same banking example - we may need teller roles that are allowed to process withdrawals/deposits and manager roles who are allowed to approve loans. At the end, all the selected roles form the **user role set** for that object.

Now, we can implement the access control scheme for a given DSO using the user role set we identified for that DSO: with each user role we associate a

bit vector encoding the methods that role is allowed to invoke (for example if a user role is allowed to invoke methods M_2 and M_5 , its vector will be 01001...). Grouping the vectors for all the user roles for a DSO, we obtain the **access control matrix** for that DSO, which is stored in the security subobject. In this matrix, the first row contains the bit vector for the first role, the second row contains the bit vector for the second role, and so on. The DSO's replicas get this matrix from the object's owner when they are created. When a user invokes a method on a given replica, the security subobject on that replica first checks the access control matrix to make sure the user's role is allowed to call that method.

4. The Reverse Access Control Problem

The reverse access control problem is how to prevent legitimate users from sending service requests to applications not entitled to provide those services. Those malicious servers can fool legitimate users into believing that they have performed some action on behalf of them, when in fact they are not even allowed to perform it under the application's security policy

For distributed systems where objects are not physically replicated, the reverse access control problem is reduced to whether or not one trusts an object to perform a given action. This can be mediated by security policies based on the location of the object, or on the entity that owns it, and so on. The situation dramatically changes with worldwide replication: an object can be replicated on thousands of different systems. It makes sense to assume that only few of these replicas should be trusted to perform the most security sensitive operations provided by the object (like changing the object's state). The rest of the replicas are probably there for performance reasons, acting as caches for example.

What we need is a systematic and scalable way of describing which replicas are allowed to execute which methods for a given DSO. This can be tackled by examining that DSO and answering the question "why does this object need to be distributed?" One answer is that the functionality the object implements is suited to be divided among many parts – the replicas – spread across the network. When running, each of these replicas will implement part of what the DSO is supposed to do; we could divide the set of all replicas of a DSO into disjoint subsets with equivalent functionality; such replicas will have the same role in implementing what the DSO is supposed to do. We can see that the disjoint subsets we described, closely follow the replication strategy used for the object. For example, a master-slave replication strategy would create two such sets - the masters and the slaves. We name such a disjoint subset a **replication role**. All the replication roles identified for a given DSO form the **replication role set** for that DSO.

We now claim that if one replica in a given replication role is allowed to execute a client request for a method M of the object, then all the other replicas in that role should also be allowed to execute that request. Remember that a replication role is the set of all replicas with equivalent functionality. If some replicas in the role are allowed to execute method M , while others are not, then those replicas would differ in functionality (some are allowed to execute M and some not), and this contradicts the assumption they are in the same role.

Now, we can implement the reverse access control scheme for a given DSO using the replication role set we identified for that DSO: with each replication role we associate a bit vector encoding the client method requests that role is allowed to execute (for example if a replication role is allowed to serve methods M_2 and M_5 , its vector will be 01001...). Grouping the vectors for all the replication roles for a DSO, we obtain the **reverse access control matrix** for that DSO. The DSO's users get this matrix from the object's owner when they set up their local representative, and store it in the security subobject of that representative. When a user wants to invoke one of the object's methods, its security subobject will have to search in the matrix to find a role allowed to execute it.

Finally, a replica needs a **role certificate** to prove it has been assigned a certain role. Such certificates are given to replicas by the object's owner. They bind the public key of the principal running the replica to the replication role assigned to that replica and are signed with the owner's private key.

5. The Replication Control Problem

The replication control problem is how to determine which replicas are allowed to propagate state updates and to which replicas state updates should be propagated. In fact, this is access control and reverse access control on a special method - *stateUpdate*. This problem is further complicated by the fact that the various elements of the DSO's state may have different security properties. This refines the granularity of the access control decision which is now also based on the parameters of the *stateUpdate* request - the state change being propagated.

The problem is simplified by the fact that only the replicas of the DSO can exchange *stateUpdate* messages. The DSO's users have no direct access to its state; they can manipulate it only through method requests.

We need to stress an important point here: the problem we are trying to solve is not when a replica should start sending state update messages, or how these state updates should be constructed. That is outside the scope of the security architecture, and is determined by the replication algorithm being used. With replication control, we simply set some data-flow policies that the replication subobject must follow. A replica can send state updates once every second or

once every hour, and those state update messages can incorporate every small write or batch a number of such writes, those are all details dependent on the replication protocol. What we want to ensure is that a replica will not send state updates to other replicas that are not trusted enough to even see that part of the object's state in the update. We also want to ensure that a replica will not accept a state update created by another replica which is not trusted enough to modify the part of the object's state in the update. For example, we may want only the replica storing the master copy of the DSO's state to be able to send state updates to the caches. Caches then should be prevented from updating the state of the master replica or of the other caches.

We claim that the replication role set we introduced for the reverse access control problem is also relevant for the replication control problem: consider two replicas in the same role - if they wouldn't have the same rights in sending and receiving *stateUpdate* messages, then they would have different functionality, hence they couldn't be in the same role. Therefore, all replicas with the same role must have the same set of sending and receiving rights for stateUpdate requests.

Now we need to accommodate the requirement that *stateUpdate* exchanges can have different security requirements based on which elements of the DSO's state they modify. This can be done by dividing the DSO's state into disjoint subsets of elements with the same security sensitivity. Those subsets are called **partitions**. State updates for elements in the same partition would then be propagated in the same way through the DSO.

Using these constructs, we can now implement the replication control scheme: for each replication role, we need to specify for which partitions it is allowed to generate any *stateUpdate* messages, and to which roles it can send those messages. This information can be organized in a vector, with one entry for each partition in the DSO's state. Each entry would store the roles to which *stateUpdate* messages for that partition can be sent. Combining the vectors for all the replication roles of a DSO, we obtain the **replication control matrix** for that DSO. This matrix is again stored in the security subobject of each replica.

6. Putting the Pieces Together

The **access control matrix**, **reverse access control matrix** and **replication control matrix** fully describe what interactions are allowed between the different replicas and users of a DSO, according to the security policy set by the object owner. These structures are kept in the security subobject, where they are initialized when local objects are created. In order to ensure that the security policy enforced by a local object is the one set by the object owner, these data structures will be digitally signed by the owner with its private key, and sent to

the principal creating the new local object together with the user role certificate (and possibly the replication role certificate, if the principal creates a replica).

Replicas and users can interact only after they have authenticated each other. The first step is for the entities to exchange their user/replication role certificates. A user/replica then authenticates itself by proving it has access to the private key corresponding to the public key in its user/replication role certificate (for example by signing a random nonce sent by the other party with its private key).

Once two entities have authenticated each other, they can set up a secure communication channel using some Diffie-Hellman or RSA public key cryptography technique [7]. Secure channels are managed by the security subobject, and provide at least data integrity, authenticity, and possibly confidentiality. The secure channel (identified by a channel ID) is the only thing the security subobject makes visible to the other subobjects. All the details regarding the role assigned to the replica/user at the other end of the channel, keys and cryptographic algorithms used to protect the data on the channel, are managed by the security subobject only, and hidden from the rest of the application.

Before any subobjects part of a local object can perform any actions, they need to check with the security subobject that these actions do not violate the security policies set by the object owner. When a user wants to invoke a method M , it needs to make a call $findReplica(M)$ to its security subobject. This will check the reverse access control matrix, find the replication role allowed to execute M , find a replica in that role (this is done with the help of the Globe Location Service [14], but the details are outside the scope of this paper), establish a secure channel with that replica and return the channel ID for that channel. Finally, the method request is sent on that channel.

A similar scenario happens when a replica receives a request to invoke a method M from one of the secure channels it has established with the users. The replication subobject needs to call $allowAccess(channelID, M)$ to the security subobject. The security subobject uses the $channelID$ to retrieve the role of the user at the other end. It then checks the access control matrix to see if that user role is allowed to invoke M , and returns $true/false$ according to the entry in the matrix.

Finally, replication control decisions are needed when replicas receive *state Update* messages. The first step is to identify the partition to which the state update is targeted. Following that, the replication object needs to call $allowUpdate(channelID, partition)$ to the security subobject, where $channelID$ is the id of the secure channel where the update comes from. The security subobject checks the $channelID$ to retrieve the replication role of the replica at the other end. It then checks the replication control matrix to see if that replication role is allowed to update that partition, and returns $true/false$ according to the entry in the matrix.

7. An Example

In this section we'll show an example on how the security scheme we described in the previous sections can be used to construct a secure Globe application.

Consider a DSO modeling an electronic newspaper: this contains articles and on-line advertising. Those are stored on a set of core replicas. However, we want separation between core replicas dealing with articles, and those dealing with advertising, since the later could be managed by a third party (DoubleClick for example). Newspaper content is pushed by the core group toward a much larger group of caches, which in turn provide this content to the newspaper's readers. Readers fall into two roles: (1) registered users that can only read the headlines, and (2) subscribers who are be allowed to read full articles.

We can model such an application with a DSO that has the following methods: *add_news()*, *add_advert()*, *read_headln()*, *read_article()*. We identify the user roles for this application as the *Editor* (manages articles), *Advertising Manager* (deals with advertising content), *Registered User* and *Subscriber*. *Editors* should not be allowed to add advertising; *Advertising Managers* should not be allowed to add articles; *Registered Users* and *Subscribers* should not be allowed to add anything. Furthermore, *Registered Users* should only be allowed to call *read_headln()*. Figure (3) shows the access control matrix for the object.

		Methods			
		add_news	add_advert.	read_headln	read_article
User Roles	Editor	True	False	True	True
	Advertising Mngr	False	True	True	True
	Registr. User	False	False	True	False
	Subscriber	False	False	True	True

Figure 3. The Access Control Matrix for the E-Newspaper DSO

As for reverse access control, we identify the replication role set to have three elements - Core Articles Stores, Core Advertising Stores and Caches. Only Caches should be allowed to serve the *read* requests. Only Core Articles Stores should be allowed to execute *add_news()* requests. Only Core Advertising Stores should be allowed to execute *add_advert()* requests. Figure (4) shows the reverse access control matrix for the object.

Finally, for replication control, we separate the DSO's state into two partitions, one for article content (P_0), and the other for advertising content (P_1). Only Core Articles Stores can generate *stateUpdate* messages for P_0 , and those messages should be sent only to other Core Articles Stores (active replication) and Caches. Only Core Advertising Stores can generate *stateUpdate* messages for P_1 , and those messages should be sent only to other Core Adver-

Replication Roles	Methods			
	add_news	add_advert.	read_headln	read_article
Articles Store	True	False	False	False
Advertising. Store	False	True	False	False
Cache	False	False	True	True

Figure 4. The Reverse Access Control Matrix for the E-Newspaper DSO

tising Stores (active replication) and Caches. Figure (5) shows the replication control matrix for the object.

Replication Roles	Partitions	
	Articles Partition	Advertising Partition
Articles Store - ArtS	ArtS, Ch	Not Allowed
Advertising. Store - AdvS	Not Allowed	AdvS, Ch
Cache - Ch	Not Allowed	Not Allowed

Figure 5. The Replication Control Matrix for the E-Newspaper DSO

8. Conclusions and Future Work

In this paper we describe three access control problems we encounter when designing a security architecture for the Globe system. The classic access control problem is extremely general and is found in distributed systems, operating systems and databases security. Reverse access control is common to systems where there is a large number of servers, and users need ways of identifying which of them are the legitimate providers of a given service. Finally, replication control is the problem of adding a security policy to a state consistency protocol.

The general techniques we use to tackle these problems is to organize entities (users, replicas, state elements) into sets with equivalent security properties and to design security policies based on these equivalence sets. This approach is influenced by previous work done on Role Based Access Control (RBAC).

As for future work, we plan to integrate this security architecture in the implementation of the Globe system. We would also like to investigate ways of adding mandatory (site-specific) access control policies to the current architecture. Another topic of research is developing more fine-grained access control mechanisms, based on predicates on environment conditions (time, geographical location, ...) and on the parameters of the method requests.

References

- [1] The common object request broker: Architecture and specification. www.omg.org, Oct 2000. Document Formal.
- [2] Corba security service specification. www.omg.org, March 2001. Document Formal.
- [3] M. Abrams, S. Jajodia, and H. Podell, editors. *Information Security - An Integrated Collection of Essays*. IEEE Computer Society Press, Los Alamitos, CA, 1995.
- [4] A. Bakker, M. van Steen, and A. Tanenbaum. From remote objects to physically distributed objects. In *7th IEEE Workshop on Future Trends of Distributed Computing Systems*, pages 47–52, December 1999.
- [5] G. Eddon and H. Eddon. *Inside Distributed COM*. Microsoft Press, Redmond, WA, 1998.
- [6] A. Grimsaw and W. Wulf. Legion - a view from 50000 feet. In *Fifth IEEE International Symposium on High Performance Distributed Computing*. IEEE Computer Society Press, Aug 1996.
- [7] C. Kaufman, R. Perlman, and M. Speciner. *Network Security*. Prentice Hall, Upper Saddle River, NJ, 1995.
- [8] J. Leiwo, C. Hanle, P. Homburg, C. Gamage, and A. Tanenbaum. A security design for a wide-area distributed system. In *Second International Conference Information Security and Cryptology (ICISC'99)*, volume 1787 of *LNCS*, pages 236–256. Springer, 1999.
- [9] J. S. Park and R. Sandhu. Rbac on the web by smart certificates. In *ACM Workshop on Role-Based Access Control*, 1999.
- [10] C. P. Pfleger. *Security in Computing*. Prentice Hall, Upper Saddle River, NJ, second edition, 1997.
- [11] R. Sandhu, E. Coyne, H. Feinstein, and C. Youman. Role-based access control models. *IEEE Computer*, 29(2):38–48, Febr. 1996.
- [12] R. Sandhu and Q. Munawer. How to do discretionary access control using roles. In *ACM Workshop on Role-Based Access Control*, 1998.
- [13] L. D. Stein. *Web Security*. Addison-Wesley, Reading, MA, 1998.
- [14] M. van Steen, F. Hauck, P. Homburg, and A. Tanenbaum. Locating objects in wide-area systems. *IEEE Communications Magazine*, pages 104–109, January 1998.

- [15] M. van Steen, P. Homburg, and A. Tanenbaum. Globe: A wide-area distributed system. *IEEE Concurrency*, pages 70–78, January-March 1999.