

# A Security Architecture for Object-Based Distributed Systems

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

Maarten van Steen  
Vrije Universiteit  
Amsterdam, The Netherlands  
steen@cs.vu.nl

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

## Abstract

*Large-scale distributed systems present numerous security problems not present in local systems. In this paper we present a general security architecture for a large-scale object-based distributed system. Its main features include ways for servers to authenticate clients, clients to authenticate servers, new secure servers to be instantiated without manual intervention, and ways to restrict which client can perform which operation on which object. All of these features are done in a platform- and application-independent way, so the results are quite general. The basic idea behind the scheme is to have each object owner issue cryptographically sealed certificates to users to prove which operations they may request and to servers to prove which operations they are authorized to execute. These certificates are used to ensure secure binding and secure method invocation. The paper discusses the required certificates and security protocols for using them.*

## 1 Introduction

Security in large-scale 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.

A popular trend in distributed systems is to encapsulate functionality as objects and provide mechanisms for their location, migration and, persistence, as well as for remote method invocation. CORBA [2] [3], DCOM [9], and Legion [13] 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 con-

trol 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 non-replicated objects. This makes it easier to implement a security infrastructure, since security policies for individual objects have to be enforced at only one point: the host where the object resides.

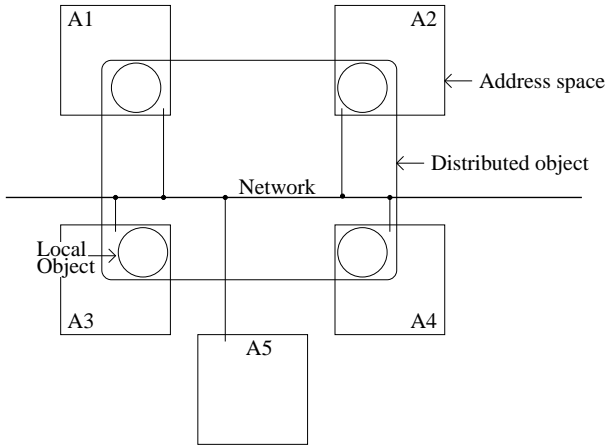
Globe [26], is a wide-area distributed system based on *distributed shared objects* (DSO). The notion of a DSO stresses the property that objects in Globe are not only shared by multiple users, but also physically replicated at possibly thousands of hosts over a wide-area network. Thus, a single object may be active and accessible on many hosts at the same time. Obviously this leads to a consistency problem, but that has been addressed elsewhere [6].

This paper describes the Globe security architecture. Our main contribution is a design that (1) makes a clear separation between the security issues to be dealt with at the middleware level as opposed to the application-specific ones, (2) provides concrete solutions to some unique security challenges, which derive from the fact that Globe objects can be (massively) replicated with some of the replicas running on untrusted, possibly malicious hosts, and (3) is truly decentralized - it does not require any global authority or trusted third party that would severely limit the scalability of the system.

The rest of the paper is organized as follows: Section 2 gives an overview of Globe, the internal structure of a DSO, and the services provided by the Globe middleware that facilitate the creation and deployment of DSOs. In Section 3 we identify the security problems we are trying to solve and in Section 4 we present our trust model. The problems identified in Section 3, which can be grouped as **secure binding** problems, **platform security** problems, and **secure method invocation** problems, are then discussed in Sections 5, 6 and 7 respectively. Finally, in Section 8 we discuss related work and in Section 9 conclude.

## 2 The Globe System

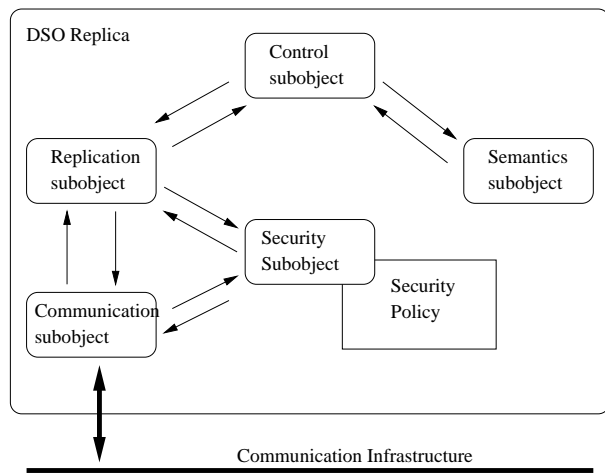
A central concept in the Globe architecture is the distributed shared object (DSO). As seen in Figure 1, a DSO is built from a number of **local objects** that reside in a single address space and communicate with local objects in other address spaces. Each Globe DSO is identified by a unique object ID (**OID**).



**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**. When a user wants to invoke methods on a DSO, it will have to create a local object for that DSO in his own address space. Often, such a local object acts as a **user proxy**, and does not store the object's state, but simply forwards the user requests to replicas that can execute them (but except for possibly increased response latency, this is transparent to the user). For this to happen, the user proxy needs to be initialized with an **object handle**, which consists of the OID of that DSO plus the information needed to find other replicas of that object (e.g. the network address of a replica running a directory service). To facilitate finding DSO replicas, we have implemented the Globe Location Service (**LS**) [25]. DSOs can (optionally) register with this service, in which case they do not have to keep track of their replicas, but only register them with the LS.

All the replicas 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 replica is shown in Figure 2 (a user proxy has a similar structure), and is as follows:



**Figure 2. The internal structure of a Globe DSO. The arrows indicate the possible interactions between the subobjects**

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.

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 replica's state consistent with the other replicas. All replicas part of a DSO participate in an object-specific replication protocol: each replication subobject implements its part of the protocol by mediating the exchange of state-update messages with other replicas. In the case of a user proxy, the replication subobject is responsible for providing the user with the view of a logical, non-replicated object. This view is accomplished by transforming local method invocations into requests that are sent to replicas for further processing.

The control subobject's job is to take care of invocations from client processes on the host where the local object resides and to mediate the interaction between the semantics subobject and the replication subobject. It is comparable to a skeleton stub in object adaptors [24]

The security subobject [17] is responsible for enforcing the DSO's security policy at the level of local objects by mediating the communication flow between the other local subobjects.

Replicas in Globe are generally hosted on **Globe Object Servers (GOS)**. A Globe user wishing to run a replica or a proxy, needs a GOS on his computer, either stand-alone, or integrated in some other application, a Globe-aware Web browser [27] for example. One can think about the GOS as something similar to the ORB (Object Request Broker) in

CORBA. The GOS is responsible for managing the lifecycle of local objects - downloading the class code needed to create them (in most of the cases the DSOs will be responsible with providing this code), instantiating their subobjects, and mediating the use of computing resources (e.g. memory, CPU, disk, network interface). A detailed description of resource management by a GOS is outside the scope of this paper, it is enough to say that our Java prototype GOS implementation deals with all these issues using traditional operating systems techniques. In this paper we will cover only the security-related issues in the GOS design.

### 3 Security Issues in Globe

When designing the Globe security architecture, we chose to follow a modular approach, similar to the one described in [15]: the first step is to analyze the Globe security requirements and identify all the possible mechanisms that can be used to satisfy these requirements. The second step is to select a subset of these functions to be actually implemented as part of the middleware (not all, since some of the functions can be better handled at the application level). Finally, in the third step, selected measures are to be implemented and evaluated. The first stage has been already completed, and the resulting document [17] can be seen as a specification of all security functions that could be incorporated in Globe applications. This paper deals with the second stage, and we are using it as a specification document for our prototype implementation.

As seen in the scientific literature, security issues in distributed systems are not trivial to identify and structure. In our case, the situation is even more complex due to the fact that Globe objects can be replicated across multiple machines, which introduces a series of new problems. For example there is the threat of malicious insiders (replicas running on malicious Globe object servers) which introduces the need to restrict the functionality of replicas depending on the trustworthiness of the system they are running on. Looking at the wide range of security problems identified in [17], we decided to group the security issues relevant to our design into three categories: **secure binding**, **platform security** and **secure method invocation**.

Secure binding effectively establishes that a client, given an OID, installs a local object that is indeed part of the DSO identified by that OID. In addition, it ensures that a replica can be verified to be part of a DSO of which the OID is known. Finally, secure binding allows us to securely associate an OID with real-world entities such as an individual, organization, or company.

Platform security issues derive from the fact that Globe relies heavily on mobile code. The security design should address the problem of protecting hosts from Trojan horses and viruses embedded in the object code that is downloaded

on the fly to start replicas and user proxies. However, platform security does not deal only with threats posed by mobile object code on the host. The reverse problem is also an issue: we need to protect a DSO against possibly malicious hosts. In Globe, a DSO will optimize its performance by placing replicas close to its clients. This placement requires the cooperation of servers over which the (owner of a) DSO has no control, and which may act maliciously. What we need is a mechanism to assure host administrators that running other people's DSOs replicas will not corrupt their system, and also assure DSO owners that replicas of their objects running on hosts outside their control are still following the security policy they have set for their DSO.

Finally, there are a number of issues related to secure method invocation. Any distributed system where security plays even a minor role has to deal with issues like authenticating clients and servers, enforcing an access control policy on user requests, and protecting network traffic. However, with replication involved, as in Globe, we are faced with a new problem. What we also need is **reverse access control**, that is, a means for deciding which replicas should be allowed to execute certain user requests. We need to ensure clients that their requests are sent only to replicas trustworthy enough to execute them.

In the following section we will describe the architectural elements used in Globe security. After having described the basic building blocks, we will see how these blocks are combined in an infrastructure that addresses the issues just outlined.

## 4 The Globe Trust Model

The cornerstone of the Globe trust model is that individual DSOs are fully in charge with their security policies. This means a Globe object **does not need** any external trust broker in order to run securely (but there are mechanisms that allow DSOs to interoperate with external trust authorities, if they **choose** to do this, we will describe these mechanisms later in this section.)

### 4.1 The DSO Trust Hierarchy

Because DSOs can be massively replicated across wide-area networks, we have chosen public key cryptography as the basic cryptographic building block for implementing the DSO trust hierarchy. The alternative, namely to use only shared secret keys, has the disadvantage that we need to take special measures to reduce the number of keys, for example, by using a Key Distribution Center. Although public keys introduce their own scalability problems, such as those related to certificate revocation, we have nevertheless decided to associate public/private key pairs with all distinct

Globe entities (DSOs, replicas, users), believing that these are more easy to deploy in a large-scale system.

We require that each DSO has a public/private key pair, which we term as the **object key**. The object key acts as the ultimate source of trust for the object, and any principal that has knowledge of the object's private key can set the security policy for that object (we term such a principal the **object owner**).

We also associate a public/private key pair with every DSO replica (we call this the **replica key**). The replica key is generated by the GOS hosting the replica at the moment when the replica is instantiated. If multiple replicas of different DSOs run on the same GOS, they cannot tamper with each other's keys, thus replicas of different DSOs do not have to trust each other, even when they run on the same server (however, they would have to trust the server to some extent; we will talk more about this when discussing platform security). Having the GOS protecting the replicas it runs from each other is an architectural requirement. The way this is enforced in practice it is dependent on the way the GOS is implemented.

For DSO users, public-key cryptography is used for authentication and access control. For a given DSO, permissions are simply associated with user public keys, and users are granted those permissions if they can prove knowledge of the associated private keys.

Finally, Globe objects use digital certificates to grant permissions. There are three types of such certificates: user certificates, replica certificates, and administrative certificates. Each of these certificates binds a public key to a set of rights the entity possessing the corresponding private key has with respect to the object. When using digital certificates, one should also consider the problem of revoking them. A detailed description of the revocation mechanisms in Globe is outside the scope of this paper, details are given in [21].

A user certificate specifies which of the DSO's methods the user is allowed to invoke. This information is encoded as a bitmap  $U$  of size equal to the number of methods for that object (for now, we assume the object's public methods do not change during the object's lifetime). A 1 means the user is allowed to invoke that method; 0 means he is not. An example is shown in Figure 3(a).

We can see that user certificates describe the access control policy for the DSO. Replica certificates are used for **reverse access control**, that is, ensuring that user requests are sent only to replicas allowed to execute them. Whenever a user wants to invoke a given DSO method, his user proxy has to find a replica that is allowed to execute the method under the DSO's security policy. Replica certificates are useful when the object owner wants to restrict the execution of security-sensitive methods (e.g. those that change the object's state) to a set of core replicas, while less sensi-

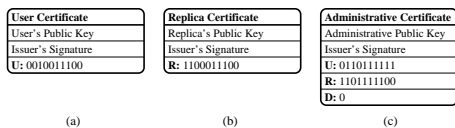
tive operations (e.g. reads) can be executed by less trusted caches. We use replica certificates to specify which methods a replica is allowed to execute. This information is encoded as a bitmap  $R$  of size equal to the number of methods for the DSO. A 1 in the bitmap means the replica is allowed to execute the corresponding method, while a 0 means it is not. An example is shown in Figure 3(b).

Finally, administrative certificates govern the way certificates are issued. They specify the types of certificates an administrative entity (i.e. user or replica) is allowed to issue under the DSO's security policy. For a DSO, any certificate either has to be signed with the object's private key, or has to be part of a certificate chain that starts with an administrative certificate signed with the object's private key. All the certificates in this chain, except possibly the last one, need to be administrative certificates, such that certificate  $C_{k+1}$  is signed with the private key corresponding to the public key in certificate  $C_k$ , and  $C_k$  has been delegated the right to issue certificates of the type of  $C_{k+1}$ . Figure 4 shows an example of such a chain.

An administrative certificate will contain two bitmaps,  $R$  and  $U$ , and a bit  $D$ . The  $R$  bitmap specifies what types of replica certificates the administrative entity is allowed to issue. The  $U$  bitmap specifies what types of user certificates the administrative entity is allowed to issue. The  $D$  bit is called the **delegation bit**, and controls whether the administrative entity is allowed to issue administrative certificates. This organization is shown in Figure 3(c) and works as follows:

- If the  $R$  bitmap in an administrative certificate is not all 0s, the corresponding administrative entity is allowed to issue certain types of replica certificates. The  $R$  bitmap in these replica certificates has to be a subset of the  $R$  bitmap in the administrative certificate.
- If the  $U$  bitmap in an administrative certificate is not all 0s, the corresponding administrative entity is allowed to issue certain types of user certificates. The  $U$  bitmap in these user certificates has to be a subset of the  $U$  bitmap in the administrative certificate.
- **Delegation Rule:** if the delegation bit  $D$  in an administrative certificate is 1, the corresponding administrative entity is allowed to issue certain types of administrative certificates. The  $U$  and  $R$  bitmaps in these newly produced certificates should be subsets of the bitmaps in the issuer's administrative certificate. The delegate can itself be delegated.

At first it may seem strange to have both users and replicas as administrators for an object, since one would usually associate a human with such a role. Administrative replicas come in handy when we deal with massively replicated DSOs. For such DSOs, a highly dynamic pattern in client



**Figure 3. (a) User certificate that allows the invocation of methods  $M_2, M_5, M_6,$  and  $M_7$  of a DSO. (b) Replica certificate that allows the execution of methods  $M_0, M_1, M_5, M_6,$  and  $M_7$  of a DSO. (c) Administrative certificate that allows issuing both (a) and (b) but does not allow issuing any administrative certificates**

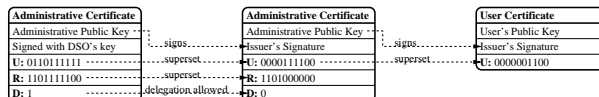
requests can be better handled by creating new replicas on the fly, in places where most of the user requests come from. In such a scenario, one user administrator can manually create (through his user proxy) a number of administrative replicas, and issue administrative certificates granting them the right to issue replica certificates. These administrative replicas could in turn monitor user requests and create regular replicas in places where they can better handle these requests.

Since digital certificates are extensively used to grant permissions, we should also consider the problem of certificate revocation. In Globe, administrative entities are responsible for generating certificate revocation lists (CRLs), and posting them to on-line directories, where they can be consulted by interested parties. Replicas are responsible for checking CRLs before servicing user requests, and also with proving to the users that their replica certificates have not been revoked.

The last concept we need to introduce in this section is replica location. Since some methods can be executed only by certain replicas, a user proxy needs a way to query for replicas allowed to execute certain methods. If the DSO implements its own replica directory service, such a service should register the bitmaps associated with replicas and support queries on individual bits in these bitmaps. Alternatively, a DSO can simply register its replicas with the Globe Location Service, which has been implemented to fully support such queries [5]. It is important to understand that the location service need not be trusted (except for not mounting DoS attacks); the results of a location query are just hints to where replicas with bitmaps allowing the execution of certain methods can be found. Before dispatching a request to a given replica, a client has to verify that replica has indeed been allowed to execute that request. We will see how this is done in Section 7.

Let us now summarize the concepts introduced so far: the Globe security architecture is based on public key cryptography. DSOs, replicas, and users are assigned pub-

lic/private key pairs, so that an entity can be identified through its public key. Finally, Globe entities are granted permissions through the use of digital certificates, as we saw in Figure 3.



**Figure 4. A certificate chain in Globe**

## 4.2 Integrating External Trust Authorities

In the previous subsection, we have shown how Globe DSOs create their own trust hierarchies. However, Globe was designed to support billions of objects, and having to deal with billions of trust roots is clearly not scalable. In practice, human users have a small number of external organizations they trust (the DNS root, the university's system administrator, maybe the local Internet provider). In this section we will show how our DSO-centric trust model can accommodate external trust authorities.

First, a DSO trust hierarchy can be linked to a larger external hierarchy. This can be easily accomplished by having the external trust root sign a digital certificate that associates the DSO's public key to whatever role that DSO plays as part of the external trust hierarchy. Alternatively, a Certification Authority (CA) can certify an organization, so that the organization can then use its certified public key to certify the public keys of all the DSO's that belong to it. To facilitate the inspection of such external certificates, each Globe DSO provides a *show-pedigree()* method that returns all digital certificates that link that DSO to external trust roots.

Another possibility is to generate a DSO trust hierarchy based on a larger external hierarchy (for example a company-wide role based access control scheme). This can be accomplished by providing a mapping that specifies which of the DSOs methods each entity in the external trust hierarchy is allowed to invoke (such an entity can be a principal, group of principals or role). Such a mapping will have to be distributed only to the administrative replicas of the DSO that are in charge with issuing user certificates. These administrative replicas will also have to be fitted with the mechanisms necessary to authenticate principals in the external hierarchy (mechanisms which may or may not be public-key based.) Because these authentication mechanisms are application-specific they will have to be implemented as part of the semantics subobject of the administrative replicas. It is important to understand that the vast majority of non-administrative replicas of the DSO do

not need to know anything about external trust hierarchies. They only deal with the compact security policy described in user certificates.

## 5 Secure Binding

Secure binding boils down to establishing a trust relation between a DSO and its users. Namely, we want to securely associate a DSO to its public key, securely associate a replica to a DSO and securely associate a DSO to a real-world entity.

The first problem - securely associating a DSO to its public key - can be solved by simply making the object's public key a part of the object ID (however, this has the disadvantage of having to change the OID whenever the object key needs to be changed). This is not a new approach. Systems like SFS [18] have pioneered the idea of making the resource key a part of the resource name. For Globe, we decided to apply the same idea to object IDs. As a result, we define a DSO's OID to be the 160 bit *SHA-1* hash [1] of the object's public key. The self-certifying OID is also an elegant solution to another problem, namely how to generate unique OIDs without relying on a central authority. In this case, by simply generating the OID, a user is statistically guaranteed that OID is unique (given it has used a good public key generator algorithm so the key is statistically unique, the probability of a collision for *SHA-1* is extremely low).

Now that we can securely associate a public key with an object, we can also solve the problem of securely associating replicas to DSOs. In Section 4.1 we explained that replica certificates are used to specify which methods a replica is allowed to execute. The replica certificate itself, together with the associated administrative certificate chain, can be used as a proof that the replica is indeed part of the object. Remember that the replica certificate plus the administrative certificate chain securely bind the object's public key to the replica's public key (the key of the user who runs that replica) because the administrative certificate chain must start with a certificate signed with the object's key. We just showed how a self-certifying OID securely binds that OID to the object's public key. Therefore, by simply looking at the OID, at the replica certificate, and its associated administrative certificate chain, one can determine whether a replica is indeed part of the object.

We have shown how secure bindings can be established between objects and their public keys, and between objects and their individual replicas. However, trusting these associations will not convince a user that an object actually does what it is supposed to do. For example, a user may not be willing to use a DSO modeling a home banking application unless she is convinced that a real-world bank is in charge of that DSO. Simply associating a replica with an

object ID and a public key is clearly not enough to establish such a trust relationship. What we need here is a secure name binding, in the example we gave - a binding between an OID (implicitly bound to a public key) and a bank name. We claim that such a name binding needs human interaction in order to be secure. This idea is close to the rationale presented in the SDSI document [23] that accepting another individual's public key and associating it with a local name should always be human mediated.

There are multiple ways one can achieve secure name binding. For example, one could go to the local bank office, sign for the home banking service, and then receive the object handle (OID plus information on how to find replicas) for the home banking application together with the user certificate on a floppy disk. Another possibility is for the user to get the object handle from the bank's Web site through a secure HTTP connection, or through secure e-mail, bind to the object and use one of the object's methods to register for an account and receive a user certificate. Yet another possibility is to get the object handle from an on-line untrusted directory, and invoke the DSO's *show\_pedigree()* method to get the object's "pedigree" certificate signed with the bank's private key (which in turn can be certified by a trusted CA). Globe does not rely on any automatic way of discovering trusted applications, it is the user's responsibility to decide which objects she wants to use. The Globe object server provides only a front end for associating human readable application names to object handles in such a way that when the human user selects an application name, a user proxy is then created on the server, so that the user can invoke the object's methods.

Let us summarize what we have presented in this section. In Globe we make use of self-certifying OIDs to establish secure bindings between DSOs and their public keys. The replica certificates create secure bindings between replicas and the DSOs they are part of. Finally, it is the individual clients' responsibility to establish secure name bindings for the DSOs they are using (but external CAs could mediate this).

## 6 Platform Security

Globe relies heavily on mobile executable code - code that is downloaded on the fly from possibly untrusted sources to instantiate replicas and user proxies. Globe also relies heavily on remote code execution - object code is uploaded and executed on possibly untrusted hosts in order to bring computation close to the clients (this is the dynamic replica creation we briefly discussed in Section 4.1). This is where platform security issues come from. We distinguish two categories: protection against malicious code, and protection against malicious hosts. Since Globe object code is executed on Globe object servers, it is here where platform

security issues are handled.

One of the aims of our research is to create a highly secure object server that prevents malicious replicas from corrupting the host on which they run. Knowing they can host other people's code without danger for their own machine, would convince more people to share their computing resources and run DSO replicas even if they have no prior knowledge of the owners of these DSOs. Our ultimate goal is to create a large peer-to-peer community running Globe object servers. In such a community, object server owners could negotiate to host each other's DSO replicas (we are investigating mechanisms that could automate such negotiation, but this is outside the scope of this paper).

As for a motivating example, consider creating a Globe object modeling a popular Web site and placing replicas of this DSO on hosts on the Internet according to where most of the download requests come from [20]. A community of Globe users running Globe servers that facilitate such Web-site mirroring would, in fact, create a peer-to-peer version of a Content Delivery Network. Mirroring Web documents is only one possible application for a Globe community and other types of applications that could also benefit from adaptive replication algorithms easily come to mind.

Looking at such possible applications, we realize that while the host protection from malicious replica code is essential for the acceptance of such an architecture, it is the reverse problem - protecting replicas from malicious hosts - that would ensure the usability of the system. For example, for Web page mirroring, we need to ensure that an object server that has agreed to host a mirror replica will not maliciously alter the pages it is hosting.

## 6.1 Protection against Malicious Code

With respect to the problem of protecting of hosts against malicious mobile code, our approach is a combination of sandboxing and code signing. We want to emphasize that the focus of this work is not designing new sandboxing tools, but rather using existing ones. We have decided to implement the Globe object server in Java 2.0, which provides extensive support for per-class and per-package configurable security policies, but using other secure sandboxing environments, such as Janus [11], should produce similar results. We require mobile Globe object code to be signed with the object's private key, and we make use of the Java protection domains [12] to associate permissions with code packages signed with different keys. We differentiate between object code actively installed by the user running the object server, and replicas that are installed on the server as a result of remote requests (when a DSO wants to place one of its replicas on that particular server). The user running the object server installs new local objects whenever he wants to use a new DSO. Recall that before using a DSO

a user has to create a name binding between that object's OID and the human-readable name of the application modeled by that DSO. It is during this name-binding process that the user also sets the local permissions for the DSO, namely what set of actions the DSO's local object is allowed to perform on the user's machine. This local security policy is then associated with the object key derived from the OID, so when the object code signed with that key is downloaded, that security policy is automatically associated with it. By default, the security policy for DSO's local objects is quite restrictive, it does not allow any access to the file system or to other system resources such as the printer. It is up to the object server owner to grant more rights to a local object. For example, the local object for a DSO modeling a video-on-demand application might be allowed to write data in the /tmp directory for buffering purposes.

The other case when mobile object code is installed is when a DSO replica with administrative privileges decides to start another replica on some object server. Before the object code for the new replica is installed, there is a negotiation phase between the object server and the administrative replica. The administrative replica has to prove that (1) it is part of the identified DSO, and (2) it is authorized to create a new replica for that DSO. The administrative replica will generally need to negotiate with the object server about resource usage, such as required memory, storage, CPU capacity, and network bandwidth. Note that our scheme easily permits differentiating between objects by associating permissions with OIDs. For example, certain DSOs may be given permission to access part of the server's file system, while others may be prohibited to be hosted at all.

Once this negotiation is completed, the administrative replica needs to produce a replica certificate for the replica to be created. This new replica certificate will contain the object server's public key together with the bitmap corresponding to the object methods the new replica is allowed to execute.

## 6.2 Protection against Malicious Hosts

The second problem we are trying to solve is how to perform trusted computations on an untrusted host. We believe the general problem is extremely hard, possibly intractable. Despite new protection techniques, such as code cloaking, and a variety of hardware solutions, illegitimate modification of software is still a major issue. Since it seems unfeasible to solve the general problem, in Globe we will focus on techniques that reduce the threat of catastrophic DSO state corruption due to replicas running on malicious hosts. In other words, we assume we can always have malicious replicas, but we concentrate on minimizing the negative effect such replicas can have on the DSO's functionality.

One way to achieve this protection is through the reverse

access control mechanism described earlier. Recall that replicas are issued replica certificates that specify which of the DSO's methods they are allowed to execute. In this way, execution of security-sensitive actions can be restricted to replicas running only on trusted hosts. For example the object owner can select a trusted group of core replicas, and set a security policy where all methods that change the DSO's state are executed only on these core replicas. The core replicas can propagate state updates to a much larger set of less-trusted cache replicas. Users can perform read operations on the cache replicas. Although malicious cache replicas can choose to ignore state updates, or even return bogus data as answers to read requests, the harm they can do is limited for two reasons. First, such bogus data is sent only to the fraction of users that are connected to the malicious cache, and a malicious cache cannot propagate bogus state updates to other caches, since the reverse access control mechanism allows only core replicas to execute write requests (of course, there are applications for which a great deal of harm can be done sending bogus data even to a small percentage of the users - stock quotes for example - in such a case, different protection mechanisms need to be considered).

Another mechanism that can be used to achieve security guarantees for methods executed by untrusted replicas is state signing. For example, a Globe-powered Web site (as described in [27]) can have all its individual documents time-stamped and signed with the object's key, so that for each GET request, the client's semantics subobject would check to make sure the untrusted cache replica is returning a properly signed fresh document, with the same title as the link being followed. Through state signing we can achieve highly secure distributed objects, since now all the harm that malicious replicas can do is denial of service. No cache will be able to produce a bogus document, because it would have to fake the DSO's signature. However, state signing is rather application specific. If the state is large, as is the case, for example, with Web sites, we need to find a way to partition that state so that each part can be signed separately. Partitioning needs to be done in units that match the result values of read operations, which is not always possible as is easily seen by considering a result value that needs to be computed such as an average value. Nevertheless, we believe that in many cases, state signing in combination with the more general reverse access control mechanism is a very useful tool in making certain classes of Globe applications more secure.

Finally, we are investigating how a reputation or user-rating mechanism can be used to offer security guarantees to running replicas on untrusted servers [19]. A highly trusted server can be granted more permissions for executing methods (expressed in the replica certificate) than a less-trusted server. The problem is that we need to be able to securely

check whether trust in a server is still justified. This mechanism is subject of current research.

## 7 Secure Method Invocation

Before starting our discussion about secure method invocation, let us first formally define this concept in the Globe context. A method invocation  $M$  issued by a user  $U$  and to be executed on a replica  $R$  is said to be secure if the following conditions are met:

- $U$  is allowed to invoke  $M$  under the DSO's security policy (i.e.,  $U$  has been issued a user certificate with the bit corresponding to  $M$  set).
- $R$  is allowed to execute  $M$  under the DSO's security policy (i.e.,  $R$  has been issued a replica certificate with the bit corresponding to  $M$  set).
- all the network communication between  $U$  and  $R$  takes place through a channel that preserves data integrity, origin and destination authenticity, and possibly also secrecy.

In this context, we say that a user proxy and a replica have established a secure channel if they have exchanged and verified each other's certificates and have established a communication channel that preserves data integrity. Such secure channels are established between security subobjects of local representatives on top of the regular communication channels established at the communication subobject level. They are identified through channel IDs.

For a user, invoking methods on a DSO involves only calling those methods on his proxy. The replication, communication and security subobjects of the user proxy work together to transform the user requests into remote invocations, send them to replicas allowed to handle them, wait for the return values and present these values to the user. We will look at this process in more detail and explain what mechanisms are employed to make it secure.

The user invokes a DSO method by using the interface presented by the control subobject of his proxy. The control subobject first makes sure the user is allowed to invoke that method by inspecting the user certificate (this can also help in providing personalized user interfaces - only show what has a chance of succeeding). This check can be easily circumvented by a malicious user, and is in place only to warn the user if he by mistake invokes forbidden methods. As we will show below, a DSO does not need to trust the user's local object. If the method call is allowed, the control subobject then marshals the method and parameters and passes them to the replication subobject. The replication subobject decides whether the method invocation can be performed locally, or whether it needs to be sent to another replica.

In many cases, user proxies will not even have a semantics subobject, so the request will have to be executed somewhere else. The replication subobject looks at the request, and asks the security subobject to establish a secure channel with a replica that is allowed to execute such a request. The security object first searches in a list of established secure channels to determine whether such a replica is already in that list. If it is, it simply returns the corresponding channel ID to the replication subobject, which uses it to send the request through the communication subobject.

The other possibility is that none of the replicas with which the user proxy has established secure channels is allowed to execute that particular method request. In this case, the security subobject has to find such a replica, either using the Globe Location Service or some other directory service provided by the object itself. As we mentioned earlier, we do not have to trust any of these services, they only provide hints that help the proxy find a replica allowed to execute certain methods. Once such a candidate replica is identified, it will have to prove that it indeed has these rights. Such a proof is part of the protocol for secure channel establishment, which is outlined in Figure 7.

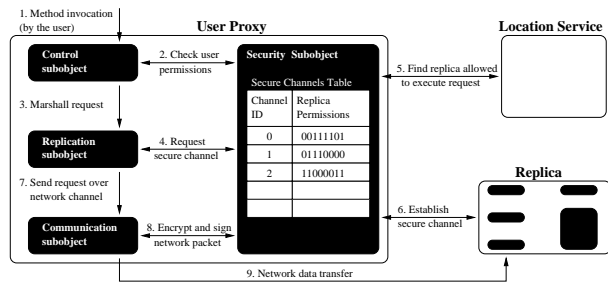


Figure 5. Transforming a user request into a remote method invocation

This protocol is derived from the ISO/IEC 9798-3 mutual authentication protocol [30]. We have chosen to use a challenge-response 4-pass protocol instead of a more compact 3-pass one based on timestamps because in this way our security does not depend on synchronized clocks, which some clients may not have. Note that the replica commits to expensive public key cryptographic operations only **after** the proxy has done the same. This provides basic protection against denial of service attacks. Also, the shared key is generated by the replica since we assume a user may want to install a lightweight version of the object server that may not implement a very strong random number generator algorithm.

After the completion of the above protocol, *Key* becomes the shared secret between the user proxy and the replica and can be used to protect the integrity and secrecy of the data further exchanged by the two parties. Once a secure chan-

nel has been established with the replica, and assuming that the replica is allowed to execute the requested method, the user proxy's security subobject can return this new channel ID to the replication subobject which will use it to send the request through the communication subobject. Figure 5 illustrates the steps described so far.

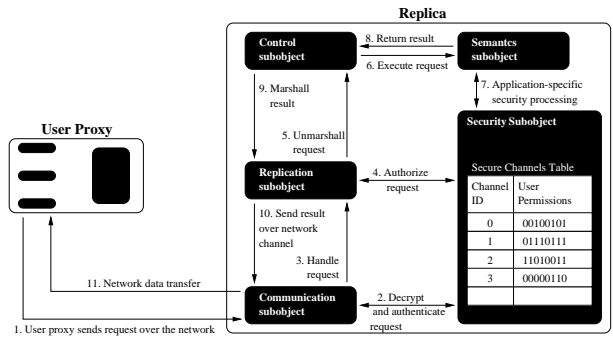


Figure 6. DSO replica handling a user request

At the other end, things happen as shown in Figure 6. Once a replica has established a secure channel with a user proxy, it will store the bitmap from the user certificate in the table of secure channels in the security subobject. For any method invocation request coming in through that channel, the replication subobject asks the security subobject to check the user's permissions. Once the request is approved, the replication subobject passes the marshaled request to the control subobject, which unmarshalls it and passes it to the semantics subobject. After the method is executed, the return value is passed back to the caller over the same secure channel.

**Initial Data**

- |  |   |
|--|---|
| <b>User - U</b>                        | <b>Replica - R</b>                        |
| $C_u$ - user's certificate             | $C_R$ - replica's certificate             |
| $Y_u/X_u$ - user's public/private keys | $Y_R/X_R$ - replica's public/private keys |

**Protocol**

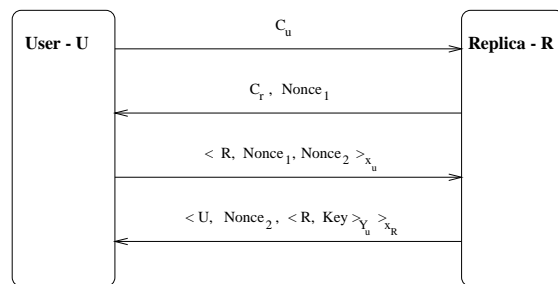


Figure 7. Protocol for establishing a shared key between a user and a replica

## 8 Related Work

In the past decade, distributed systems security has received considerable attention. What makes our design different compared to other work in this area is the fact that we explicitly deal with security problems that arise from dynamically replicating objects on a possibly large set of servers with various degrees of trustworthiness. All the security architectures we are aware of consider at most replication of objects in the same administrative domain, or in mutually trusted domains. The distinctive feature of our architecture is that object replicas can be placed on trusted, but also on less trusted hosts, and replica functionality can be restricted depending on how much trust is put on the host that is running the replica's server.

One of the most comprehensive security models is the one designed for CORBA [3]. The CORBA model has provisions for user authentication, authorization, access control, security of network traffic, auditing, non-repudiation, and security administration. Security itself is implemented in the form of application-specific policy objects, which are invoked when a remote request is dispatched or received. While the CORBA security design is extremely flexible, it is also server centric and may be less scalable over wide-area networks. Furthermore, the CORBA model does not deal at all with mobile code, and has little support for inter-domain security.

What makes the Java [12] security design close to our model is the fact that it explicitly considers the issue of protecting hosts against malicious mobile code. In fact, the platform security part of our design can be implemented using the security features offered by Java 2.0. However, there are a series of security issues handled by Globe which are outside the Java model, such as user authentication, and support for object replication.

Two other projects related to our security design are Globus [10] and Legion [29]. Globus is a distributed system designed for computational grids. Its security model gives extensive support for inter-domain user authentication and remote-process creation, but it is less concerned with trust models for hosts, so in the end users have little control on which machines their code is running. The reverse access control mechanisms in Globe offer a lot more flexibility from this point of view. Finally, the Globus security architecture aims "to provide a thin layer of homogeneity to tie together disparate and, often incompatible, local security mechanisms", which in the end may turn out to be very restrictive.

Legion is another effort in the scientific computation area. There are some similarities between Globe and Legion. For example, they are both object based, and both make use of self-certified object identifiers. However Legion does not deal with dynamic object replication, and in-

troduces a more high-level security design, stressing flexibility and extensibility, but less architecture and protocols.

OASIS [14] is a distributed security architecture centered on role-based access control. Principals can acquire new roles based on roles they already have and their credentials. OASIS also includes a Role Definition Language that can be used for representing security policies based on these roles. However, OASIS does not explicitly deal with replicated applications, and has no support for reverse access control.

Finally, in the past few years we have seen an explosion of peer-to-peer (P2P) applications that have sprung out either as academic projects (SETI@home [4], Publius [28]), or as freeware tools to facilitate media exchange (Napster and Gnutella). What makes such applications interesting is the fact they rely on storage and computation on unsecure platforms and, despite traditional security wisdom, manage to get reasonably accurate results. Much effort is put into models and mechanisms by which the security of these systems can be improved. For example, in OceanStore [16], content can be integrity-checked by clients, whereas other systems concentrate on anonymity [22] or content traceability [7]. Another interesting attempt to provide security and privacy for a P2P architecture is described in [8]. However, many of these systems put emphasis on immutable files, which may severely restrict the area of possible applications. In general, research on secure P2P systems is still in its infancy.

## 9 Conclusion

In this paper we have presented the security architecture for Globe, a distributed system based on replicated shared objects. Our design allows defining per-object security policies, fine-grained (per method) access control and does not rely on any centralized authority that would limit the scalability of the system. Furthermore, we deal only with general security services (since Globe is a middleware) and allow application-specific features to be built on top of these services. Our architecture makes use of well-proven security techniques to address a range of security issues, some common to distributed systems, and others specific to Globe. The fact that Globe objects can be dynamically replicated and simultaneously run on multiple hosts introduces a series of new security problems such as reverse access control for object replicas and protection of distributed objects against malicious hosts running instances of their code. These issues have not been extensively addressed in previous work, and form the major contribution of the research described in this paper.

As for future work, we plan to integrate our security design in the Globe Object Server prototype we have already built. For implementing the platform security features described in Section 6, we plan to use the facilities offered

by Java 2.0 (per-class protection domains), but intend to implement the same functionality using some other sandboxing tool such as Janus. Finally, we plan to integrate a reputation/rating mechanism as a service offered by the Globe middleware, and investigate whether such a mechanism could efficiently filter out malicious Globe Object Servers.

## References

- [1] Secure Hash Standard. FIPS 180-1, Secure Hash Standard, NIST, US Dept. of Commerce, Washington D. C. April 1995.
- [2] The Common Object Request Broker: Architecture and Specification, revision 2.6. www.omg.org, Oct 2000. OMG Document formal/01-12-01.
- [3] CORBA Security Service Specification, Version 1.7. www.omg.org, March 2001. Document Formal/01-03-08.
- [4] D. Anderson. *Peer-to-Peer: Harnessing the Power of Disruptive Technologies*, chapter 5. O'Reilly&Associates, Sebastopol, CA 95472, July 2001.
- [5] A. Baggio, G. Ballintijn, M. van Steen, and A. Tanenbaum. Efficient Tracking of Mobile Objects in Globe. *The Computer Journal*, 44(5):340–353, 2001.
- [6] A. Bakker, M. van Steen, and A. Tanenbaum. From Remote Objects to Physically Distributed Objects. In *Proc. 7th IEEE Workshop on Future Trends of Distributed Computing Systems*, pages 47–52, December 1999.
- [7] A. Bakker, M. van Steen, and A. Tanenbaum. A Law-Abiding Peer-to-Peer Network for Free-Software Distribution. In *Proc. IEEE Int'l Symp. on Network Computing and Applications*, Cambridge, MA, February 2002.
- [8] F. Cornelli, E. Damiani, S. D. C. di Vimercati, S. Paraboschi, and P. Samarati. Choosing Reputable Servents in a P2P Network. In *Proc. of the Eleventh Int'l WWW Conference*, Honolulu, HI, May 2002.
- [9] G. Eddon and H. Eddon. *Inside Distributed COM*. Microsoft Press, Redmond, WA, 1998.
- [10] I. Foster, C. Kesselman, G. Tsudik, and S. Tuecke. A Security Architecture for Computational Grids. In *Proc. ACM Conference on Computer and Communications Security*, pages 83–92, San Francisco, CA, 1998.
- [11] I. Goldberg, D. Wagner, R. Thomas, and E. A. Brewer. A Secure Environment for Untrusted Helper Applications. In *Proc. 6th Usenix Security Symposium*, San Jose, CA, 1996.
- [12] L. Gong. *Inside Java 2 Platform Security*. Addison-Wesley, Palo Alto, CA 94303, 1999.
- [13] A. Grimsaw and W. Wulf. Legion - A View from 50000 Feet. In *Proc. 5th IEEE Symp. on High Performance Distr. Computing*, Aug 1996.
- [14] J. H. Hine, W. Yao, J. Bacon, and K. Moody. An architecture for distributed OASIS services. In *Proc. Middleware 2000*, pages 104–120, Hudson River Valley, NY, April 2000.
- [15] R. Kruger and J. Eloff. A Common Criteria Framework for the Evaluation of Information Technology Security Evaluation. In *IFIP TC11 13th International Conference on Information Security, (SEC'97)*, pages 197–209, 1997.
- [16] J. Kubiawicz, D. Bindel, Y. Chen, P. Eaton, D. Geels, R. Gummadi, S. Rhea, H. Weatherspoon, W. Weimer, C. Wells, and B. Zhao. OceanStore: An Architecture for Global-scale Persistent Storage. In *Proc. 9th ACM ASPLOS*, pages 190–201, Cambridge, MA, November 2000. ACM.
- [17] J. Leiwo, C. Hanle, P. Homburg, C. Gamage, and A. Tanenbaum. A Security Design for a Wide-Area Distributed System. In *Proc. Second International Conference Information Security and Cryptology (ICISC'99)*, volume 1787 of LNCS, pages 236–256. Springer, 1999.
- [18] D. Mazieres, M. Kaminsky, M. F. Kaashoek, and E. Witchel. Separating Key Management from File System Security. In *Proc. 17th Symp. on Operating Systems Principles*, pages 124–139, Kiawah Island, SC, 1999.
- [19] G. Pierre and M. van Steen. A Trust Model for Cooperative Content Distribution Networks. Technical report, Vrije University, Amsterdam, 2001.
- [20] G. Pierre, M. van Steen, and A. Tanenbaum. Dynamically Selecting Optimal Distribution Strategies for Web Documents. *IEEE Transactions on Computers*, 51(6):637–651, 2002.
- [21] B. Popescu and A. Tanenbaum. A Certificate Revocation Scheme for a Large-Scale Highly Replicated Distributed System. Technical report, Vrije University, Amsterdam, 2002. In preparation.
- [22] M. K. Reiter and A. D. Rubin. Anonymous Web transactions with Crowds. *Communications of the ACM*, 42(2):32–48, 1999.
- [23] R. L. Rivest and B. Lampson. SDSI – A Simple Distributed Security Infrastructure. Presented at CRYPTO'96 Rumpsession, 1996.
- [24] D. Schmidt and C. Vinoski. Object Adapters: Concepts and Terminology. C++ Report, 9(11), November 1997.
- [25] M. van Steen, F. Hauck, P. Homburg, and A. Tanenbaum. Locating Objects in Wide-Area Systems. *IEEE Commun. Mag.*, pages 104–109, January 1998.
- [26] M. van Steen, P. Homburg, and A. Tanenbaum. Globe: A Wide-Area Distributed System. *IEEE Concurrency*, pages 70–78, January-March 1999.
- [27] M. van Steen, A. Tanenbaum, I. Kuz, and H. Sips. A Scalable Middleware Solution for Advanced Wide-Area Web Services. *Distributed Systems Engineering*, 6(1):34–42, March 1999.
- [28] M. Waldman, A. D. Rubin, and L. F. Cranor. Publius: A Robust, Tamper-Evident, Censorship-Resistant, Web Publishing System. In *Proc. 9th Usenix Security Symposium*, pages 59–72, Denver, CO, August 2000.
- [29] W. A. Wulf, C. Wang, and D. Kienzle. A New Model of Security for Distributed Systems. Technical Report CS-95-34, 10, 1995.
- [30] R. Zuccherato. ISO/IEC 9798-3 authentication SASL mechanism. RFC3163, August 2001.