ARTICLE IN PRESS

Available online at www.sciencedirect.com
ScienceDirect
Computer Networks xxx (2007) xxx–xxx
Computer Networks
www.elsevier.com/locate/comnet

Design and implementation of a secure wide-area object middleware

Bogdan C. Popescu [a,*], Bruno Crispo [a,b], Andrew S. Tanenbaum [a], Arno Bakker [a]

[a] *Dept. of Computer Science, Vrije Universiteit, De Boelelaan 1081A, 1081 HV Amsterdam, The Netherlands*
[b] *DIT, University of Trento, Italy*

Received 2 January 2006; received in revised form 18 October 2006; accepted 3 November 2006

Responsible Editor: L. Salgarelli

Abstract

Wide-area service replication is becoming increasingly common, with the emergence of new operational models such as content delivery networks and computational grids. This paper describes the security architecture for Globe, an object-based middleware specifically designed to support dynamic replication of services over wide-area networks. Replication introduces a series of new security issues, including the need to restrict replica privileges with respect to method execution, and protection of distributed objects against malicious hosts running instances of their code. Our modular security design addresses these new threats, as well as a broad range of traditional ones, and is validated through a series of performance measurements. Additional contributions include a novel authentication mechanism specifically designed for wide-area deployment, which combines some of the best features of public key authentication protocols (reliance on an *offline* trusted third party in particular) with the computational efficiency characteristic to symmetric key schemes.
© 2006 Elsevier B.V. All rights reserved.

Keywords: Distributed systems; Security; Middleware; Wide area replication

1. Introduction

During the 1980s and 1990s distributed systems were designed according to the traditional client–server model. However, in the recent few years alternative operational models have emerged, including peer-to-peer [59], content delivery networks [67], and computational grids [35]. With these new operational models special emphasis is put on data and application replication. Nevertheless, distributed middleware [8,32] has been slow to adapt to these changes.

Globe [72] is a middleware architecture based on *distributed shared objects* (DSO) which has been specifically designed to cope with data and application replication. The notion of a DSO stresses that objects in Globe are not only shared by multiple users, but also physically replicated on many hosts over a wide-area network. Thus, a single object

* Corresponding author. Tel.: +31 61 767 6123.
E-mail addresses: bpopescu@cs.vu.nl (B.C. Popescu), crispo@cs.vu.nl (B. Crispo), ast@cs.vu.nl (A.S. Tanenbaum), arno@cs.vu.nl (A. Bakker).

1389-1286/$ - see front matter © 2006 Elsevier B.V. All rights reserved.
doi:10.1016/j.comnet.2006.11.008

may be active and accessible on many hosts at the same time.

This paper describes the design and implementation of a security architecture for Globe. Our contribution is threefold: first, we describe a comprehensive security design which is truly decentralized and deals with a large variety of security issues in a modular way. Second, we identify a number of unique security challenges which derive from the fact Globe objects can be dynamically replicated, with some of the replicas running on untrusted, possibly malicious hosts. Finally, we present a number of security mechanisms specifically designed to support replicated services, including a novel symmetric key authentication protocol based on an *offline* trusted third party (TTP), suitable for many wide-area application scenarios.

A prototype of secure middleware architecture we describe has been implemented (in Java) and is available under a BSD-style license. This paper also provides performance measurements for evaluating the overhead introduced by our security mechanisms. These measurements validate our design and show that although security introduces substantial overhead, careful choice of mechanisms and implementation can mitigate this negative impact on performance. Finally, we want to stress that this paper consolidates and expands on some of our earlier contributions, presented in [62,23,64,63], and combines these with the performance numbers we obtained by profiling the secure Globe prototype.

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. Section 3 introduces the threat model considered when designing the Globe security architecture, which is described in detail in Section 4, both at design level, but also in terms of the actual security mechanisms that have been implemented. Section 5 presents our performance measurements. Finally, in Section 6 we discuss related work, and in Section 7 we conclude.

2. An overview of the Globe middleware

The Globe project is aimed at designing and building a flexible and comprehensive middleware platform for developing large-scale distributed applications [72]. Flexibility is achieved by basing the middleware platform on a new model of distributed objects, called the *distributed shared object* (DSO) *model*. A distributed shared object controls all aspects of its implementation. In particular, it controls its extra-functional aspects such as replication and security, rather than delegating them to a common underlying layer. This autonomy enables a developer to naturally apply application-specific replication and security solutions.

A DSO, which is identified by a 160 bit unique object ID (OID), consists of a number of *local representatives* located throughout the Internet. A local representative can act as user proxy, or object replica, or both, allowing client-side replication. As shown in Fig. 1, it is composed of a number of *subobjects*, modules that take care of a particular aspect of the object's implementation: the semantics subobject implements the actual application functionality; the control subobject provides a programming interface to services above the middleware layer, such as user interfaces; the replication subobject takes care of keeping the DSO state consistent across replicas, as well replica placement, while the security subobject takes care of all security-related aspects; finally, the communication subobject hides all network communication aspects from the other subobjects. This modular structure enables the application developer to change the subobjects that handle extra-functional aspects on a per-object basis, providing a great deal of flexibility; furthermore, interfaces of these subobjects are standardized, allowing developers to build extensive libraries of subobjects that can be reused across many different applications. If need be, the developer can also create subobjects entirely tuned to the application.

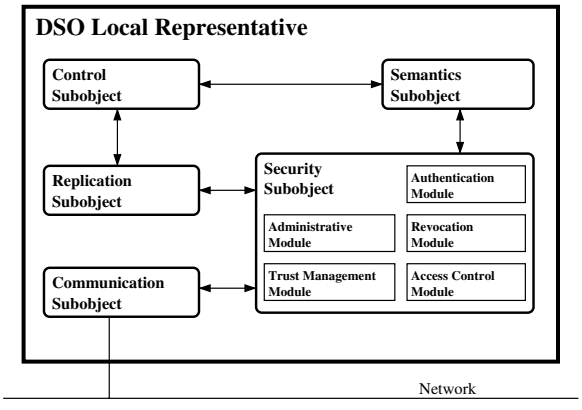

Fig. 1. Internal structure of a Globe DSO local representative. Arrows show possible interactions between the various subobjects.

Please cite this article in press as: B.C. Popescu et al., Design and implementation of a secure wide-area ..., Comput. Netw. (2007), doi:10.1016/j.comnet.2006.11.008

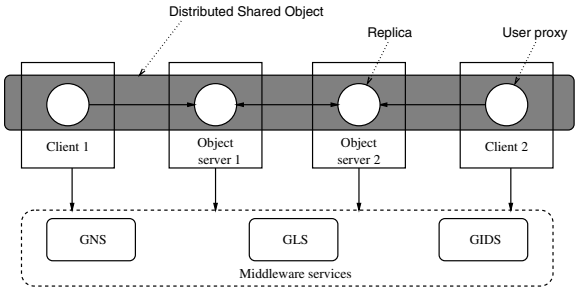

Fig. 2. The architecture of the Globe middleware.

The local representatives of a DSO are generally located on special, user-level *Globe object servers* (GOS), which provide a crash-and-reboot proof hosting platform. The DSOs, the object servers and the client accessing them are supported by a number of middleware services. The *Globe Name Service* [17] maps symbolic object names to binary, location independent OIDs. The *Globe Location Service* [71] efficiently maps an OID to the address of a suitable local representative of the object. Both services have been designed and tested to scale up to 10^{12} objects. To support dynamic replication of objects, Globe provides a *Globe Infrastructure Directory Service* [48], which allows objects to locate additional object servers to host replicas on, given a set of selection criteria regarding network location and security. The architecture of Globe is summarized in Fig. 2.

3. Threat model

Coming up with a comprehensive threat model for a system like Globe is a difficult task for two reasons: (1) there is no central trust authority – no such authority could ever scale to a system envisioned to accommodate millions of users and objects; (2) Globe objects are replicated and replicas may run on platforms controlled by third-parties – it is possible for DSO and GOS administrators to be in different administrative domains.

To simplify defining the threat model, we have divided the security issues Globe needs to address into five groups: *trust management*, *authentication*, *access control*, *Byzantine fault tolerance*, and *platform security*. The first three groups are common when dealing with security in distributed systems: trust management addresses the fact that human users do not implicitly trust software services, but real world entities such as banks, government institutions or other humans; thus it is necessary to map physical world trust relationships into the digital realm. The fact that interactions between DSO local representatives take place across unsecure wide-area networks introduce the threat of masquerading and man-in-the-middle attacks. To counter these, Globe needs to provide strong authentication mechanisms allowing for the establishment of secure communication channels between DSO local representatives. Finally, access control mechanisms are required to counter the threat of un-authorized invocation and execution of object methods.

Besides these "traditional" security goals, the specifics of the Globe system introduce additional issues. The fact that it is possible to have object replicas running on GOSes controlled by marginally trusted third parties introduces the possibility that some replicas may be corrupted by malicious GOS administrators. In order to counter this threat, our security architecture needs to provide Byzantine fault tolerance mechanisms allowing DSO administrators to deal with corrupted object replicas.

Finally, we also need to consider possible threats from the GOS administrators point of view. In Globe, mobile code is used to instantiate new replicas, and this raises the threat of malware (viruses, Trojans, etc.) incorporated into replica executable in order to attack the hosting platform, or perform various types of network denial of service. To counter this, Globe needs to provide platform security mechanisms allowing GOS administrators to protect their computing infrastructure.

4. The Globe security architecture

We now proceed with describing the Globe security architecture, by showing how it deals with each group of security issues identified in the previous section. For each of these groups, we discuss two aspects: the general design guidelines, and the mechanisms actually implemented. It is important to stress that for each of these groups, there are multiple mechanisms for handling them. In most of the cases, we have implemented only one of the possible points in the design space, the one we believe is best suited for the widest class of potential Globe applications. However, as explained in Section 2, Globe objects are composed of a number of subobjects and modules separated through standard interfaces. It is therefore straightforward to replace any of our mechanisms with alternative implementations better suited for specific Globe applications.

Please cite this article in press as: B.C. Popescu et al., Design and implementation of a secure wide-area ..., Comput. Netw. (2007), doi:10.1016/j.comnet.2006.11.008

4.1. Trust management

Trust management is the process of mapping trust relationships between real world entities (for example, between a customer and her bank) to the software instantiation of these entities in the digital world.

Fig. 3a shows the typical trust management scenario in e-commerce applications. The five parties interacting are the user, a service provider (identified by a unique symbolic name), their two software representatives, as well as a trusted third party (TTP) whose purpose is to mediate trust establishment between the software entities. The latter party is typically represented by certification authorities, such as Verisign. There are direct (implicit) trust relationships between the user and the service provider (e.g. bank), the user and its software representative (her web browser in this case) and the browser and the TTP. The TTP is then used to certify the association between the service and its software representative – typically this involves certifying a public key associated with that representative. That public key is then used by the user's browser to authenticate the service's software instantiation, thus accomplishing the mapping of trust from the real to the digital world.

When translating this trust management model to Globe, things become more complicated, given the fact that services can be replicated, and the user's local software representative is normally instantiated using mobile code which cannot be implicitly trusted. The new trust management model is shown in Fig. 3b: here the TTP is used to certify the association between the symbolic object name (which identifies the service provider to the user) and the object ID, which represents the object in the digital world. Associated with each Globe DSO, there is a public/private key pair – the *object key*. The object key is cryptographically bound to the OID – this is accomplished by setting the OID to be the secure hash (SHA-1 [4]) of the object's public key. The net result is *self-certifying OIDs*, similar to the self-certifying file names introduced in [51].

Once trust has been established between the user's Globe runtime and the object's key, this can be used to bootstrap the remaining trust relationships. First, the object key is used for certifying (signing) the mobile code for instantiating the user proxy. The same key is also used for issuing credentials for the replica and the proxy, which use them to authenticate each other, and thus establish trust in the digital realm.

It is important to stress that the presence of a TTP is only required for bootstrapping the trust between the user's Globe runtime and the object's instantiation identified through its OID; all the other trust relationships can be derived from that. As a result, it is also possible to replace the TTP-mediated trust bootstrapping with alternative models, such as the ''resurrecting duckling'' [69], where the user obtains the OID directly from the service provider, for example by going to the local bank office, or a PGP-like [78] recommender model. The Globe architecture makes the trust bootstrapping engine a pluggable module, so that DSO developers can make use of the trust model that best fits their application needs.

4.2. Authentication

After establishing trust, authentication is the next layer required when building secure systems; other security services, such as access control, non-repudiation, audit, and so on, can then be built on this foundation.

Before two DSO local representatives can authenticate each other, they need to obtain the credentials proving they are part of the same object. This is done by registering with a special administrative replica of the object controlled by the object

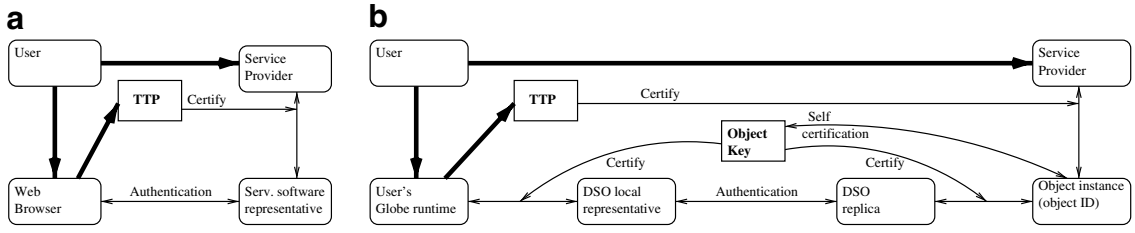

Fig. 3. Trust management (a) typical e-commerce application, (b) Globe DSO; the bold arrows represent implicit trust relationships.

Please cite this article in press as: B.C. Popescu et al., Design and implementation of a secure wide-area ..., Comput. Netw. (2007), doi:10.1016/j.comnet.2006.11.008

administrator (for authentication purposes, the administrative replica is issued a set of credentials certified directly from the object key). Each administrator is free to set any policy for issuing credentials to users and replicas; for users, these policies may include the receipt of a payment (in the case of e-commerce applications for instance), or possession of a credential issued by an external authority (this can be the case for e-government applications). In the case of replicas, the decision to issue credentials can be based on the properties of the GOS hosting the replica (and whether the DSO administrator deems that GOS as trustworthy).

Whenever a DSO entity is registered, it is assigned a per-DSO unique entity ID (replica ID or user ID) which is incorporated into the authentication credentials together with some additional information needed for access control (see Section 4.3). The entity ID is primarily used when revoking credentials, which is done by the same administrative replica. Our implementation assumes only one administrative replica per DSO, which simplifies the management of administrative information. However, this can be changed (by rewriting the administrative module of the security subobject) to support replication of administrative privileges.

4.2.1. Public key authentication mechanisms

Because DSOs can be massively replicated across wide-area networks, public key authentication mechanisms seem to be the natural choice. The great advantage of such mechanisms (as opposed to classic symmetric key based approaches such as Kerberos [45]) is that they do not require the presence of an online trusted third party to mediate authentication.

When public key cryptographic algorithms are used for authentication, there is a public/private key pair associated with every user and replica of the object. All these keys are certified through digital certificates signed by the administrative replica (whose public key is certified by the object key). Once two DSO entities (users, replicas) have these certified public/private key pairs, they can use them for mutual authentication. Our implementation supports the TLS/SSL protocol [30]; however, in Globe the authentication module is pluggable, so it should be straightforward to add other public key authentication protocol implementations.

When dealing with a public key infrastructure, the authentication protocol is only half of the story. Another essential (and often neglected) aspect is key revocation. Globe handles this issue, by allowing the DSO's administrative replica to revoke previously issued credentials. Our implementation provides a simple, yet effective revocation mechanism based on certificate revocation lists (CRLs) [42]. For each Globe DSO there are two such CRLs: one listing revoked user certificates (*user CRL*), and one listing revoked replica certificates (*replica CRL*). The reason we provide separate CRL distribution mechanisms for revoking replica and user certificates, is because of their rather different properties.

In the case of user certificates, it is only replicas that are concerned with checking their revocation status, since direct user to user interactions are not part of the Globe model. Therefore, the user CRL is only distributed to the DSO's replicas; the DSO administrative replica periodically updates the CRL and distributes it to the other replicas. Replicas that are temporary down are responsible for retrieving the latest user CRL when they come back online.

In the case of replica certificates, the replica CRL needs to be distributed to both users and replicas, since both user-replica (for method invocation) and replica–replica (for state update) interactions are possible. Replica certificates are quite different from the user certificates: they are short lived (in some cases replicas may last only few hours in order to handle flash-crowd events) and, in numbers, they are expected to be a few orders of magnitude fewer than user certificates (one replica should be capable of handling hundreds, even thousands of users, otherwise there would be no point in replication). Hence, replica CRLs are much smaller than user CRLs. For this reason, replica CRLs are distributed using a "pull" mechanism: each replica is responsible with storing the latest copy of the replica CRL (which it periodically fetches from the administrative replica), which it provides to users proxies if requested during the authentication protocol.

We believe the revocation mechanisms we have implemented should cover the vast majority of application scenarios for Globe DSOs. Nevertheless, there are many other possible revocation schemes [54,52,53], which can be plugged in Globe DSOs by rewriting the revocation module in the security subobject.

4.2.2. Symmetric key authentication mechanisms

One drawback of public key authentication protocols is the fact they are computationally

Please cite this article in press as: B.C. Popescu et al., Design and implementation of a secure wide-area ..., Comput. Netw. (2007), doi:10.1016/j.comnet.2006.11.008

expensive. After integrating the TLS protocol suite with Globe, our measurements showed a significant performance penalty. The performance penalty was particularly visible for "light" transactions, where the security overhead was several times higher than the actual "useful" CPU work (we will discuss more about performance in Section 5).

On the other hand, symmetric key authentication protocols are computationally efficient. However, the problem with existing symmetric key protocols [57,58,45] is that they rely on an *online* trusted third party. While this is acceptable for a LAN environment, it is not acceptable on a wide-area network, where an on-line TTP becomes a performance bottleneck, a single point of failure, and a target for DoS attacks.

To address this problem, we have designed a new symmetric key authentication protocol based on an *off-line* TTP. The price paid for the increased security offered by an off-line TTP is increased storage requirements for clients. Basically, each participant in the protocol needs to store a number of authentication credentials equal to the size of the authentication realm; this makes the new protocol less scalable than public key schemes. Essentially, the symmetric key authentication mechanism we propose targets small to medium-size Globe applications (hundreds of replicas, tens of thousand clients). It is worth noticing that there are numerous IT environments (small to medium-size companies, university campuses, etc.) which would solicit exactly this type of distributed applications. Furthermore, a possible extension of our protocol, based on *geographical clustering* of clients and replicas, could further increase its scalability to massively distributed applications.

We have developed a new symmetric key authentication protocol to address this issue. The protocol consists of three phases: administrative replica initialization, user/replica registration, and finally the actual authentication, which work as follows.

For initialization, the administrative replica generates two sets of 128-bit AES [1] keys: the replica master key list (RKL) and the user master key list (UKL). The size of the RKL is the maximum number of replicas expected to be instantiated for the DSO, while the size of the UKL is the maximum number of users expected to register.

In the registration phase, users and replicas register with the administrative replica. A user who registers is assigned an unused key from the UKL, and receives an *authentication credentials set* which consists of a number of (*authentication key*, *authentication ticket*) pairs. Authentication keys are shared between users and replicas part of the DSO, with a user having authentication keys for each of the running replicas *as well as* for all *potential* replicas that may be instantiated in the future (thus, the number of authentication keys is equal to the size of the RKL). In this way, when new replicas are instantiated, the users already registered need not be updated when switching replicas.

A similar process occurs when replicas register, except that a replica receives a credentials set including symmetric keys for every potential user (including those expected to register in the future). Thus the number of keys a replica receives is equal to the size of the UKL.

There is an authentication ticket associated with each authentication key. The (*authentication key*, *authentication ticket*) pair allowing DSO entity A (user or replica) to authenticate to entity B has the form

$$(K_{AB}, \{K_{AB}, \mathrm{OID}, \mathrm{ID}_A, \mathrm{ID}_B, T_{\mathrm{issue}}, T_{\mathrm{expire}}\}_{K_B})$$

where K_{AB} is a 128 bit AES key shared between A and B, OID is the object ID, ID_A is the entity ID for A, ID_B is another entity ID (possibly not yet assigned, but expected to be assigned in the future), K_B is another master key (which is assigned or *will* be assigned to B), while T_{issue} and T_{expire} are timestamps that determine the validity interval for the authentication credentials . The ticket can be used by A to prove to B that it is indeed part of the DSO – since it is encrypted with a key shared only between B and the administrative replica.

Once a DSO entity has registered, it can use its credentials to authenticate other DSO entities following the protocol shown in Table 1. The protocol is based on a number of techniques introduced in [22] in order to overcome some of the limitations of the Kerberos authentication protocol [45]. In contrast to Kerberos, our protocol relies on the security property of *keyed hash functions* used as a basic primitive to generate fresh session keys (we have proven the correctness of our protocol using

Table 1
New symmetric key authentication protocol

1	$A \rightarrow B$:	ID_A, N_A
2	$B \rightarrow A$:	ID_B, N_B, $authenticationTicket_{BA}$
3	$A \rightarrow B$:	$\{N_B\}_K$, $authenticationTicket_{AB}$
4	$B \rightarrow A$:	$\{N_A\}_K$

Please cite this article in press as: B.C. Popescu et al., Design and implementation of a secure wide-area ..., Comput. Netw. (2007), doi:10.1016/j.comnet.2006.11.008

the BAN [24] authentication logic, but due to space limitations the proof is not included in this paper).

In the above protocol, N_A, N_B are random nonces, and $K = SHA-1(K_{AB}, K_{BA}, N_A, N_B)$. At the end of the protocol K is the shared secret between A and B and can be used for securing the data traffic between the two.

What makes this protocol different from classical symmetric key authentication schemes is that credentials are long-lived, much in the same way as for public key protocols. Because of this, we have to make provisions for two additional mechanisms: credentials update and credentials revocation.

An update is needed because, in order to prevent possible cryptanalytic attacks, credentials cannot be used forever; after certain time they need to be discarded and replaced with fresh material. One property we want to have in this case is *locality*: an update should only affect the DSO entity that performs it, and none of the others. This is essential for ensuring the scalability of the protocol. Assuming the maximum credential lifetime is T, and M, N are the maximum number of replicas and clients that register during T, the locality property can be achieved by ensuring the administrative replica always has at least M unused keys in the RKL and at least N unused keys in the UKL. Whenever a user registers, it then receives additional credentials for at least M unassigned replica keys, which should cover all possible replicas that may register until the credentials will expire; similarly, when a replica registers it receives additional credentials for at least N unassigned user keys, which should cover all possible users that may register until the replica will update the credentials. In this case, each user receives a credential set of size $2 * M$ and each replica two credential sets of size $2 * M$ (for authenticating to other replicas) and $2 * N$ (for authenticating to users).

When exceptional circumstances occur, the administrative replica may need to revoke credentials before their natural expiration. We use the same mechanisms outlined in Section 4.2.1, namely a separate CRL for replicas and users, which include the ID's of the revoked entities. Validating these CRLs requires a public key signature verification, but this is not a problem from a performance point of view, since CRL verification does not occur that often (a 1 h freshness interval is sufficient for most applications). Furthermore, our implementation uses the RSA algorithm for public key operations, and here signature verification is relatively inexpensive.

Maintaining the locality property with respect to credentials revocation requires further increasing the size of the RKL and UKL. Assuming that an entity that has its credentials revoked is assigned new credentials (not always the case, but this keeps us on the safe side), means that the total number of credentials to be issued becomes larger than the maximum number of entities expected to register. If the probability for credentials revocation is P, the new formulas for the maximum size of the RKL and UKL become $2 * (M + P)$ and $2 * (N + P)$, respectively.

The great advantage of the symmetric key authentication protocol introduced above is the fact that it operates very much in the same way as public key protocols, in the sense that authentication does not require interacting with an online trusted third party. A DSO entity only needs to contact the administrative replica for registration, key update, possibly for obtaining fresh revocation information (in the case of replicas). As a result, interactions with the administrative replica are much less frequent, thus the administrative replica becomes less of a performance bottleneck. Furthermore, it is now possible to enhance the administrative replica with mechanisms for preventing denial of service attacks (for example crypto puzzles [12,76]); this would be killer overhead should this replica have to act as an online TTP, as in traditional symmetric key protocols. However, these advantages come at a certain price.

First, there are increased storage requirements: essentially, each user proxy needs to store a number of credentials proportional to the maximum number of replicas, while each replica needs to store a number of credentials proportional to the maximum number of users and replicas. However, with storage price dramatically decreasing every year, we believe this is an acceptable tradeoff.

Based on our implementation, the size of one (authentication key, authentication ticket) pair is about 100 bytes. Considering the formulas we derived for the RKL and UKL size, we can compute that even for reasonably large DSOs (thousands replicas, hundreds of thousand users), the storage requirements would be in the order of hundreds of kilobytes for user proxies and tens of megabytes for replicas (which is not that much of a problem when considering the average disk size exceeds 50 GB these days). For extremely large DSOs, scalability can be achieved by partitioning authentication credential sets based on replica and user geographical clustering. For example, a user could be given

Please cite this article in press as: B.C. Popescu et al., Design and implementation of a secure wide-area ..., Comput. Netw. (2007), doi:10.1016/j.comnet.2006.11.008

authentication credentials only for replicas and potential replicas) in her network vicinity (the whole point of replication is to match clients with nearby replicas, so there is no point in giving users in Europe credentials for replicas in Australia). It is also important to stress that the application target for Globe is not massive replication, but instead dynamic replication in order to achieve better performance and fault tolerance. In this context, we expect most Globe applications to require a moderate number of replicas (in the order of hundreds) which could be easily handled with our authentication mechanism.

Another drawback of this scheme is the fact that the maximum size of the (user, replica) population needs to be known in advance. However, for many classes of applications, predicting an upper bound of the number of users is not that difficult. This is the case for applications serving closed communities, such as the employees of a company or the students at a university campus. Furthermore, for this upper bound one only needs a rough estimate (the order of magnitude as opposed to the exact number); ''guessing up'', and allocating slightly more master keys that necessary, at worst leads to some small fraction of the users/replicas' storage being wasted. In the case of applications serving worldwide ''open'' communities, where predicting and upper bound for the number of users may be difficult, it is always possible to make use of geographical clustering of users and replicas in order keep the size of credential sets in check.

Finally, the new authentication protocol does not support non-repudiability and delegation, but this is an intrinsic limitation of symmetric key algorithms. For Globe applications where non-repudiation and delegation of privileges are required, DSO administrators should select the public key authentication module.

4.3. Access control

Replication makes the access control model for Globe more complicated, since the concern is not only what permissions users are given, but also what kind of operations a given DSO replica is allowed to execute, since DSO replicas can run on third-party controlled platforms, which may not be equally trustworthy. Thus, the Globe access control model needs to handle two issues: traditional (forward) access control which deals with restricting what operations a user is allowed to invoke, and *reverse access control* – which deals with restricting which operations an object replica is allowed to execute.

Globe supports a discretionary access control model: users and replicas are supposed to enforce the correct invocation/execution of the operations they are allowed to perform. Forward access control is therefore enforced by replicas: a replica is supposed to check the correct invocation (by users) of the methods it is allowed to execute. Reverse access control is enforced by users: a user is supposed to ensure that the execution of the methods she is allowed to invoke only goes to replicas allowed to execute these methods under the object's security policy.

There are three types of rights that can be associated with Globe DSOs:

- Method invocation rights – they grant the permission to invoke the DSO's methods.
- Method execution rights – they grant the permission to execute the DSO's methods.
- Administrative rights – they grant the permission to further delegate rights.

Access control lists (ACLs) are stored by administrative replicas, and associate entity IDs with the rights granted to those entities. The way rights are actually expressed and encoded is dependent on the type of access control mechanism implemented by a given DSO. At this point we support two types of access control mechanisms: coarse-grain and fine-grain. The Globe software distribution provides separate (library) security subobjects for each of these mechanisms; Globe developers can thus choose which mechanism is more appropriate for their objects and incorporate it in the DSO, by selecting the appropriate subobject. We briefly discuss each of these mechanisms in Sections 4.3.1 and 4.3.2.

All our security subobject implementations only supports one administrative replica (which has all possible administrative rights) per DSO. This greatly simplifies ACL management. If administrative tasks need to be replicated, it is important that the replication protocol ensures ACLs are kept consistent. It is up to the DSO administrator to decide on the policies to follow when granting rights to users and replicas. Similar to what we described in Section 4.2, these policies can be based on external certification of entities, payments, or web of trust-based schemes, like PGP [78]. Once the privileges

Please cite this article in press as: B.C. Popescu et al., Design and implementation of a secure wide-area ..., Comput. Netw. (2007), doi:10.1016/j.comnet.2006.11.008

associated with a DSO entity have been set, they are encoded into a *entity rights descriptor* – an opaque, platform-independent, data structure – which is then stored in the (global) DSO ACL and also incorporated in the entity's authentication credentials; in this way, once two entities authenticate, they know exactly what methods they are allowed to invoke/execute.

The access control module in the security subobject provides a very simple interface, consisting of three methods:

- *registerRights*(*channelID*, *privilegeDescriptor*) is called by the authentication module once a secure channel has been established with another DSO entity. The result of this method is that the rights descriptor part of the other entity's authentication credentials is associated with the secure channel.
- *isAllowed*(*channelID*, *methodID*, *parameters*) is called by the replication subobject once it receives a method invocation request from another DSO entity (with which a secure channel has been previously established). The replication subobject provides the channel ID corresponding to this secure channel to the caller, as well as information about the method being invoked (e.g. which method, which invocation parameters). The access control module returns *True* or *False*, depending on whether the rights associated with the corresponding channel (rights previously set by *registerRights*()) allow or deny the given method invocation request.
- *channelID findReplica*(*methodID*, *parameters*) is called by the replication subobject when a (local) method invocation request cannot be executed locally, and thus needs to be sent to another DSO replica. The module first evaluates the request against the DSO's (reverse) access control policy, and determines the rights needed for executing the request. The module then queries the Globe Location Service (see Section 2) to find a replica that has the required rights. To assist this query process, replicas register their rights descriptor with the GLS. The GLS treats this descriptor as an opaque bitmap, supporting exact match queries as well as queries on individual bits. The actual GLS query format depends on how rights are encoded in the rights descriptor. We provide concrete examples in Sections 4.3.1 and 4.3.2. It is important to understand that the GLS needs not to be trusted, essentially the replica contact points returned by a *findReplica*() call being treated as mere hints. The local representative always authenticates a replica before sending the method invocation request, in order to ensure the rights registered with the GLS match those in the replica's authentication credentials.

4.3.1. Coarse-grained access control

For our coarse-grain access control mechanism, the access control granularity is set to the object method level. In this case, the rights descriptor associated with a DSO entity (user/replica) consists of two bitmaps – the *forward access control bitmap* (FACB), and the *reverse access control bitmap* (RACB) – encoding the method invocation rights (FACB), and the method execution rights (RACB) granted to that entity. Each bit in these bitmaps corresponds to one of the object's methods; a 1 in the bitmap allows the invocation/execution of the corresponding method, while a 0 denies it.

In this case the implementation of three interface methods described in the previous section is quite straightforward: *registerRights*() simply associates the FACB and RACB in the authentication credentials with the corresponding channelID; *isAllowed*() returns *True* if the bit corresponding to the method being invoked is set in the FACB associated with the channel (and *False* otherwise); *findReplica*() does a bit-query on the GLS, essentially searching for a replica that has registered a RACB bitmap with the bit corresponding to the requested method set.

Finally, an additional *delegation bit* is used to express administrative rights. An entity that has the delegation bit set is allowed to further delegate all the rights it has been granted, including the delegation right. Operationally, administrative rights are implemented as special administrative methods, provided by each DSO as a standard administrative interface. This interface includes methods for registering users and replicas, issuing and retrieving revocation lists and so on. An entity that has the delegation bit set in its rights descriptor (implicitly) has execution rights for all these administrative methods.

The coarse-grain access control model described in this section is quite simple, but very intuitive, and should cover a wide variety of application scenarios (in most cases, security policies tend to be simple, consider for example the UNIX *rwx* protection model). Although it cannot support com-

Please cite this article in press as: B.C. Popescu et al., Design and implementation of a secure wide-area ..., Comput. Netw. (2007), doi:10.1016/j.comnet.2006.11.008

plex access control policies (based on method parameter values for example), it still gives DSO administrators a considerable degree of flexibility by allowing them to define finer-grain privileges by choosing a more fine-grain object interface (for example a DSO modeling an e-newspaper could provide separate *read*() methods to differentiate between subscribers and the general public). For applications that require more complex policies, we provide a fine-grain access control model, which is described next.

4.3.2. Fine-grained access control

Although there are many application scenarios where simple access control policies are sufficient, there are cases where a more refined access control granularity may be necessary; for example, an e-banking application may require different security properties when requesting the same action – a money transfer – depending on the amount being transferred.

To address these issues, we have designed a new access control policy language capable of describing fine-grain method invocation and execution rights, based on parameter values, as well as other external conditions. Furthermore, using this policy language it is also possible to express exceptional method invocation and execution behavior:

- *Composite invocation subjects* – used to describe access control policies where a number of users must collaborate for invoking a particular method instance.
- *Composite execution targets* – used to describe access control policies where a number of DSO replicas must collaborate for executing a particular method instance.
- *Audited execution* – used to describe access control policies where results computed by marginally trusted replicas must be audited by other (more trustworthy) replicas.

The main reason for supporting such exceptional method invocation/execution behavior is Byzantine fault tolerance. Composite invocation subjects are useful for protecting sensitive DSO operations by means of separation of privileges [21], which may be mandated by organizational policies (for example, many banks require high-value transactions to be approved by more than one bank official). Composite execution targets and audited execution are useful for protecting DSO operations against possible compromise or malicious behavior from some of their less trusted replicas.

4.3.2.1. Basic idea. The basic idea for designing a more fine-grain access control framework for Globe comes from the observation that, for a given DSO, there are only a limited number of logical "roles" that can be associated with entities part of that object. Such "roles" may be dictated by the DSO's replication strategy (for example "master" and "slave" replicas in the case of master-slave replication), or by the administrative and security policies required by the application (for example "gold" and "platinum" users for a DSO modeling a subscription-based e-newspaper).

RBAC [66] is a relatively new access control framework where privileges are associated with logical roles, and entities are granted membership into roles based on their competencies and responsibilities in a given application scenario. Our idea is to design a fine-grain access control framework for Globe DSOs based on RBAC. As such, we define a *DSO role* as a subset of the set of all rights that can be associated with a Globe DSO. An observation here is that when rights granularity is set to method parameter values level, the total number of rights that can be associated with a DSO can be very large, possibly infinite; in this case, the number of possible roles is even larger (since it grows exponentially to the number of the possible rights). However, as previous research on RBAC has shown [65], only a small fraction of all possible roles are actually useful for describing meaningful security policies; as such, we believe that for most DSO role hierarchies will be quite small.

There are three types of DSO roles: *administrative roles*, *user roles* and *replica roles*. Administrative roles include administrative rights, while user and replica roles only contain method invocation and method execution rights. A DSO entity may be assigned to more than one role.

Based on this definition of a DSO role, the security policy for a given Globe object can be (logically) split into two parts: (1) the *DSO role specification* – a data structure defining the DSO role hierarchy (the set of all roles and the relationships among them), as well as the detailed permissions (rights) associated with each role; (2) the *DSO role assignment list* details what roles are assigned to each DSO entity (user replica). The role specification is essentially a text file written in a special declarative policy language; this file consists of policy state-

Please cite this article in press as: B.C. Popescu et al., Design and implementation of a secure wide-area ..., Comput. Netw. (2007), doi:10.1016/j.comnet.2006.11.008

ments describing the object's role hierarchy and the specific rights associated with each role. Ref. [64] provides a detailed description of our policy language; in this section we only give a brief overview of this language.

The DSO role specification file is generated by the DSO administrator when the object is created, signed with the object key, and distributed to all DSO entities at initialization time (i.e. when a user registers with the object, or when a new replica is instantiated). The role assignment list consists of *entity ID*, *entity Role* pairs, and is stored by administrative replicas. In addition to this, an entity's authentication credentials also include the role(s) assigned to that entity. DSO replicas also register their role(s) with the GLS.

4.3.2.2. Administrative rights. An administrative right represents the ability of an administrative role to delegate another role. Such a right is expressed through a policy statement of the form:

$Role_A$ **canAssign** $Role_B$

where $Role_A$ is an administrative role and $Role_B$ is another role (possibly an administrative one as well). A DSO's role specification file contains a number of such statements, describing the role hierarchy for that DSO.

Then, an intuitive way to see a DSO's role hierarchy is as a directed graph, with each node corresponding to a role; in such a graph, an edge from a node $Role_A$ to a node $Role_B$, implies that role $Role_A$ is an administrative role, and has the right to delegate role $Role_B$ under the DSO's security policy.

The way in which this role graph is organized may restrict the types of authentication mechanisms that can be used. For instance, it is difficult to support delegation by means of symmetric key authentication; in this case a flat administrative hierarchy (corresponding to a two-level tree), with one administrative entity doing both the key distribution and the role assignment is recommended. Applications requiring delegation of administrative privileges (essentially a multilevel role graph) should use public key authentication. In this case, there is a credentials chain associated with each DSO entity: this chain consist of the entity's public key certificate and all the certificates associated with intermediate certification authorities that have (recursively) delegated rights to that entity. In this case, the *registerRights* method takes this entire credentials chain as one of its inputs. This method checks this credentials chain to ensure the following properties:

- Certificates in this chain are correctly "chained" – this means each certificate is signed with the public key in the previous certificate in the chain.
- The first certificate in the chain (corresponding to the top-level role in the DSO's role hierarchy) is signed with the DSO's public key (which is implicitly granted all possible rights that can be associated with the DSO).
- The sequence of roles in the certificates in the chain correspond to a valid path in the DSO role graph (this means that for each certificate, the role in the certificate **canAssign** the role in the next certificate in the chain).

4.3.2.3. Method invocation rights. A method invocation right represents the ability of an (non-administrative) role to invoke a given method instance. Such a right is expressed through a policy statement of the form:

RoleSubject **canInvoke** *Method* **underConditions** *Conditions*

Here *RoleSubject* is a role name, or a combination of role names, previously declared in a **canAssign** statement; role combinations are useful for allowing granting of method invocation rights to *composite role subjects*. *Method* is the name of one of the DSO's methods; *Conditions* is a boolean expression that puts constraints on the way the *Method* can be invoked by the role, or composite role subject.

A composite role subject consists of a number of entities assigned certain roles, which must cooperate in order to perform a specific action (a method invocation in this case). The generic role subject format is

```
<RoleSubject>:: <Role> | <CompositeRoleSubject>;
<CompositeRoleSubject>:: <RoleGroup> |
                         <RoleGroup> ''&&''
                         <CompositeRoleSubject>;
<RoleGroup>:: <PositiveInteger> ''*'' <Role>;
```

For example a role expression of the form $3 * Role_A \&\& 2 * Role_B$ implies that a given DSO method, under certain condition, can be invoked only by a composite role subject consisting of three

Please cite this article in press as: B.C. Popescu et al., Design and implementation of a secure wide-area ..., Comput. Netw. (2007), doi:10.1016/j.comnet.2006.11.008

users assigned into $Role_A$ and two users assigned into $Role_B$. These five users must collaborate and agree on this method invocation. Composite subjects are useful for protecting sensitive operations by means of separation of privileges [21]; in many cases, such separation of privileges is mandated by organizational policies (for example, banks requiring high-value transactions to be approved by two bank officials).

A DSO's forward access control policy can be fully described through a set of **canInvoke** statements, specifying which roles, or composite role subjects are allowed to invoke each of the DSO's methods, and under which conditions. This policy is stored in the DSO role specification file. Upon receiving a method invocation request, a replica calls the *canInvoke*() function (implemented by the access control module), passing the method name, parameters, as well as the role of the entity making the request (or the identities of entities collectively making the request, in case of a composite invocation subject). *canInvoke*() then searches the role specification file; if an appropriate *canInvoke* statement is found, *canInvoke*() returns *True* (allowing the request to be executed), otherwise it returns *False*. DSOs that support composite subjects need to provide a special user interface allowing multiple users to collaborate in making a composite subject method invocation.

4.3.2.4. Method execution rights. A method execution right represents the ability of an (non-administrative replica) role to execute a given method instance. Such a right is expressed through a policy statement of the form:

RoleExpr **canExecute** *Method* **underConditions** *Conditions*

Here *RoleExpression* is a role expression, *Method* is the name of one of the DSO's methods; *Conditions* is a Boolean expression that puts constraints on the way the *Method* can be executed.

RoleExpr is an expression of the form:

A role expression has the generic form:

```
<RoleExpr> :: <CompositeRoleSubject> |
              <CompositeRoleSubject>
              ''auditedBy'' <Role> |
```

Such an expression is used to describes a group of replicas, and the way these replicas should be contacted by a user that wants to invoke *Method* (under some given conditions) on the DSO. There are three types of method execution behavior:

Regular execution: the method is executed by one replica. For example, a policy statement of the form:

$Role_A$ **canExecute** M **underConditions**..;

states that an execution request for method M in the given conditions can be handled by a replica in role $Role_A$.

Replicated execution: the same method is executed on a number of replicas, and the result is accepted only when a majority of them agree on the return value. This allows expressing Byzantine fault-tolerance policies based on *state machine replication* (see Section 4.4). In this case the number of types of replicas than need to be contacted is expressed as a *composite role subject*. For example, a policy statement of the form:

$3 * Role_A \&\& 2 * Role_B$ **canExecute** M **underConditions**..;

states that an execution request for method M needs to be handled by three replicas in $Role_A$ and two replicas in $Role_B$, and a majority of these have to agree on the result before the user accepts it.

Traceable execution: a replica or composite replica subject that executes a method has to sign (with their respective replica keys) the invocation request and the return value. The user proxy then forwards these traceable request-result pairs to an auditor replica. Auditing involves re-executing the request and comparing the result with the one signed by the replicas. In this way, less trusted replicas acting maliciously can be traced and eventually excluded from the DSO (we discuss this technique in more detail in Section 4.4). For example, a policy statement of the form:

$Role_A$ **auditedBy** $Role_B$ **canExecute** M **underConditions**..;

states that an execution request for method M can be handled by a replica in role $Role_A$. However, this replica needs to sign the computed result, before returning it to the user. The user proxy needs to forward this signed result to a replica in role $Role_B$ which does the auditing.

A DSO's reverse access control policy can be fully described through a set of **canExecute** statements, specifying the execution behavior for the DSO's methods. This policy is also stored in the DSO role specification file. When a user invokes a

Please cite this article in press as: B.C. Popescu et al., Design and implementation of a secure wide-area ..., Comput. Netw. (2007), doi:10.1016/j.comnet.2006.11.008

given DSO method, the user proxy calls the *findReplica*() function (implemented by the proxy's access control module), passing the method name and invocation parameters. *findReplica*() then searches the role specification file in order to find a **canExecute** statement describing how the given method instance should be executed. Once such a statement is found, *findReplica*() queries the GLS for replicas matching the roles in the statement's role expression, connects to these replicas and mutually authenticates, and finally issues the request according to the execution behavior specified in the statement (regular, replicated, or traceable execution).

4.3.3. Conditions

The **underConditions** clause in a **canInvoke/ canExecute** policy statement allows to express fine grain constraints on method invocation/execution rights. Essentially, a condition is a logical expression, or a number of logical expressions connected using the *and*, *or* and *not* logical operators. Such logical expressions combine numbers, method parameter names, external functions, and additional entity attribute names using logical and arithmetical operators. The exact syntax for describing such conditions is described in detail in [64]. Here, we only briefly discuss how the various syntactical elements can be combined:

- *Method parameter names* can be used for expressing constraints on method invocation rights based on the actual parameter values for a given request. For example, consider a Globe e-banking application; *transferFunds* is one of its public methods, *amount* is one of the method parameters, and *Clerk* is a role. A policy statement of the form:

  Clerk **canInvoke** *transferFunds* **underConditions** $(\text{amount} < 10,000)$;

  restricts a DSO user assigned a *Clerk* role to only transfer amounts less that 10,000 dollars.
- *External functions* can be used for expressing constraints on method invocation rights based on functions external to the access control module. Such functions have to be separately declared in the policy data structure, so that the policy engine knows how to invoke them. External functions can impose constraints on the way a method can be invoked based on things like the DSO state, the resources available on the system running the replica, time of the day, or the location where the request originates. The only requirement here is that such external functions are synchronous – this ensures the policy engine cannot be blocked on an external function.
- *Additional attributes* – these are expressed as name-value pairs and are incorporated in the caller's authentication credentials. The main purpose of such attributes is to allow a more refined differentiation among DSO entities, while keeping the role hierarchy reasonably simple. As an example, considering the same e-banking application, we assume a policy requirement stating that a manager can only read the accounts of the customers she has been assigned. In this case, *Manager* is a role, *readAccount* is a DSO public method, *customerName* is one of its parameters, *extAttr_name* is an additional attribute – part of a user certificate – that stores the user's name, and *whichManager* is an external function which, provided a customer name, returns the name of the manager assigned to that customer. Then, the policy requirement can be expressed through a statement of the form:

  Manager **canInvoke** *readAccount* **underConditions** $(extAttr\_name == whichManager(customerName))$;

  Without external attributes, the only way to express such a rule would be to create a separate role for each individual manager; this clearly would not be a scalable solution.

4.4. Byzantine fault tolerance

As discussed in Section 3, one of the key aspects in Globe is that DSO local representatives are decoupled from the resources/physical infrastructure on which they run, since DSO administrators have the option of creating replicas on Globe object servers outside of their administrative control. It is important to stress that Globe provides this feature as additional flexibility given to DSO administrators for choosing the replication strategy that best fits their needs, as opposed to a design constraint. Each DSO is free to choose any policy with respect to which servers are allowed to host its replicas. While for highly secure applications such as e-banking, running on untrusted infrastructure may be unacceptable, there are many less security-critical scenarios where replication on marginally trusted platforms can be extremely beneficial, as

Please cite this article in press as: B.C. Popescu et al., Design and implementation of a secure wide-area ..., Comput. Netw. (2007), doi:10.1016/j.comnet.2006.11.008

demonstrated by the advent of content delivery networks and grid applications.

Deciding on how much trustworthiness to demand from the hosting infrastructure is ultimately a cost/benefits analysis problem for DSO administrators. Nevertheless, perfect security is practically impossible, so for any DSO we need to consider the possibility that at least some of its replicas may be corrupted, either due to malicious GOS administrators, or to vulnerabilities in the hosting platform itself (in the operating system, or in the GOS code). In this context, it becomes essential to provide mechanisms for Byzantine fault tolerance.

Our observation is that mechanisms for handling Byzantine failures fall into two categories: those aiming at damage prevention, and those aiming at damage control. We will look at each of these categories, and explain how they could be integrated with the Globe security architecture.

4.4.1. Mechanisms for damage prevention

The aim of such mechanisms is preventing malicious replicas from causing any damage to their DSOs, except possibly denial of service. The optimal solution in this case is *trusted remote computation* as proposed in [37,11,31,7,60]. Unfortunately, research in this area is still in an incipient phase, and we expect it will take years until such secure platforms will be widely available. Because of this, in Globe we focus on techniques for minimizing the impact of corrupted replicas on the correct operation of a DSO.

Another mechanism that can be used for damage prevention is state signing [36]. The idea is to have a trusted replica sign the results of read operations, which can then be distributed to marginally trusted replica which then serve them to clients. Before accepting a result, a given client first verifies that the result is correctly signed by a trustworthy replica. The fine-grain access control mechanism outlined in the previous section can be used for differentiating between these different classes of replica (e.g. in the simplest case, having two roles *TrustedReplica* and *Marginally TrustedReplica*). As an example, a Globe-powered Web site (as described in [73]) can have all its individual documents timestamped and signed with the object's key, so that for each GET request, the client's proxy can check that the untrusted cache has returned a valid document. Through state signing we can achieve highly secure distributed objects, since the harm that malicious replicas can do is limited to denial of service. 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 (consider a result value that needs to be computed, e.g. an average value). Nevertheless, we believe that in many cases, state signing, in combination with the more general reverse access control mechanism, can make certain classes of Globe applications safe to run on untrusted infrastructure.

Yet another mechanism for handling Byzantine fault tolerance is through state machine replication [68,26,46]. The idea here is for the client to invoke the same method on a number of replicas (quorum), and accept the result only when the same result is returned by the group majority. The fine-grain access control framework described in Section 4.3.2 can be used to express such replicated method execution behavior.

4.4.2. Mechanisms for damage control

The techniques outlined in the previous section are quite powerful, but cannot be used indiscriminately. Research on trusted computing is still in an incipient stage, and it will take years until software/hardware platforms supporting it become mainstream. State signing is only suited for a rather restricted class of applications – those mostly dealing with static data reads. Finally, state machine replication can be quite expensive to implement, since the amount of resources required for a given method invocation grows linearly to the quorum size. The client-perceived request latency is also likely to increase in this case, since it is dictated by the slowest replica in the quorum.

Given these shortcomings, for the Globe security architecture we are also considering a second class of Byzantine fault tolerance techniques, which aim at restricting the amount of damage malicious replicas can cause to a DSO, detecting when this damage occurs, and possibly taking corrective action after the breach. Clearly, restricting, detecting, and repairing is not as good as preventing altogether, but on the other hand, this second class of mechanisms are much more efficient to implement, hence the potential tradeoff. Damage-restricting mechanisms are based on an optimistic assumption that replica corruption happens relatively infrequently, so security mechanisms should be designed to handle the common case efficiently – namely when everything works correctly.

Please cite this article in press as: B.C. Popescu et al., Design and implementation of a secure wide-area ..., Comput. Netw. (2007), doi:10.1016/j.comnet.2006.11.008

One way to do damage control is through the reverse access control mechanisms described in Section 4.3. Recall that replicas are granted method execution rights that specify which of the DSO's methods they are allowed to handle. In this way, execution of security-sensitive operations can be restricted to trusted replicas (for example those running on trusted hosts). This technique can be used in conjunction with *result auditing*. Here, the basic idea is to have marginally trusted replicas sign the results they produce; these results are then audited by (trusted) auditor replica. In order to be able to do this, the auditor replica needs to "lag behind" with state updates, essentially delaying such updates until it has audited all results computed for a given state version.

For the remaining of this section we will examine this audit-based Byzantine fault tolerance mechanism in more detail, and show how it can be integrated with the overall Globe security architecture. A more formal description of this technique is presented in [63].

4.4.2.1. Operational model. Without loss of generality, we assume that a Globe DSO employing our audit-based Byzantine fault tolerance mechanism provides two types of operations (methods): *read operations* do not change the DSO state, while *write operations* do change the DSO state. Furthermore, we group the entities (clients, replicas) involved with the DSO into four categories:

- *Clients* perform read and write operations by invoking the DSO's methods.
- *Slave replicas* handle the clients' read requests.
- *Master replicas* handle the clients' write requests.
- *Auditor replicas* audit the results of the read requests computed by the slaves in order to ensure they are correct, and take corrective action when detecting incorrect results.

A DSO following this operational model can be designed using the fine-grained access control mechanism described in Section 4.3. This can be done by designing a role hierarchy that specifies different replica roles for executing the DSO state changing and non-state changing methods, and using the **auditedBy** policy statement (see Section 4.3.2) to specify the auditor roles. We also require the DSO to employ public key-based authentication mechanisms between clients and replicas.

As for our state consistency model, we guarantee that write operations are executed in some sequential order (state changes occur in the same sequence on all replicas). On the other hand, we only provide a weaker form of consistency for read operations: essentially the result of a read request (from a given client) may not take into account state changes that have occurred in the "recent past".

A special parameter *maxLatency* is used to quantify this concept of "recent past". Essentially, our read consistency model can be formalized as follows: a write operation that occurs at time T_1 is guaranteed to be taken into account for all read operations performed after time $T_2 = T_1 + maxLatency$. For this definition, we assume that all DSO replicas and clients have loosely synchronized clocks, and the clock skew is negligible compared to the *maxLatency* parameter.

4.4.2.2. The write protocol. Given this consistency model, the algorithm for handling write operations is shown in Fig. 4:

There are four (logical) steps:

1. A client connects to a master replica and invokes a write method. The replica performs the access control check, and if the client is allowed to invoke the method, it executes it.
2. The master replica propagates (broadcasts) the write to the other master replicas. The write is propagated in a way that guarantees some sequential ordering of concurrent write operations. There are many atomic broadcast protocols that accomplish this [44,27,19]; for example efficient atomic broadcast [44] could be a likely candidate. In addition to the sequential ordering of write operations, we also require the DSO to have a special internal variable – *stateVersion* – which is (atomically) incremented with each write

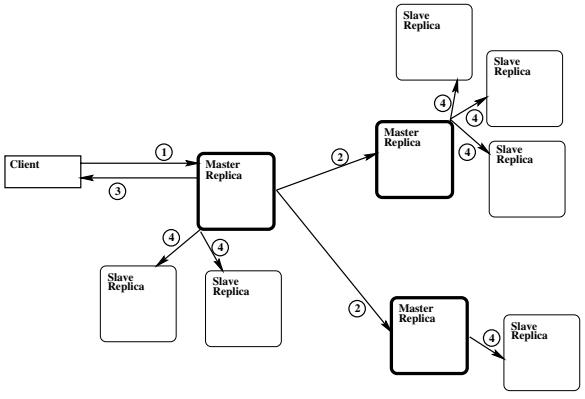

Fig. 4. The write protocol.

Please cite this article in press as: B.C. Popescu et al., Design and implementation of a secure wide-area ..., Comput. Netw. (2007), doi:10.1016/j.comnet.2006.11.008

operation. Given the authentication and access control mechanisms described in the previous sections, it is assumed that any messages exchanged between trusted replicas (including those part of the atomic broadcast protocol used to propagate writes) are transmitted over secure (i.e. preserving at least data integrity, possibly confidentiality). As such, any atomic broadcast protocol that operates on a crash-fault model is sufficient here. If the network is the attacker, it can only drop or delay messages, but, given the secure communication channels assumption, it cannot inject or manipulate valid messages. In these circumstances, any network attack is equivalent to a node crash.

3. At the end of Step 2, the write has been propagated to all the master replicas, and all these replicas have incremented their *stateVersion* internal variable to reflect the change. At this point, the write operation has successfully completed, and the first master replica (the one that has executed the write) reports this to the client.
4. The master replicas propagate the state update to the (marginally trusted) slaves. We assume each slave is registered with one of the masters. Upon receiving a write, each master propagates it to all the slaves that are registered with it. In addition to the state update, each master also sends its slaves a *lease* for the new state. A lease is simply the value of the *stateVersion* variable, time-stamped, and signed by the master. Essentially, a lease gives a slave the permission to serve read requests on a given version of the DSO's state. A lease is valid from the moment it is issued, and it expires after *maxLatency* time. This guarantees that a slave cannot serve read requests that are based on stale state older than *maxLatency*. We provide more details on how leases are used when we discuss the read protocol.

The distinctive feature of the write algorithm described above is the "lazy" propagation of writes (state updates) to slave replicas. Essentially, a slave receives an update only *after* the corresponding write operation has completed (from the point of view of the client that has initiated it). The reason we require this "lazy" state update algorithm, as opposed to having masters *and* slaves participate in some sort of total ordering broadcast, is performance. Since only masters are trusted (not to exhibit Byzantine-faulty behavior), a total ordering broadcast protocol including the slaves would have to be resistant to Byzantine failures, and implementing such an algorithm over a WAN is extremely expensive. "Lazy" state updates make the write protocol much more efficient, but also weaken the consistency model, since a client cannot be guaranteed that once his write is committed it will be seen in all subsequent reads.

4.4.2.3. The read protocol. Read operations are invoked by clients on slave replicas. Since these replicas are only marginally trusted, they may return incorrect results to client requests. In order to leverage this threat, results produced by slave replicas need to be audited; the read protocol is designed to facilitate this process. This protocol is shown in Fig. 5, and consists of four (logical) steps:

1. A client invokes a read method which is protected by means of auditing against Byzantine faults. In the DSO's security policy, this is specified by a rule of the form:

   $Role_A$ **auditedBy** $Role_B$ **canExecute**

   Method **underConditions** *Conditions*

   The user proxy finds a DSO (slave) replica assigned $Role_A$ (using the Globe Location Service – GLS – see Section 4.3.2), connects to it and sends the method invocation request.
2. The slave replica performs the access control check to ensure the user is allowed to invoke *Method* under the DSO's security policy. If this is the case, the slave executes the read request, and sends back the result, together with a *pledge*, a valid *lease* for the version of the DSO state on which the result was computed, as well as the DSO credentials of the master replica that has issued the lease.
   The pledge sent by the replica has the following format:

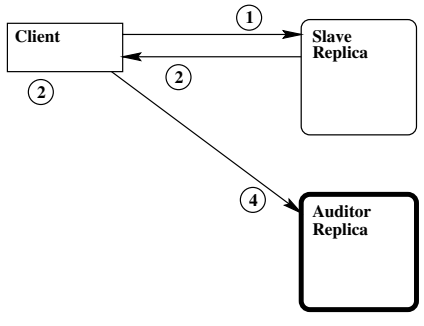

Fig. 5. The read protocol.

Please cite this article in press as: B.C. Popescu et al., Design and implementation of a secure wide-area ..., Comput. Netw. (2007), doi:10.1016/j.comnet.2006.11.008

$\{H(request), H(result), stateVersion, timestamp\}_{y_{slave}}$

where H is a secure hash function, and y_{slave} is the slave replica's private key. The pledge contains a secure hash of the original request, a secure hash of the result, the value of the *stateVersion* variable (as it appears in the lease) and a timestamp, indicating the time when the result was computed by the slave.

3. Upon receiving the data from the slave, the user proxy performs the following checks:
   - It verifies the master replica credentials received from the slave, ensuring that all signatures are correct. If these checks pass, it means that the master replica that has these credentials has indeed the right to issue state version leases to the slave that has produced the read result.
   - It verifies the lease, by checking the master replica's signature on it, and ensuring the timestamp in the lease is less than *maxLatency* old.
   - It verifies the pledge, by checking that the hashes in the pledge match the hash of the original request, and the hash of the result (as sent by the slave), that the *stateVersion* in the pledge matches the one in the lease, and that the timestamp in the pledge is within *maxLatency* from the timestamp in the lease.
4. If all the checks pass, the user proxy accepts the result, and forwards the original request, the lease, the pledge, and master replica credentials to the auditor. As specified by the access control rule (see Step 1), the auditor is a replica assigned $Role_B$, which the proxy can find using the GLS.

The distinctive characteristic of the read protocol is that the client performs a series of checks on the result, the lease, and pledge associated with the result, before accepting them. These tests alone are not sufficient to detect erroneous results; instead their purpose is to ensure that the pledges signed by the slaves can potentially be used as proofs of misbehavior. Essentially, a pledge, constructed as described above, commits the slave replica that has signed it to a unique combination of the DSO's state, read request, and result (for that request). A trusted auditor that has access to the same version of the DSO state can simply re-execute the request; should the hash of the result (as computed by the auditor) not match the one in the pledge, the pledge becomes a self-incriminating proof of the slave's misbehavior. We explain this auditing process in detail next.

4.4.2.4. Auditing. Probably the most important feature of the Byzantine-fault tolerance mechanism we propose is auditing. This is done by trusted auditor replicas; its purpose is to detect potentially erroneous results produced by the (untrusted) slaves, so that corrective action can be taken against them.

Compared to regular (non-auditor) replicas, each auditor replica holds two additional data structures: *the read requests list* (RRL) and the *state updates list* (SUR).

The RRL consists of five fields, corresponding to the information sent by clients at the end of the read protocol (see the previous section): the read request (as issued by the client), the pledge signed by the slave that has computed the result, the lease, the credentials of the master replica that has issued the lease, as well as the local user Id corresponding to the user proxy that has requested the audit. Entries in the RRL are ordered by the value of the *stateVersion* variable, as it appears in the lease.

The SUR consists of state update (write) requests, in the order in which they change the object's state. Essentially, each entry in this list corresponds to a new version of the DSO's state, and describes how the new state can be reached.

The auditor fills these lists with data it receives from clients – (request, pledge, lease, master credentials) tuples – and master replicas – state update messages. The auditor re-executes the read requests in order to check their correctness, while occasionally applying state updates in order to (loosely) keep in sync with the DSO state. The invariant we want to preserve here is that a state transition is performed only *after* all the read requests that depend on the previous state version have been verified by the auditor. One way to achieve this is to introduce some additional delal – *minStateUpdateDelay*, between the time a state update is initiated, and the time the update is applied by the auditor. The purpose of *minStateUpdateDelay* is to ensure that the auditor can move to a new state only *after* all audit requests for possible read requests performed by clients on a previous state have been sent received by the auditor. Remember that our state consistency model allows slaves to compute read results on an old (stale) state of the DSO, but this state cannot be older than *maxLatency* compared to the latest state update (otherwise the client would reject the result after examining the slave's lease –

Please cite this article in press as: B.C. Popescu et al., Design and implementation of a secure wide-area ..., Comput. Netw. (2007), doi:10.1016/j.comnet.2006.11.008

see step 3 of the read protocol). As such, the *minStateUpdateDelay* should be at least equal to *maxLatency*. In practice, *minStateUpdateDelay* should be slightly larger than *maxLatency* in order to offset possible network delays and clock skews between the client and the auditor. In this context, it is important that the client forwards the pledge to the auditor immediately after it has received it from the slave (essentially, sending the pledge to the auditor is integral part of the read operation, and the result of the read is not accepted by the client until the pledge has been successfully send to the auditor). A malicious client may arbitrarily delay sending the pledge to the auditor, but the only outcome of this would be that the client itself runs into the risk of accepting a read result which will not be audited (essentially, a malicious client can only harm itself).

The audit process works as follows:

1. The auditor takes a client read request from the RRL and executes it.
2. The auditor takes the hash of the result, and compares it to the hash in the pledge. If the two hashes match, the result is correct, and the auditor moves to the next request; otherwise, the auditor moves to Step 3 in order to perform additional checks.
3. If the two hashes do not match, the auditor verifies the pledge, lease, and master replica credentials for the audit request, performing all the checks that should have been performed by the client before accepting the result being audited (see previous section). The purpose of these checks is to determine whether the slave has indeed produced an erroneous result, or the client is misbehaving, sending bogus data for auditing. Depending on the outcome of these checks we have two possibilities:
   (a) The pledge, lease, and master replica credentials are all valid. In this case, the client is honest, and the slave replica has indeed produced erroneous results. The auditor proceeds with taking corrective action against the (now proven) malicious slave.
   (b) At least one of the pledge, lease, or master replica credentials is not valid. In this case, the client is acting maliciously, since, according to the read protocol, it should have verified their correctness before requesting the audit. In this case, the auditor needs to take corrective action against the client. Possible options here include rejecting future audit requests from that client, or even revoking the client's DSO credentials, essentially excluding the client from the DSO domain.

The distinctive feature of the audit protocol is that it has been optimized to ensure fast verification of read results. Given our optimistic assumption (that slave misbehavior is infrequent), in normal circumstances this verification only requires the auditor to re-execute the request and compute a hash value (a fast operation). The auditor does not perform any signature or credentials verification, since it is assumed this was done by the client, as part of the read protocol. A malicious client could have omitted these checks, but our assumption is that clients are rational, and are interested in getting correct results.

In order to ensure fast auditing, further optimizations can be introduced: for example, the auditor can keep a small associative cache containing (*read request*, *result hash*) pairs. Before re-executing a request, the auditor can search for it in this cache; if an entry is found, the auditor can simply take the hash of the result in the cache and compare it against the one in the pledge, thus saving the request execution time. This cache needs to be flushed after each state update.

The reason for all these optimizations is to ensure the auditor can keep up with the workload. It is also important to understand that (as the name suggests), the *minStateUpdateDelay* is only the minimum delay the auditor may have to wait before applying a state update. Depending on the number of read requests it needs to audit for a given DSO state, the auditor may have to delay the next state update even longer (i.e. until it has finished auditing all read requests for the current state). This will likely happen if client read requests patterns are bursty, or they exhibit significant rate variations during the day (e.g. many requests during the working hours, quiet periods during the night). In this case, the auditor may significantly "lag behind" with applying state updates during the busy period, but eventually recover (as long as it can keep up with the *average* workload). The bottom line is that the auditor has to perform the same work as (many) slaves, this being the only way we could achieve the economies of scale to justify our Byzantine fault tolerance mechanism. There are a number of solutions for tackling the problem of the auditor not being able to keep up with the workload:

Please cite this article in press as: B.C. Popescu et al., Design and implementation of a secure wide-area ..., Comput. Netw. (2007), doi:10.1016/j.comnet.2006.11.008

- Replicate the auditor. This has the drawback that more trustworthy computing resources are required.
- Perform probabilistic auditing. In this case the auditor only verifies a random subset of all the received audit requests. This is more economical (does not require extra auditors), but has the drawback that it may take longer to detect a malicious slave.

4.4.2.5. Taking corrective action. The last part of the Byzantine fault-tolerance mechanism we propose regards the corrective action that should be taken once a malicious slave is identified. As explained earlier, this happens once the auditor detects a mismatch between the hash of the result of a read request it re-executes, and the hash committed in the pledge by the slave which has originally computed the result.

The first action the auditor needs to take is to revoke the slave replica's local credentials. This essentially excludes the replica from the DSO domain, prevents further client requests from being directed to that replica, and thus prevents further damage. Depending on the contractual agreements between the DSO administrator and the administrator of the GOS hosting the malicious replica, further legal action can be taken. The pledge signed by the slave can be used in courts as irrefutable evidence of its misbehavior.

Preventing further damage is useful, but something may need to be done to fix the damage already caused by the malicious replica. Here there are two alternatives that may be considered.

First, damage can be fixed by reverting the DSO to an old state, not affected by the erroneous results computed by the malicious replica. Although slave replicas only compute read results, depending on the application functionality, it may be possible that a client uses such an (erroneous) read result in a subsequent write operation, in which case the DSO state is (indirectly) compromised by the malicious slave.

Reverting to a safe old state is relatively straightforward to implement, since the auditor is guaranteed to always "lag behind" with state updates. At the moment an erroneous read result is detected, the auditor can simply broadcast a "recovery" state update, reverting the state of all DSO replicas to the version it has at that moment (which is not affected by the erroneous read). Having detailed knowledge of the application functionality may allow the auditor to perform even more "intelligent" recovery, for example only reverting to an old state if it detects write operations that may depend on the erroneous read result.

Although such a recovery strategy may seem appealing, we were not able to identify any realistic application scenarios where this may be useful. The main problem with reverting state is that it does not work well with interactive user applications, where an erroneous read result may cause the user to perform some irreversible action (printing a file, making an electronic payment, etc.). Reverting state is effective in case of applications doing (distributed) batch processing; in this case, reverting only requires some of the batched operations to be re-executed. However, as explained earlier, one motivation for having an audit-based Byzantine fault-tolerance mechanism is reducing response latencies, by executing user requests on (marginally trusted) replicas close to the user. In case of batch processing, response latency is much less of a concern, and in this case alternative Byzantine fault-tolerance mechanisms (based on state machine replication, for example) may be more effective.

As a conclusion, taking corrective action by reverting to an old state is definitely possible with our scheme, but we do not expect this correction mechanism to suit well the kind of applications to which our scheme can be applied.

The second damage-repair mechanism that can be applied is actually extremely simple: revoke the malicious replica from the DSO, and simply ignore whatever damage it might have caused. Ignoring damage may seem a strange way of fixing it, but nevertheless, we were able to identify a rather wide class of applications where this may suffice. We elaborate more on this next.

4.4.2.6. Discussion. The audit-based approach to Byzantine fault-tolerance described in this section was first introduced by us in [63]. We view it as an economical alternative to state machine replication, since the amount of resources needed to serve an audit-based execution request is smaller than for replicated execution; furthermore, in case of replicated execution, the user-perceived latency is dictated by the slowest replica in the composite execution target, while for traceable execution the user can select a "fast" (low-latency) replica, and let the result be audited later. However, we must point out that the BFT scheme described in this section requires a DSO employing public key authentication mechanisms, which in itself (as we will see in

Please cite this article in press as: B.C. Popescu et al., Design and implementation of a secure wide-area ..., Comput. Netw. (2007), doi:10.1016/j.comnet.2006.11.008

Section 5) may have a significant impact on the user-perceived latency.

One class of applications which may be particularly well-suited for this audit-based Byzantine fault tolerance technique consists of applications that provide information aggregation and manipulation – search engines, news aggregators (Slashdot-like), and online databases. In this case, it may be possible to filter erroneous results by means of direct inspection (e.g. human users ignore data that is incomprehensible, or simply too far-fetched); furthermore, individual information items may be protected by means of state signing (for example all news articles are signed before publication). In this case, only aggregate query results may be (maliciously) altered by rogue replicas. As such, rogue replicas can at most provide incomplete results to queries, which amounts more or less to denial of service. In this situation, simply removing malicious replicas after detection may be reasonable.

In the Globe model writes are handled by master replicas and reads by slaves. The model can be easily extended to allow masters to serve both reads and writes, for example for users or applications that require 0-staleness. This extension should be applied only to a limited number of privileged users or actions or a specific application. If generalized this extension conflicts with one of the main goal of Globe that is the ability to offload work from trusted (expensive) master replicas to un-trusted (cheaper) slaves.

4.5. Platform security

In order to bring computation close to clients, DSO replicas are often instantiated on GOSes controlled by third parties, similar to the grid distributed computing model [35]. Trusted or untrusted GOS administrators are free to choose any policy for hosting replicas; however, it is essential that all hosted replica receive a ''fair'' treatment, in the sense that they correctly share the resources provided by the GOS. This is far from a trivial requirement, given the fact that replicas are instantiated using mobile code, which can be used to launch all sorts of attacks on the hosting platform (corrupt storage, interfere with other replicas, waste memory and CPU cycles, and launch all sorts of DoS attacks). There are two general solutions for the problem: sandboxing and code signing.

Sandboxing aims at creating an isolated execution environment for each hosted replica, and ensuring all interactions outside the sandbox are tightly controlled by the GOS. We emphasize that the focus of this work is not designing new sandboxing tools, but rather using existing one. Since the Globe middleware is primarily implemented in Java, we decided to use the Java sandboxing mechanisms and implemented custom sandboxing of untrusted local representatives, which still allows replicas (controlled) access to persistent storage. Another issue we are concerned with is preventing hosted replicas from launching distributed DoS attacks, by creating network connections to arbitrary hosts on the Internet. To prevent this, we only allow replicas to connect to a limited numbers of replicas *of the same DSO* (in order to propagate state updates). In the future, we plan to complement these sandboxing mechanisms with a Java resource-management system such as JSeal [74] to also constrain memory allocation and CPU usage. For different Globe middleware implementations (in C/C++ for instance), use of alternative sandboxing tools, such as Janus [40], should be also possible.

While sandboxing allows GOS administrators to restrict the amount of computing resources a given DSO replica can use, to decide on the actual resource limits, we use code signing. Essentially, the mobile code used to instantiate a DSO replica is signed with the DSO's key, so that GOS administrators can then set resource limits on a per-DSO basis, limits which are stored in a GOS resource management configuration file. It is up to the administrator to decide on the resource allocation policy: this can be either absolutely egalitarian (all replicas receive the same fraction of the available resources), based on external DSO certification (DSO owned by certain organizations receive more resources), or even based on pay-per-use economic models.

5. Performance evaluation

To measure Globe's performance we performed two experiment series. In the first series we focus on security overhead during client method invocation on a DSO replica. We break down this method invocation process into (logical) stages (for example the client finding the replica, the client contacting the replica, authentication, etc.) and we measure the duration of each stage in various security settings. Based on these measurements we calculate the total overhead on the client and replica side, as well as the (theoretical) replica throughput (i.e. how many

Please cite this article in press as: B.C. Popescu et al., Design and implementation of a secure wide-area ..., Comput. Netw. (2007), doi:10.1016/j.comnet.2006.11.008

invocations per second can the replica handle). We then validate this theoretical throughput against the real one, which we measure by having multiple clients concurrently issuing requests. The second experiment series consists of a number of (synthetic) micro-benchmarks, stressing different parts of the replica hosting system – the CPU, disk and network stack. The purpose of these micro-benchmarks is to measure the security overhead by comparing the real throughput of a replica accessed by concurrent clients in various security settings.

For our experiments we only used the coarse-grain (bitmap-based) access control framework described in Section 4.3.1. Due to time and human resources (programmers) limitations, we were not able to integrate fine-grain access control and Byzantine fault tolerance mechanisms with our secure Globe prototype. This is a limitation of our approach, but we hope to address it as part of future work.

The purpose of our experiments is to compare the overhead introduced by our security architecture. For this we considered four implementation scenarios: a DSO with all security mechanisms disabled (marked *Plain* in our graphs), and three security-enabled implementations.

For the security-enabled DSO, we consider three different authentication mechanisms:

- The TLS protocol for public key authentication using the PureTLS library [3] (marked *PureTLS* in our graphs).
- Our own implementation of the TLS protocol (marked *OwnTLS* in our graphs).
- Our implementation of the symmetric key authentication protocol described in Section 4.2.2 (marked *Symmetric* in our graphs). In this case, the *RKL* size is set to 100 and the *UKL* size is set to 10,000.

Our TLS implementation uses the latest Java cryptographic extensions (JCE); the reason we decided to develop an alternative TLS implementation is that at least at the time we were working on the Globe prototype, there was no off-the-shelf TLS library based on Sun's Java Cryptography Extension Package (PureTLS uses the Bouncy Castle Java cryptographic library [6], which is not optimal from a performance point of view). The performance numbers associated with our own TLS implementation can be seen as a best case scenario for public key authentication mechanisms; however, our TLS library has been developed for research purposes only, and has not been hardened through years of deployment and public scrutiny, so for implementing actual Globe applications we recommend using off-the-shelf authentication components.

Table 2
Experimental setup

Replica host
PIII 933 MHz, 2 GB memory, 100 Mb/s Ethernet, Adaptec 29160 Ultra-160 SCSI controller Seagate 73 GB 10.000 rpm disk, RedHat 7.2, Custom configured 2.4.9 kernel
Client proxy hosts
PIII 1 GHz, 1 GB memory, RedHat 7.2, Custom configured 2.4.19-pre10 kernel

The configuration for the machines hosting the replica and the clients is shown in Table 2. On all hosts the Globe middleware runs on top of the Java 2 runtime version 1.4.2_02.

The replica hosts are located at the Vrije Universiteit in Amsterdam, while the client proxy hosts are part of the DAS-2 distributed grid [15], consisting of five LAN clusters located at five Dutch universities. Each of these clusters use a Myrinet [2] multigigabit LAN for local communication, while the wide-area connectivity is provided by SurfNet – the Dutch university Internet backbone [5]. We believe this setting, where WAN interactions are "localized" (clients and replicas are in close network proximity), realistically simulates deployment conditions for Globe applications (as explained in Section 2, Globe DSOs have the ability to replicate in order to bring computation close to their clients).

5.1. Work breakdown and maximum throughput

The purpose of the first experiment series is to evaluate the security overhead during the normal operation of a Globe DSO, namely during the processing of client requests. For this series we use the *Integer* DSO under different security settings. The *Integer* DSO is a distributed implementation of the *Integer* Java class, using a master-slave replication algorithm; as such, this is probably one of the simplest possible Globe objects.

In this case, a (master) replica of the object is instantiated on the replica host. We instantiate a number of clients (user proxies) on the client proxy hosts. The user proxies bind to the *Integer DSO* master replica, and issue a write request (*Integer.setValue*()). Given this setup, we perform two experiments, one to obtain the detailed workload

Please cite this article in press as: B.C. Popescu et al., Design and implementation of a secure wide-area ..., Comput. Netw. (2007), doi:10.1016/j.comnet.2006.11.008

breakdown, and the other to measure the maximum replica throughput.

5.1.1. Experiment 1

The purpose of this experiment is to obtain a detailed breakdown of replica/proxy workload during regular DSO operation. For this experiment, the object ID for the DSO being accessed is cached by the client (thus, there is no need for a GNS lookup). The client is also already registered with the object and has cached its local DSO credentials, as well as the proxy blueprint. Fig. 6 shows the stages involved during a client request on a DSO replica:

1. The client instantiates the proxy for the DSO. When security is enabled, the client initializes the proxy's security subobject with the (cached) client authentication credentials.
2. The proxy queries the GLS for a replica allowed to execute *Integer.setValue*().
3. The proxy connects to the contact point returned by the GLS.
4. If security is enabled, the proxy and the replica mutually authenticate and establish a secure channel.
5. The proxy sends the *Integer.setValue*() request to the replica (over the secure channel established at Stage 4 if security is enabled, or over the regular TCP channel established at Stage 3 if not). The replica executes the request and sends the result back to the proxy.

We repeat the *Integer.setValue*() operation 500 times for different security settings and measure the average duration for each of the five stages. The results are shown in Fig. 7.

We can see that authentication and secure channel establishment is by far the most expensive stage

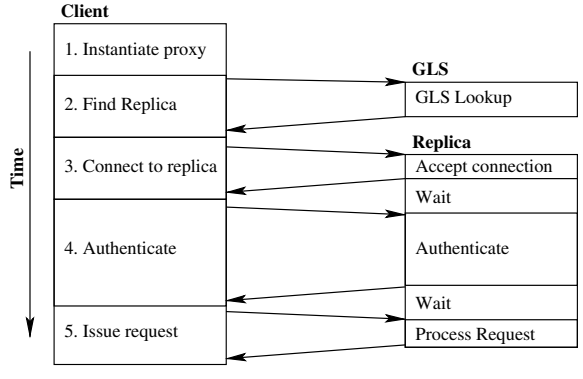

Fig. 6. Stages involved in a client request.

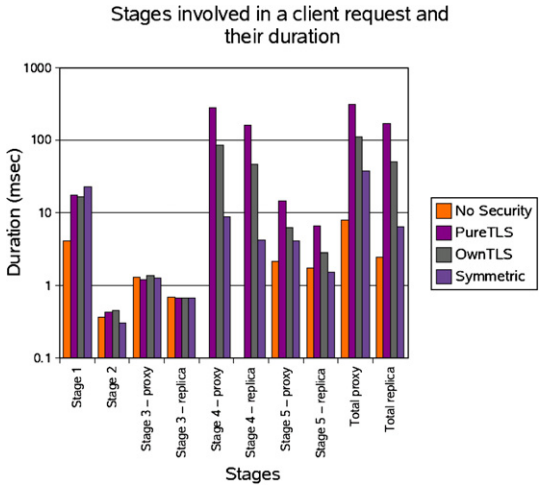

Fig. 7. Stages involved in a client request and their duration (*Y*-axis on logarithmic scale). The "Total proxy" values correspond to the client-perceived latency for each setting.

during regular DSO operation. Public key authentication is particularly expensive. When using the *PureTLS* library (which, as already mentioned, is not particularly efficient), authentication alone increases the client-perceived latency by almost 300 ms. Even when using our own (optimized) TLS implementation, authentication still accounts for almost 100 ms of the client-perceived latency. The good news is that using the symmetric key authentication protocol introduced in Section 4.2.2 can dramatically improve performance. In this case, the authentication-introduced latency is an order of magnitude smaller compared to compared to public key authentication mechanisms.

Another measurement of interest is the total amount of CPU time used by the replica when serving one client request. Having this, essentially allows us to compute the maximum (theoretical) throughput for the replica (e.g. how many client requests per second can the replica serve). Fig. 8 shows the results for this measurement:

From Fig. 8 we can see that using the symmetric key authentication protocol introduced in Section 4.2.2 can significantly increase the maximum replica throughput. In the next section we will validate the numbers in Fig. 8 by having the replica serve concurrent client requests.

5.1.2. Experiment 2

The purpose of this second experiment is to validate the theoretical throughput results from Exper-

Security features enabled	Total replica CPU time (msec)	Max. theoretical throughput (req/sec)
No security	2.47	404.85
PureTLS	169.17	5.91
OwnTLS	50.05	19.98
Symmetric	6.49	154.08

Fig. 8. Replica CPU time spent per request and maximum replica throughput.

iment 1. The idea is to have concurrent clients continuously sending requests to an *IntegerDSO* replica. We want to measure the maximum throughput the replica can sustain, and compare this to the (theoretical) values derived in the previous section. We run different numbers (ranging from 1 to 30) of concurrent clients; these clients continuously send *Integer.setValue*() requests to a master replica for 500 iterations. We measure the amount of time necessary to complete one request. Based on this, we then calculate the "real" replica throughput. The results are shown in Fig. 9.

The results of this experiment validate the maximum theoretical throughput values derived in Experiment 1. We can see from Fig. 9, that with only one client continuously sending request, the replica throughput is considerably lower than the maximum throughput achievable; the reason for this is that the replica sits idle during the time the (sole) client is doing processing, and during the time requests/results are transfered over the network. The replica throughput rapidly increases as more clients join in with sending requests; in this case, the previously idle replica time is used to serve concurrent requests. The maximum throughput is achieved with somewhere between 5 and 10 concurrent clients. Above this threshold, the overhead associated with multithreading (each client request is served as a separate thread) begins to affect the replica performance (basically some CPU cycles are used on multithreading instead of useful work).

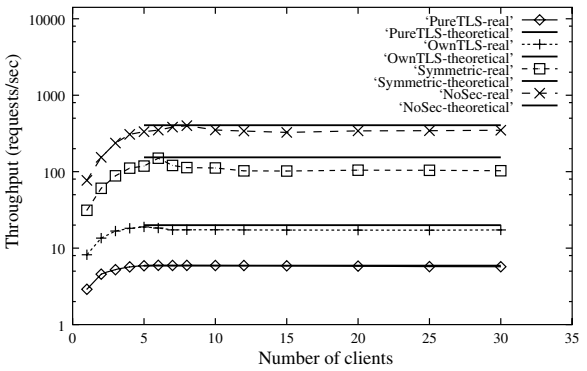

Fig. 9. Measured throughput for various security settings. The bold lines represent the theoretical throughput derived from Experiment 1. The *Y*-axis of the graph is on logarithmic scale.

In general, these results show that public key authentication mechanisms are quite heavyweight. In the case of *Pure TLS*, about six concurrent clients are enough to saturate the replica. Even our optimized TLS implementation can at most handle about twenty concurrent clients before the replica becomes saturated. We view this as an inherent limitation of public key authentication; not surprisingly, Web servers that handle SSL/TLS connections are typically equipped with hardware cryptographic accelerators. On the other hand, symmetric key authentication is relatively lightweight.

5.2. Microbenchmarks

The purpose of this second experiment series is to evaluate the security overhead through a series of (synthetic) benchmarks. For each benchmark, each client repeatedly executes a transaction for many iterations (typically 500). At the end, we measure how many transactions per second the replica can sustain for different numbers of concurrent clients, different security configurations and different workloads. Each transaction consists of the following steps:

- *bind*() – In this step the client downloads the object proxy code and instantiates its object local representative, then finds the replica, connects to it, and runs the mutual authentication protocol.
- *performOperation*() – In this step, the client issues the actual method invocation request. For our benchmarks, the operations are artificial workloads stressing different parts of the replica hosting system – the CPU, disk and network stack.
- *unbind*() – The client disconnects.

Each benchmark simulates a different type of workload – disk workload, cpu workload and network workload. For each type of workload we considered the "light" and "heavy" case as follows:

- disk access workload: "light" – 1 disk access/trans.; "heavy" – 100 disk accesses/trans.

- CPU workload: "light' – 10 ms CPU time/trans.; "heavy" – 300 ms CPU time/trans.
- data transfer workload: "light" – 10 KB network data transfer/trans.; "heavy" – 1 MB network data transfer/trans.

In addition to this we also considered the "empty" benchmark which consisted of an "empty" transaction – basically the client binds to a replica and then immediately unbinds without issuing any request.

One point we need to stress here is that for the TLS performance numbers we have disabled session caching, so each iteration requires a complete handshake. We believe such a setting is more realistic for the usage scenarios we envision for Globe applications because of the following reasons:

- There is a limit on how long a TLS connection can be cached. Ref. [30] suggests this limit to be set to at most 24 h. There are certain types of potential Globe applications – electronic newspapers and e-commerce applications for example – with a usage pattern consisting of periodic, but less frequent client requests (in the order of days rather than hours). For such applications TLS session caching would have no impact on performance.
- One of the great advantages of dynamic application replication with Globe is the ability to deal with "flash crowd" events. When such events occur, Globe objects have the ability to quickly react, and instantiate new replicas to handle the peak workload. However, a new replica comes without any cached TLS sessions; again, in this case TLS session caching has no impact on performance.
- In Globe, replica selection for method invocation is done by the replication subobject of the client proxy, based on the replica contact points returned by the Globe Location Service. In the simplest case, the client proxy may always select the closest object replica. However, for certain types of Globe applications, especially those involving extensive computational workload (service computing and Grid applications for example), the replica selection algorithm needs to take into account additional factors besides network proximity, such as balancing workload among replicas. Since replica workload is dynamic, as clients come and go, it is less likely that any given client will interact with the same replica over a number of sessions. Furthermore, a client may have to switch replicas when invoking different methods, based on the reverse access control settings for the object. In all these scenarios, caching TLS connections will have a much reduced impact on the overall performance.

Fig. 10 shows the performance results for "empty" transactions. Although this is not a very realistic workload, it gives an insight of the actual cost of security mechanisms (authentication in particular). We can see that when security mechanisms are disabled, Globe performs several times better than the most efficient security implementation. On the other hand, using the symmetric key authentication module is an order of magnitude more efficient compared to public key authentication, which proves the protocol we introduced in Section 4.2.2 can be extremely useful for securing Globe applications.

Figs. 11–16 show the detailed performance results for the other types of workload considered. We also provide a summary, "head-to-head" comparison of the maximum achievable throughput for all workloads in Figs. 17 and 18. From these figures, we can see that in the case of "light" transactions, security adds a significant overhead. On the other hand, this is also where the symmetric key authentication protocol we propose shows the most promising results, since it performs several times better than public key authentication schemes.

In the case of workloads consisting of "heavy" transactions, the performance penalty introduced by security mechanisms is much lower (in relative terms), since a larger fraction of the computing resources used per transaction are spent for doing the actual work. It is worth noting that for two types

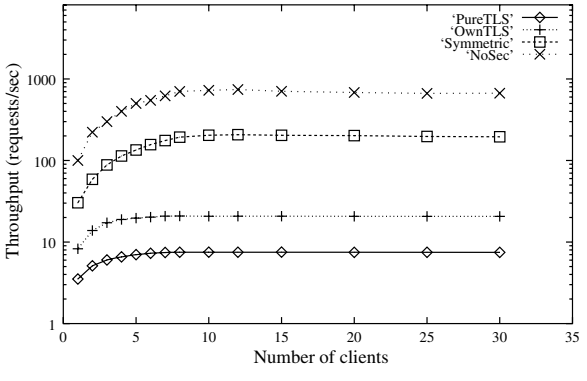

Fig. 10. GOS throughput under "empty" transactions workload.

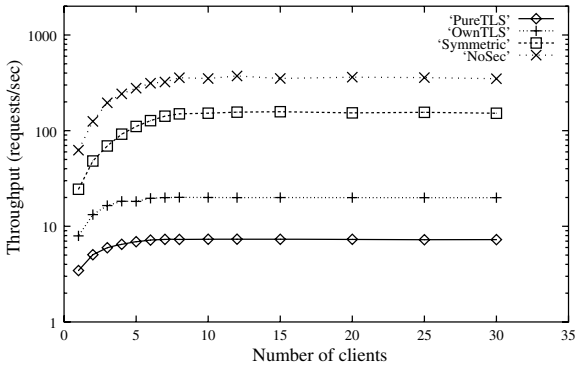

Fig. 11. GOS throughput under light disk access workload.

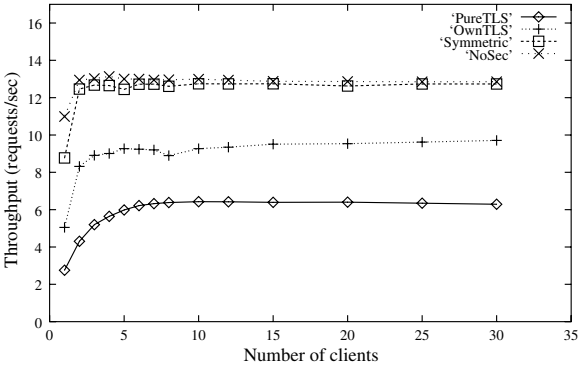

Fig. 12. GOS throughput under heavy disk access workload.

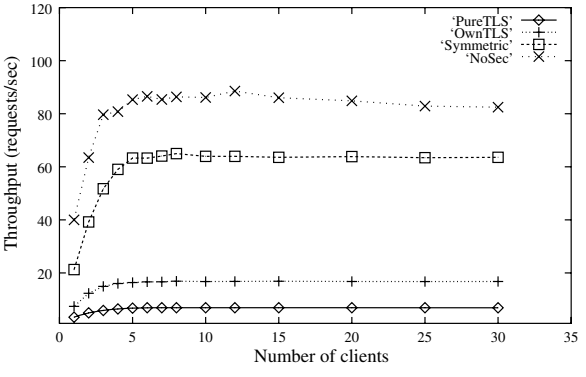

Fig. 13. GOS throughput under light CPU workload.

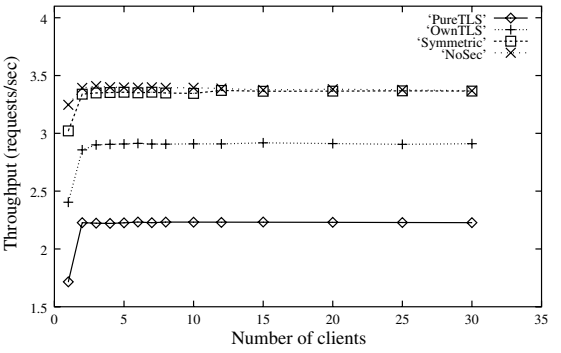

Fig. 14. GOS throughput under heavy CPU workload.

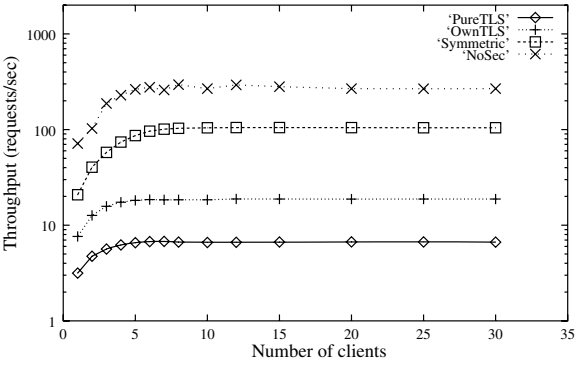

Fig. 15. GOS throughput under light network workload.

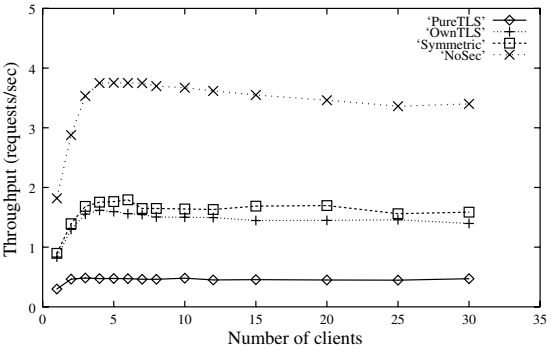

Fig. 16. GOS throughput under heavy network workload.

of workload (disk intensive and CPU intensive), the performance gap between the ''no security'' Globe implementation and the one employing the symmetric key authentication module is extremely small. In the case of network intensive workload, the efficient symmetric key authentication mechanism is offset by the overhead of encrypting a large amount of data (1 MB), which is done for all secure Globe implementations, but is disabled in the non-secure version.

5.3. Discussion

We believe the results of these experiments validate our security design. Although security introduces a significant penalty, it is by no means

Please cite this article in press as: B.C. Popescu et al., Design and implementation of a secure wide-area ..., Comput. Netw. (2007), doi:10.1016/j.comnet.2006.11.008

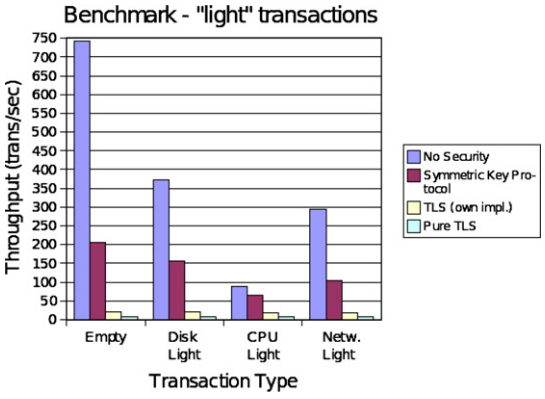

Fig. 17. GOS maximum throughput for "light" transactions.

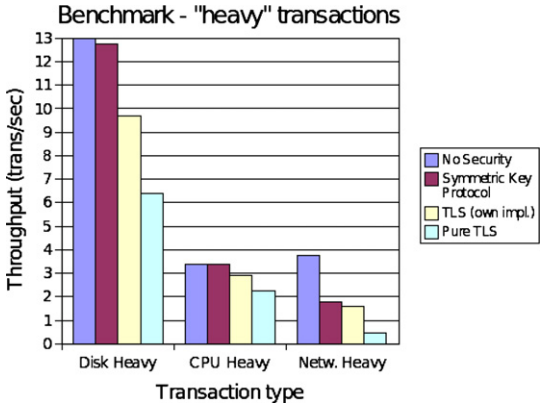

Fig. 18. GOS maximum throughput for "heavy" transactions.

prohibitive; most importantly, our design allows application developers to mediate between security and performance by choosing the modules that best fit their needs. For applications where lightweight transactions are the norm, we believe that our symmetric key authentication protocol can be a great advantage. For applications where a more heavyweight workload is to be expected, decision on whether to use symmetric or public key authentication will likely be determined by additional factors (besides performance), such as whether non-repudiation is an issue, or a highly dynamic (and unpredictable) user population needs to be accommodated.

It is also important to understand that for our experiments we did not consider the impact of replication to the overall performance. For our experiments we decided to have replicas and clients in relatively close network proximity, which would correspond to a "smart" replication strategy. In general, the impact of replication strategies on the overall performance is orthogonal to the problem we addressed in this section, namely quantifying the security overhead (i.e. the security overhead is always present, no matter how "smart" or "dumb" replica placement is). A very comprehensive analysis of the impact of replication on performance in the Globe context is presented in [61].

6. Related work

In the past decade, distributed systems security has received considerable attention. One of the most comprehensive security models is the one designed for CORBA [9]. 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. The CORBA model does not deal at all with mobile code, and has little support for interdomain security. Furthermore, mechanisms for Byzantine fault-tolerance (BFT) are not part of the original design, although various approaches for integrating such mechanisms with the overall CORBA architecture have been proposed [20,49,56,29,39] (Ref. [55] provides a good overview of all these efforts). In particular, the Immune System [56] provides an elegant solution by using an *interception strategy* for integrating BFT mechanisms to CORBA. In the Immune System, BFT mechanisms are based on *object groups* – essentially sets of replicas of the same basic CORBA object placed on different hosts. In order to ensure state consistency among replicas, Immune uses active replication. Each request is transparently routed to all replicas of a object group; results from all replicas are collected and subject to a majority voting in order to detect Byzantine faults. The strongest feature of the Immune system is that it integrates BFT mechanisms in a transparent way, by placing the replication/multicast layers between the ORB and the operating system. In this way unmodified CORBA client and server objects can be enhanced with BFT features, while running on an unmodified ORB.

The interception strategy allows the Immune System to support nested method invocations, unlike Globe. We do not support nested method invocations because there is no scalable and transparent

Please cite this article in press as: B.C. Popescu et al., Design and implementation of a secure wide-area ..., Comput. Netw. (2007), doi:10.1016/j.comnet.2006.11.008

mechanism to handle the problems that arise when a replicated object invokes the method of another object [13]. The solutions employed by the Immune System and similar efforts depend on resilient group communication protocols to coordinate the actions of the caller replicas (and callee replicas if the called object is also replicated). Such protocols are however very expensive over a wide-area network. Globe does support conceptual nesting of objects, as object identifiers can be stored in the state of a (parent) object.

What makes the Java [41] 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. There are a number of object-based distributed architectures implemented on top of Java [34,16,43,33]. Similar to us, [43] addresses some of the security issues derived from service replication, but their work focuses more on defining a policy language for expressing global security policies rather than providing a unified security architecture. Ref. [33] presents a comprehensive security architecture with provisions for trust management, mutual client-service authentication, and user access control, but does not address reverse access control and Byzantine fault tolerance issues.

One project that is similar in vision to Globe is WebOS [70]. The idea is to provide operating system services to wide-area applications, including mechanisms for naming, persistent storage, remote execution, authentication and security. CRISIS [18] is the security architecture for WebOS. Like the Globe security architecture, CRISIS combines a wide range of security primitives in order to provide a comprehensive protection architecture, with provisions for delegation of privileges, hierarchical trust, authorization and a secure time service. Unlike Globe, CRISIS relies entirely on public key certificates for authorization and delegation; this, combined with the proposed revocation mechanism that requires frequent certificate counter-signing by an online trusted authority, makes their design more heavyweight than ours. Furthermore, CRISIS does not directly address the issue Byzantine fault-tolerance, and does not provide any mechanisms for dealing with partially trusted application servers.

There are a number of projects that deal with security problems that arise in metacomputing environments. 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. Legion [77] 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 introduces a more high-level security design, stressing flexibility and extensibility, but less architecture and protocols.

There are also a number of projects that specifically deal with Byzantine fault tolerance in the context of replicated services. Phalanx [50] is a software system for building persistent services that support shared data abstractions, such as public key infrastructures and e-voting. Ref. [26] describes a toolkit for building Byzantine fault tolerant systems. Like our design, [26] stresses the importance of efficient symmetric key protocols for authentication over wide-area networks, but their key distribution scheme is less flexible than the one we propose. Furthermore, none of these systems addresses reverse access control and platform security issues.

Finally, in the past few years there has been an explosion of peer-to-peer (P2P) applications that have sprung out either as academic projects (SETI@home [10], Publius [75]), or as freeware tools to facilitate media exchange (Kazaa 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 [47], content can be integrity-checked by clients, whereas other systems concentrate on securing remote untrusted storage [38] or content traceability [14]. Other interesting attempts to provide security and integrity for P2P systems are described in Refs. [28,25]. However, many of these systems put emphasis on immutable files, which may severely restrict the area of possible applications.

7. Conclusion

In this paper we have described the design and implementation of a security architecture for Globe,

Please cite this article in press as: B.C. Popescu et al., Design and implementation of a secure wide-area ..., Comput. Netw. (2007), doi:10.1016/j.comnet.2006.11.008

a wide-area object-based middleware. Our design addresses a broad range of security threats, and does not rely on any centralized authority that would limit the scalability of the system. Our security architecture consists of a number of modules, separated through standard interfaces, each of them handling a particular set of security issues (trust management, authentication, access control, Byzantine fault tolerance, and platform security). This allows developers to easily replace any of these modules with alternative implementations, better suited for specific Globe applications.

As a result of this work, we have identified a series of new security problems that are specific to replicated services. These include reverse access control for restricting replica privileges with respect to method execution, and protection of distributed objects against malicious hosts running instances of their code. Another significant contribution is a novel symmetric key authentication protocol, which combines some of the advantages of public key authentication schemes (in particular, their reliance on an *offline* TTP), with the computational efficiency specific to symmetric key cryptographic primitives.

As for future work, we plan to integrate the JSeal [74] toolkit with our Globe implementation in order to allow fine grain management of object server resources. We are also considering implementing additional security modules for allowing more flexible Byzantine fault tolerance, based on state machine replication and auditing, as suggested in [26,63].

References

[1] Advanced Encryption Standard, FIPS 197, NIST, US Dept. of Commerce, Washington, DC, November 2001.
[2] Myricom web site: http://www.myri.com/.
[3] "PureTLS" web site: http://www.rtfm.com/puretls/.
[4] Secure Hash Standard, FIPS 180-1, Secure Hash Standard, NIST, US Dept. of Commerce, Washington, DC, April 1995.
[5] SurfNet web site: http://www.surfnet.nl/.
[6] "The Legion of the Bouncy Castle" web site: http://www.bouncycastle.org/.
[7] Trusted Computing Platform Alliance, TCPA main specification v. 1.1b, http://www.trustedcomputing.org/.
[8] The Common Object Request Broker: Architecture and Specification, revision 2.6, www.omg.org, October 2000, OMG Document formal/01-12-01.
[9] CORBA Security Service Specification, Version 1.7, www.omg.org, March 2001, Document Formal/01-03-08.
[10] D. Anderson, Peer-to-Peer: Harnessing the Power of Disruptive Technologies, O'Reilly & Associates, Sebastopol, CA 95472, July 2001 (Chapter 5).
[11] W. Arbaugh, D. Farber, J. Smith, A secure and reliable bootstrap architecture, in: Proc. 18th IEEE Symp. on Security and Privacy, May 1997, pp. 65–71.
[12] T. Aura, P. Nikander, J. Leiwo, DOS-resistant authentication with client puzzles, in: Proc. 8th Cambridge International Workshop on Security Protocols, 2000, pp. 170–177.
[13] A. Bakker, M. Van Steen, A. Tanenbaum, Replicated invocations in wide-area systems, in: Proc. of the 8th ACM SIGOPS European Workshop, September 1998.
[14] A. Bakker, M. van Steen, 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.
[15] H.E. Bal, R. Bhoedjang, R.F.H. Hofman, C.J.H. Jacobs, T. Kielmann, J. Maassen, R. van Nieuwenpoort, J. Romain, L. Renambot, T. Rühl, R. Veldema, K. Verstoep, A. Baggio, G. Ballintijn, I. Kuz, G. Pierre, M. van Steen, A.S. Tanenbaum, G. Doornbos, D. Germans, H.J.W. Spoelder, E.J. Baerends, S.J.A. van Gisbergen, H. Afsermanseh, G.D. van Albada, A. Belloum, D. Dubbeldam, Z.W. Hendrikse, L.O. Hertzberger, A.G. Hoekstra, K. Iskra, D. Kandhai, D. Koelma, F. van der Linden, B.J. Overeinder, P.M.A. Sloot, P. Spinnato, D.H.J. Epema, A.J.C. van Gemund, P. Jonker, A. Radulescu, K. van Reeuwijk, H.J. Sips, P.M.W. Knijnenburg, M.S. Lew, F. Sluiter, L. Wolters, H. Blom, C. de Laat, The distributed ASCI supercomputer project, Operat. Syst. Rev. 34 (4) (2000) 76–96.
[16] D. Balfanz, L. Gong, Experience with secure multi-processing in Java, in: Proc. 18th IEEE Intl. Conf. on Distributed Computing Systems, May 1998, pp. 398–405.
[17] G. Ballintijn, M. van Steen, A.S. Tanenbaum, Scalable user-friendly resource names, IEEE Internet Comput. 5 (5) (2001) 20–27.
[18] E. Belani, A. Vahdat, T. Anderson, M. Dahlin, The CRISIS wide area security architecture, in: Proc. 7th USENIX Security Symposium, 1998, pp. 15–30.
[19] K. Birman, A. Schiper, P. Stephenson, Lightweight casual and atomic group multicast, ACM Trans. Comput. Syst. 9 (3) (1991) 272–314.
[20] K.P. Birman, R.V. Renesse (Eds.), Reliable Distributed Computing with the Isis Toolkit, Wiley–IEEE Computer Society Press, 1994.
[21] M. Bishop, Computer Security: Art and Science, Addison-Wesley, 2002.
[22] C. Boyd, A class of flexible and efficient key management protocols, in: Proc. 9th IEEE Computer Security Foundation Workshop, 1996.
[23] Bruno Crispo, Bogdan C. Popescu, Andrew S. Tanenbaum, Symmetric key authentication services revisited, in: Information Security and Privacy: 9th Australasian Conference, July 2004, pp. 248–261.
[24] M. Burrows, M. Abadi, R.M. Needham, A logic of authentication, ACM Trans. Comput. Syst. 8 (1) (1990) 18–36.
[25] M. Castro, P. Druschel, A.J. Ganesh, A.I.T. Rowstron, D.S. Wallach, Secure routing for structured peer-to-peer overlay networks, in: Proc. 5th USENIX Symposium on Operating System Design and Implementation, December 2002.
[26] M. Castro, B. Liskov, Practical byzantine fault tolerance, in: Proc. 3rd USENIX Symposium on Operating Systems Design and Implementation, February 1999, pp. 173–186.

Please cite this article in press as: B.C. Popescu et al., Design and implementation of a secure wide-area ..., Comput. Netw. (2007), doi:10.1016/j.comnet.2006.11.008

[27] T. Chandra, S. Toueg, Unreliable failure detectors for reliable distributed systems, J. ACM 43 (2) (1996) 225–267.

[28] F. Cornelli, E. Damiani, S.D.C. di Vimercati, S. Paraboschi, P. Samarati, Choosing reputable servents in a P2P Network, in: Proc. of the Eleventh Int'l WWW Conference, Honolulu, HI, May 2002.

[29] M. Cukier, J. Ren, C. Sabnis, D. Henke, J. Pistole, W. Sanders, D. Bakken, M. Berman, D. Karr, R. Schantz, AQuA: An adaptive architecture that provides dependable distributed objects, in: Proc. 17th Symp. on Reliable Distrib. Syst., October 1998, pp. 245–253.

[30] T. Dierks, C. Allen, The TLS Protocol Version 1.0, IETF RFC 2246, January 1999.

[31] J. Dyer, M. Lindemann, R. Perez, R. Sailer, S. Smith, L. van Doorn, S. Weingart, Building the IBM 4758 secure coprocessor, IEEE Comput. 34 (2001) 57–66.

[32] G. Eddon, H. Eddon, Inside Distributed COM, Microsoft Press, Redmond, WA, 1998.

[33] P. Eronen, P. Nikander, Decentralized Jini security, in: Proc. 8th Network and Distributed System Security Symposium, February 2001.

[34] M. Fleury, F. Reverbel, The JBoss extensible server, in: Proc. Middleware 2003, Hudson River Valley, NY, June 2000, pp. 344–373.

[35] I. Foster, C. Kesselman, S. Tuecke, The anatomy of the grid: enabling scalable virtual organizations, Int. J. Supercomput. Appl. 15 (3) (2001).

[36] K. Fu, M.F. Kaashoek, D. Mazieres, Fast and secure distributed read-only file system, Comput. Syst. 20 (1) (2002) 1–24.

[37] T. Garfinkel, B. Pfaff, J. Chow, M. Rosenblum, D. Boneh, Terra: a virtual machine-based platform for trusted computing, in: Proc. 19th ACM Symp. on Operating Systems Principles, 2003, pp. 193–206.

[38] E. Goh, H. Shacham, N. Modadugu, D. Boneh, SiRiUS: Securing remote untrusted storage, in: Proc. 10th Network and Distributed System Security Symposium, February 2003.

[39] A. Gokhale, B. Natarajan, D.C. Schmidt, S. Yajnik, DOORS: Towards high-performance fault-tolerant CORBA, in: Proc. 2nd Intl. Symp. on Distributed Objects and Applications, 2000, pp. 39–48.

[40] I. Goldberg, D. Wagner, R. Thomas, E.A. Brewer, A secure environment for untrusted helper applications, in: Proc. 6th Usenix Security Symposium, San Jose, CA, 1996.

[41] L. Gong, Inside Java 2 Platform Security, Addison-Wesley, Palo Alto, CA 94303, 1999.

[42] R. Housley, W. Ford, W. Polk, D. Solo, Internet X.509 Public Key Infrastructure: Certificate and CRL Profile, RFC 2459, http://www.ietf.org/rfc/rfc2459.txt, January 1999.

[43] S. Ioannidis, S. Bellovin, J. Ioannidis, Design and implementation of virtual private services, in: Proc. 12th IEEE Intl. Workshops on Enabling Technologies, June 2003, pp. 269–274.

[44] M. Kaashoek, A. Tanenbaum, An evaluation of the Amoeba group communication system, in: Proc. 16th Intl. Conference on Distributed Computing Systems, 1996, pp. 436–447.

[45] J. Kohl, B. Neuman, The Kerberos Network Authentication Service (Version 5), Technical report, IETF Network Working Group, Internet Request for Comments RFC-1510, 1993.

[46] R. Kotla, M. Dahlin, High throughput byzantine fault tolerance, in: Proc. 2004 Intl. Conf. on Dependable Systems and Networks, June 2004.

[47] J. Kubiatowicz, D. Bindel, Y. Chen, P. Eaton, D. Geels, R. Gummadi, S. Rhea, H. Weatherspoon, W. Weimer, C. Wells, B. Zhao, OceanStore: An architecture for global-scale persistent storage, in: Proc. 9th ACM ASPLOS, Cambridge, MA, November 2000, ACM, pp. 190–201.

[48] I. Kuz, M. van Steen, H. Sips, The globe infrastructure directory service, Comput. Commun. 25 (9) (2002) 835–845.

[49] S. Maffeis, Adding group communication and fault-tolerance to CORBA, in: Proc. USENIX Conf. on Object-Oriented Technologies, June 1995.

[50] D. Malkhi, M.K. Reiter, Secure and scalable replication in phalanx, in: Proc. 17th Symposium on Reliable Distributed Systems, October 1998, pp. 51–58.

[51] D. Mazieres, M. Kaminsky, M.F. Kaashoek, E. Witchel, Separating key management from file system security, in: Proc. 17th Symp. on Operating Systems Principles, Kiawah Island, SC, 1999, pp. 124–139.

[52] S. Mical, Efficient certificate revocation, Technical report, MIT/LCS, 1996.

[53] M. Myers, R. Ankney, A. Malpani, S. Galperin, C. Adams, X.509 Internet Public Key Infrastructure Online Certificate Status Protocol – OCSP, IETF RFC 2560, June 1999.

[54] M. Naor, K. Nissim, Certificate revocation and certificate update, in: Proceedings 7th USENIX Security Symposium, January 1998.

[55] P. Narasimhan, Transparent fault tolerance for CORBA, Ph.D. thesis, University of California, Santa Barbara, 1999.

[56] P. Narasimhan, K.P. Kihlstrom, L. Moser, P.M. Melliar-Smith, Providing support for survivable CORBA applications with the immune system, in: Proc. International Conference on Distributed Computing Systems, May 1999, pp. 507–516.

[57] R. Needham, M. Schroeder, Using encryption for authentication in large networks of computers, Commun. ACM 21 (12) (1978) 993–999.

[58] R. Needham, M. Schroeder, Authentication revisited, ACM Operat. Syst. Rev. 21 (7) (1987) 7-7.

[59] A. Oram, Peer-to-Peer: Harnessing the Power of Disruptive Technologies, O'Reilly & Associates, Inc., 2001.

[60] M. Peinado, Y. Chen, P. England, J. Manferdelli, NGSCB: A trusted open system, in: Proc. 9th Australasian Conf. on Inf. Security and Privacy, July 2004, pp. 86–98.

[61] G. Pierre, I. Kuz, M. van Steen, A.S. Tanenbaum, Differentiated strategies for replicating Web documents, Comput. Commun. 24 (2) (2001) 232–240.

[62] B. Popescu, M. van Steen, A.S. Tanenbaum, A security architecture for object-based distributed systems, in: Proc. 18th Annual Computer Security Applications Conference, IEEE, December 2002, pp. 161–171.

[63] B.C. Popescu, B. Crispo, A.S. Tanenbaum, Secure data replication over untrusted hosts, in: Proc. 9th Workshop on Hot Topics in Operating Systems (HotOS IX), USENIX, May 2003, pp. 121–126.

[64] B.C. Popescu, B. Crispo, A.S. Tanenbaum, M. Zeemen, Expressing security policies for distributed objects applications, in: Proc. 11th Cambridge International Workshop on Security Protocols, Springer-Verlag, 2003.

Please cite this article in press as: B.C. Popescu et al., Design and implementation of a secure wide-area ..., Comput. Netw. (2007), doi:10.1016/j.comnet.2006.11.008

[65] R. Sandhu, Rationale for the RBAC96 family of access control models, in: Proc. 1st ACM Workshop on Role-based Access Control, ACM Press, New York, NY, USA, 1996, p. 9.
[66] R. Sandhu, E. Coyne, H. Feinstein, C. Youman, Role-based access control models, IEEE Comput. 29 (2) (1996) 38–47.
[67] S. Saroiu, K. Gummadi, R. Dunn, S. Gribble, H. Levy, An analysis of Internet content delivery systems, in: Proc. 5th Symp. on Operating Systems Design and Implementation, December 2002.
[68] F.B. Schneider, Implementing fault-tolerant services using the state machine approach: a tutorial, ACM Comput. Surv. 22 (4) (1990) 299–319.
[69] F. Stajano, R.J. Anderson, The resurrecting duckling: security issues for ad-hoc wireless networks, in: Proc. 7th Int. Workshop on Security Protocols, Springer-Verlag, 2000, pp. 172–194.
[70] A. Vahdat, T. Anderson, M. Dahlin, D. Culler, E. Belani, P. Eastham, C. Yoshikawa, WebOS: Operating system services for wide area applications, in: Proc. 7th IEEE Symposium on High Performance Distributed Computing, July 1998, pp. 52–64.
[71] M. van Steen, F. Hauck, P. Homburg, A. Tanenbaum, Locating objects in wide-area systems, IEEE Commun. Mag. (January) (1998) 104–109.
[72] M. van Steen, P. Homburg, A. Tanenbaum, Globe: A wide-area distributed system, IEEE Concurrency 7 (1) (1999) 70–78.
[73] M. van Steen, A. Tanenbaum, I. Kuz, H. Sips, A scalable middleware solution for advanced wide-area web services, Distrib. Syst. Eng. 6 (1) (1999) 34–42.
[74] J. Vitek, C. Bryce, The JavaSeal mobile agent kernel, in: Proc. 1st Intl. Symp. on Agent Systems and Applications, 1999, pp. 103–116.
[75] M. Waldman, A.D. Rubin, L.F. Cranor, Publius: A robust, tamper-evident, censorship-resistant, Web publishing system, in: Proc. 9th Usenix Security Symposium, Denver, CO, August 2000, pp. 59–72.
[76] B. Waters, A. Juels, J. Halderman, E. Felten, New client puzzle outsourcing techniques for DoS resistance, in: To be presented at the 11th ACM Conf. on Computer and Communications Security, October 2004.
[77] W.A. Wulf, C. Wang, D. Kienzle, A new model of security for distributed systems, Technical Report CS-95-34, 10, 1995.
[78] P.R. Zimmermann, The Official PGP User's Guide, MIT Press, 1995.

Bogdan C. Popescu is a Ph.D. student at Vrije Universiteit in Amsterdam. His research interests include network and system security, trusted computing, and digital rights management. He has received a B.Sc. degree from the University of Maine, and a M.Sc. degree from the University of Maryland at College Park.

Bruno Crispo is a faculty member at the University of Trento and at the Vrije Universiteit Amsterdam. His research interests are security protocols, authentication, authorization and accountability in large distributed systems, and sensors security. He has a Ph.D. in Computer Science from the University of Cambridge, UK. Contact him at crispo@cs.vu.nl.

Andrew S. Tanenbaum has an S.B. from MIT and a Ph.D. from the University of California at Berkeley. He is currently a Professor of Computer Science at the Vrije Universiteit in Amsterdam.

He is the principal designer of three operating systems: TSS-11, Amoeba, and MINIX. TSS-11 was an early system for the PDP-11. Amoeba is a distributed operating systems for SUN, VAX, and similar workstation computers. MINIX is a small operating system designed for high reliability and embedded applications as well as for teaching.

In addition, he is the author or coauthor of five books. These books have been translated into 20 languages and are used all over the world. He has also published more than 100 refereed papers on a variety of subjects and has lectured in a dozen countries on many topics.

He is a Fellow of the ACM, a Fellow of the IEEE, and a member of the Royal Dutch Academy of Sciences. In 1994 he was the recipient of the ACM Karl V. Karlstrom Outstanding Educator Award. In 1997 he won the ACM SIGCSE Award for Outstanding Contributions to Computer Science.

Arno Bakker (arno@cs.vu.nl) was awarded his Master's degree in Computer Science in 1996 and his Ph.D. in the same field in 2002, both from the Vrije Universiteit in Amsterdam. He is one of the designers and implementors of the Globe and Globule middleware platforms for large-scale (Web) applications. Since September 2005 he works in the Freeband/I-Share project, a cooperation between various universities in The Netherlands. Developed as part of this project is the Tribler peer-to-peer television software which aims to radically reduce the cost of TV broadcasting over the Internet.

Please cite this article in press as: B.C. Popescu et al., Design and implementation of a secure wide-area ..., Comput. Netw. (2007), doi:10.1016/j.comnet.2006.11.008