Replicated Secure Shared Objects for Groupware Applications

Technical Report 716

Andrew Rowley
Jean Dollimore
Department of Computer Science, Queen Mary and Westfield College, University of London

March 1995

This report describes the design and implementation of a secure shared replicated objects system, intended for use by groupware application programmers. The system enables such programmers to incorporate conceptually shared objects into their programs in order to provide the users with access to shared information. Access to each object is restricted to trusted parties by applying access control to its methods. Our system is different from existing implementations because it employs a replicated architecture. We believe that replication of the data is essential in order to achieve acceptable interactive performance levels when invoking methods that read the object state and indeed we show that such methods are significantly faster than a similar secure shared object system which is based on a client-server architecture which does not employ replication. The implementation includes a secure group communications system, the design of which is also included in this report.

---

1 Introduction

The increasing importance of collaborative computer applications (groupware) has fuelled a demand for distributed systems software that eases their creation. Such software can remove from the programmer's concerns, the complexities of the distribution and maintenance of replicated data by presenting a simple model, whilst maintaining acceptable levels of interactive performance, which is vitally important to these applications.

One such concept is the **shared object.** A shared object system has been developed at QMW [Achmatowicz94]. For reasons of fault-tolerance and performance, the implementation employs complete replication of object data at the machine of each participant that needs to invoke its methods. Inspection of an object's state can therefore be handled entirely locally, however changes of course must be communicated to all of the object replicas. This complexity is made transparent to the programmer through a **proxy** (an instance of class *SharedObject*) representing the conceptual shared object, which can be instantiated in application code as any other class. Invocations upon its methods will be dealt with either in a distributed manner or locally according to whether the method changes the replica object's state (an **update**) or not (a **read**). The proxy objects held by the participants form a group (an object group [Achmatowicz94]) and updates to the replica (an instance variable of the *SharedObject*) are communicated via multicasts to the group.

A certain subset of collaborative applications will have security requirements: for example an adaptation of a multi-user editor used to assist the preparation of examination papers, or a computer conferencing system used for the discussion of private issues. It is clear in both these examples that it is desirable to restrict access to a specified set of individuals and in the case of the exam paper editor in particular, to give different individuals different access rights. We believe that a programmer constructing such an application would find invaluable systems software that not only hides the complexities of distribution behind a simple abstraction but also

provides access control and other security features in a manner consistent with that abstraction. This paper describes the design and implementation of such a system. The system has been built as an extension of the existing replicated shared object system.

The highest level object in an application usually represents a real-life object such as the exam paper mentioned previously. The obvious means of controlling access to a shared object would be to control access to its methods. However, when the actual implementation of the shared object is distributed, the provision of method level access control may present difficulties. In particular we must ensure that the state of the shared object that is observed by all participants, i.e. the state of their replicas is consistent.

The secure shared object system requirements then, are as follows:

1 To allow the specification and enforcement of method level access control to a conceptually shared object.
2 For all participants to observe a consistent view of the shared object's state (we need not make this guarantee to a malicious or malfunctioning participant, however their actions should not affect those functioning correctly).
3 For the state of the shared object to be hidden from all but those principals explicitly entitled to observe it.
4 For security not to have an unreasonable effect on method invocation time.

All of these guarantees must hold in an environment where messages can be tampered with, replayed and observed by unauthorised parties.

The following section describes the architecture of the existing shared object system and explains how this is adapted to cater for the above guarantees. Section 3 describes the system's implementation and covers all of the important design decisions. Section 4 analyses the performance of our implementation and Section 5 summarises our conclusions.

2 The Function of the Secure Shared Object System

This section starts by further describing the design of the existing (non-secure) shared objects system. The security requirements of shared objects initially described in the introduction are then expanded in the light of a replicated architecture.

2.1 Shared Objects

As mentioned in the introduction, the system described here is an extension of an existing system which provides the abstraction of a single shared object. The programmer conceptually creates the object by passing an identifier to the constructor method of the class *SharedObject*. The programmer can then include in his program calls to the methods of the conceptual object.

The data of the shared object is in fact replicated. What the program actually creates is a proxy. This proxy maintains the local replica of the shared object and uses it to immediately satisfy any invocation that does not change its state (reads). Invocations that do change the state (updates) must be distributed to all of the other proxies who will all apply them to the replicas that they maintain.

The proxies form an object group [Achmatowicz94] to which updates are multicast (see fig.1). This architecture means that changes to the membership must also be multicast to the group and additionally that new members must receive state in order that they can create their initial replica. In the shared object system, state is transferred from some existing member to a new

participant. The multicasting and state transfer is handled in our system by a group communications layer called Horus [vanRenesse94]. Horus ensures that the multicasts are totally ordered and atomic, that is they are delivered to the target processes, and hence applied to the replicas in the same order everywhere, or not at all.

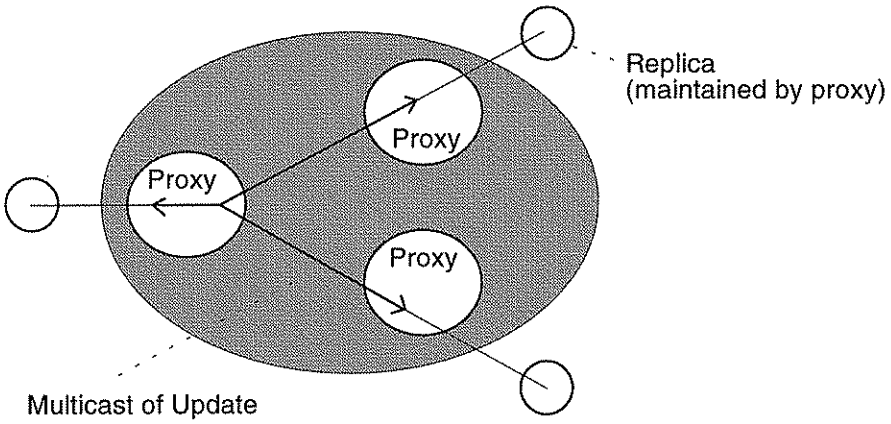

Fig. 1: Proxies maintain replicas of the shared object. Updates to the shared object are multicast to all proxies currently maintaining an object's state. Updates are then applied to the replicas. The proxies form an object group represented by the shaded area.

2.2 Secure Shared Objects

The shared object abstraction described above will only hold in a benign setting. Multicasts could be intercepted and tampered with, thus causing a false view of the state to be seen by one or all participants. Additionally, the state being transferred to a new member could be observed by an unauthorised party. In fact there is nothing stopping some outsider from joining the group and emitting any update they please. All they would need to do this is the identifier of the shared object. This identifier is a means of contacting a group member. The secure shared objects system should prevent malicious parties from disrupting the consistency of the state observed across the group.

However the world is not simply divided into two groups, those that can gain access and those that should be denied. Individuals within an organisation are trusted differently, usually according to their role, and so should be given different rights to access objects. In an example taken from [Ting90], the shared object is a medical record. Nurses should be able to update the progress of a patient, but not change prescriptions. This should only be possible by doctors. Clerical staff may need to be able to read the notes, but not to change them, and all others should not be able to see records at all. Hence different participants accessing the same objects have different rights, i.e. they can invoke different methods.

We consider a setting where communications can be observed, tampered with and replayed by intruders who possibly masquerade as others. In order that the state of a replica maintained by participants remains consistent and secure:

- Updates must be identifiable as coming from one particular principal and applied only if that principal has the right to invoke the method.
- Updates must be secret.
- Individual updates must be applied only once everywhere (or nowhere) and in an appropriate order everywhere.
- The state transferred to a new participant and updates must be authenticated and be secret.

The task of multicasting updates and transferring state is handled in our system by Horus [vanRenesse94]. Horus provides a group abstraction and is useful because it hides complexity of implementation and communication between components. However, as the above requirements show, we require security guarantees from the group abstraction before we can layer our secure shared object system on top of it and provide method-level access control. Section 3.2 describes how we have adapted Horus to provide these guarantees.

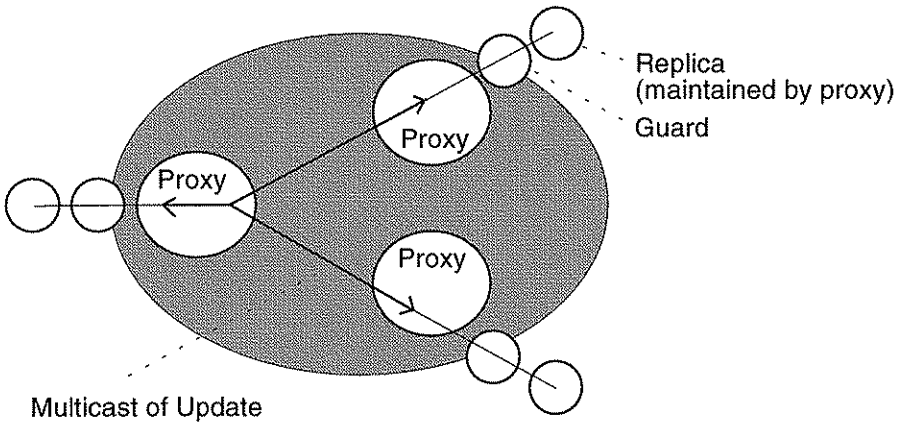

Fig. 2: Guards filter out unauthorised updates before they are performed upon the replicas.

In our model, access control is performed at each participant's machine, i.e. each participant is responsible for protecting itself. This means that uncorrupted participants will all view the same state of the object as long as they all receive the same (secure) updates and they all apply the access control consistently. This is achieved by adding a guard (see fig. 2). The guard filters out attempts to perform updates that are non permitted.

3 Implementation

This section describes the problems encountered and the decisions made when implementing this secure replicated shared objects model and the secure group communications system that is layered below it.

3.1 ACLs or Capabilities

The rights granted to principals to access objects can be expressed in an access matrix, which relates operations on the objects to the principals authorised to invoke them. This matrix is potentially large and hence holding an access control matrix as a whole rarely happens in implementations, instead the information is held either in capabilities or access control lists. The difference between using capabilities and access control lists (ACLs) for checking rights is derived from two possible methods of dividing up the matrix. Dividing it one way attributes a set of rights to a particular principal (as a capability does) whereas dividing it another way attributes a set of principals to one operation (as in an ACL).

Conventionally, a capability contains the information explained above and is given by some server to the principal, combined with additional data ensuring that it can't be altered or forged. The client can then at a later time, as it attempts to perform some operation, present the capability to the server as proof of entitlement to perform an operation. Capabilities are used this way in the Amoeba distributed operating system [Mullender86] and are not secured when transferred between servers and clients.

In our replicated object group model, the capabilities would need to be verified by the individual guards, however they would have to be issued by some other authority since it is obviously not acceptable for principals to be to able issue their own rights. In this case, the authority issuing the capability is not the same as the party performing the check. If the capability could be packaged and given to the principal by some recognised party, in such a way that it can't be altered and can be attributed to that authority, then that capability could be presented by the principal to others as proof of its rights. The integrity of the access control information could be ensured by including with it a digital signature (See [Rivest78] for a complete description of digital signatures) which is checked by the recipient. Thus the capability in this context more closely resembles a signed certificate of rights. It may have the following structure:

Principal P can perform operation X, Until Time T, Signature of some Authority

And then a request by P would contain this capability:

Request to perform X, Capability, Signature of P

Alternatively, every party responsible for the checking of rights (i.e. the proxies forming the object group) could obtain an ACL, a list of principals entitled to perform each operation. They could conduct a check before performing any operation received from another participant.

The merits of ACLs versus capabilities have been widely discussed with regard to servers controlling access to clients. Succinctly, the convenience of capabilities is counter-balanced by the difficulty in revoking rights. Revoking a right immediately would mean locating all capabilities issued. Inserting a lifetime into the capability after which it must be refreshed from source is a more practical solution.

The work involved in the checking of rights using ACLs or capabilities in our system turns out to be very similar. The major part of the work in both alternatives involves the slow process of checking signatures. Updates arrive with a signature authenticating the source. Therefore the checking of rights against an ACL involves one signature check. The same check of the signature must be made when the update arrives with a capability. Additionally however, the signature of the issuing authority contained within the capability must be checked, however, this check need only be performed the first time that a particular capability is observed. Subsequent checks need only compare the byte values of the received capability with those of the first received, since its authenticity is already proven. There is an additional overhead of distributing the ACLs to all parties which will perform checks, however since the size of the ACL is likely to be small and this will only happen at the start of a session, we expect this not to be a substantial issue.

Revoking a capability, as mentioned previously, is a problem which can be overcome by placing an expiry time within its body. This however leads to a problem resulting from not all participants using closely synchronised clocks. The exact point of expiry may not be precisely the same at all participants' machines. Hence an update issued and corroborated with a capability on the point of expiry may lead to some requests to perform an operation being accepted but others being rejected, resulting in the update being applied to only some of the replicas and hence an inconsistent state appearing across the group. This would be a violation of the second system requirement as set out in the introduction. It is this complex problem that resulted in the decision to choose ACLs at the expense of having to maintain the distributed state.

3.2 Secure Group Communications

As explained earlier our secure shared object system was built using an underlying group communication system called Horus. Horus is flexible and can provide different abstractions to

suit an application's needs. Our original (non-secure) shared object system makes use of the following features:

- Totally Ordered (Reliable) Atomic Multicasts.
- State and group membership transfer to new group members.

The purpose of any abstraction is to hide complexity. However, this has consequences when building a secure system, since data which helps maintain the abstraction can be held and transferred in an unsecured manner. For example, the group system as we use it provides a total ordering. Ordering information is transferred to the group along with multicasts. If this ordering information is not proved authentic by the recipients, then the abstraction is open to attack, since the ordering information could be tampered with resulting in an inconsistent state being observed across the group. Since this ordering information is hidden to the programmer, it is clear that the group communication system used must give its own security guarantees.

Considering the group communication features introduced above, the additional security requirements are:

- Authentic group joins: The group must authenticate the new member and the new member must be sure he is joining the authentic group.
- Multicasts which are authentic, secret, securely totally ordered and atomic.
- Secret transfer of state which can be authenticated by the recipient.

Simply encrypting communication in a group key provides us with secrecy. This shared key, referred to as the **group communication key**, must be securely distributed to all group members along with the state and membership as part of the group join process.

Adding digital signatures to all communication can provide us with authenticity, but the encryption of the signature cannot be performed with the group communication key, since this would merely pin the source of the communication down to a (non-specific) member of the group. If access control is to be performed on a per-principal basis as we wish, then the communication must be identifiable as coming from one particular member. The multicast primitive forces us to send the same message to each member and hence group communication is signed with **individual private keys**. An alternative which we rejected is to establish separate secret keys between all pairs of members (this decision is explored further in the performance section). Members merely drop multicasts that cannot be correctly authenticated as coming from other members of the group.

We do not deal with how to make the multicasts securely atomic. Making multicasts atomic in a benign setting is impossible if no assumptions are to be made about message delays (as in a completely asynchronous system). See [Fischer85] for a thorough discussion of this issue. We say only that making the existing multicast secure amounts to ensuring that the source actually includes every destination, which could be achieved by making the source prove that an authentic multicast primitive was used. This may be achieved through secure loading of the primitive which may have to be under the control of the operating system. [Lampson92] includes a description of exactly what secure loading can achieve.

Note that our implementation did not actually implement security of ordering information contained within update messages. This was due to the complexities involved in changing the low-level communication primitives of the Horus system. Ensuring a secure ordering within a group however is merely a matter of ensuring the integrity of the ordering information. It could be incorporated into the protocol outlined in section 3.4 by including the information in the signed body of the update message.

Secret and authentic transfer of state is also achieved using the keys introduced above. The state is encrypted using the group update key and signed using the sender's private key. There

are no constraints at the group communications level concerning from whom a new participant receives the state. This is an issue of trust which is properly left to the secure shared object layer (see section 3.3 below).

How a new participant securely authenticates the group is dealt with in the following section. The details of the protocol for group joins and secure multicasts is given in section 3.4.

3.3 The Source of State

Potentially many participants can hold a shared object's state. If all have been applying access control correctly, then all will have a state which is consistent with the others. However a need to impose method level access control implies that not all of the members are trusted equally, so from where does a new member request an initial copy of the state? The options were identified as follows:

- Receive state from all members (some of which are potentially corrupt) and resolve any conflicts.
- Receive state from some principal that we trust above the others and assume not to be corrupted.

Both of these solutions have potential problems. The first causes problems if there is only one other member. We will end up receiving the state which that member wishes us to observe because the first member can effectively cause the state of the object to be anything it wishes. Additionally, if the group consists of members all intent on sending us their corrupt versions of the state, then it may be impossible to decide upon a correct version. The second solution is better, but will cause problems if that trusted principal is not currently participating.

The solution is to ensure that there is always some member that is participating which can be trusted above ordinary members. To make the additional trust feasible we can ensure that the process containing the member runs on a more trusted machine. We refer to this new component as the **pseudo-participant**. This component will run with the authority of a principal known to all participants and so the state received can be authenticated as coming from this source. This component does not issue updates of its own as it is not associated with an interactive user. We refer to this principal as the **task owner**, as we assume that the same principal will be used for secure shared objects related to the same task. An ultimate goal of this work is to integrate it into the Task structured method of specifying access rights that is being researched at QMW [Coulouris94]. This principal provides the new participant with a means of securely authenticating the group associated with the object.

Note that because pseudo-participants are the same in every respect to other participants, there is no limitation on the number of them in any current group. Therefore, they could be numerous in order to maintain the advantages of fault-tolerance gained by using replicated state, however there must always be at least one running at all times if new members are to securely obtain the state for their replicas.

The pseudo-participant constructs the initial state (upon creation of the group) by executing the object's constructor function.

3.4 The Protocol for Secure Group Communication

This section describes the specific protocols chosen for the communications between components. It provides the secure group communications described in section 3.2. Its implementation was built on top of the non-secure group system Horus and so the protocol description is stated in terms of the abstraction Horus provides, i.e. group multicasts and some

point-to-point messages as part of the group join process. Below is a description of the notation used:

$A \rightarrow B: X$	Principal A sends message X to Principal B
{X}K	Message component X is encrypted using key K.
[X]K	Message component X is digitally signed using key K.
n	A Nonce for preventing replay.
t	A logical time stamp (a sequentially increasing integer).

Each application process has a proxy object for each shared object it is currently accessing. The processes run on behalf of single principals. The protocol description below is expressed in terms of the principals sending and receiving and not the components. The pseudo-participant is run with the authority of a principal called the task owner. As the name suggests, there would normally be one task owner associated with each organisational task. The identity of the task owner must be securely obtained by a prospective member.

The following notation is used to refer to principals:

T	Refers to the task owner principal;
G	Refers to the entire group, i.e. $A \rightarrow G$ represents a multicast to the group.

The keys introduced in section 3.2 are as follows:

K_G	Refers to the shared group update key.
K_A, K^{-1}_A	Refers to principal A's public and private keys.

If one does not already exist, a proxy object randomly generates its own public and private key pair for the principal on whose behalf the process is running. The public half is made available through a key distribution service from which other components can securely obtain it. The private half is never transmitted outside the process in which it was generated. The proxy is also responsible for renewing the key periodically in order to reduce any risk of cryptoanalytical attack. Likewise the pseudo-participant is also responsible for renewing the shared group update key.

The initial state of the shared object is obtained by the pseudo-participant by running a constructor method provided by the programmer implementing the secure shared object's class. There is no exchange of information as this initial member establishes itself. Subsequent members however must securely obtain the state (including the state of the ACL) from the trusted pseudo-participant.

There are two points of view when considering how to securely exchange the state. A new member must be sure that the state it is receiving is really from the pseudo-participant which it trusts to maintain the authoritative state. Secondly however, the pseudo-participant must ensure that the new participant receiving the state has the right to view it and of course must ensure that no one that isn't can interpret it.

The state should not be given to every requester. Possession of an object's state effectively provides a means of executing all of the object's read methods, even if this is disallowed by the ACL. Hence, the pseudo-participant will only transfer the state to principals which have the right to invoke all of the object's read methods.

Messages 1 and 2 will achieve the secure transfer of state, where principal A's proxy becomes the new member:

1.	$A \rightarrow T$:	$[A, n]K^{-1}_A$	A sends request to task owner.
2.	$T \rightarrow A$:	$[\{state, n, acl\}K_G, \{K_G\}K_A]K^{-1}_T$	Encrypted state is returned.
3.	$T \rightarrow G$:	$[A, n, t, t_{ARRAY}]K^{-1}_T$	Task owner informs the group.

Threats	*Message 1*	*Message 2*	*Message 3*
Masquerading Src.	Prevented by sig.	Prevented by sig.	Prevented by sig.
Eavesdropping	N/A	Prevented by encr.	N/A
Tampering	Prevented by sig.	Prevented by sig.	Prevented by sig.
Replaying	N/A	Prevented by nonce.	Prevented by time-stamp and nonce.

Only holders of the group update key will be able to interpret the state of the object. The group update key is included in message 2, encrypted with the receiver's public key. The nonce is included to prevent replay of the state to A next time it joins the group. The pseudo-participant then multicasts to the entire group to announce the new member. Also the new member is informed of all members logical time-stamps in an array plus his own starting time-stamp which is randomly generated by the pseudo-participant. The nonce in this message means that the new participant can verify that it is not a replay, the timestamp is so that all other members can do the same. Time-stamps are included in all subsequent messages and are used to prevent replay. Additionally, we do not consider a principal's participation to be secret, hence message 3 is not encrypted.

The exchange of messages involved when an existing member leaves the group are as follows:

4.	$A \rightarrow PC$:	$[A, t]K^{-1}_A$	A informs PC that it is leaving.
5.	$PC \rightarrow G$:	$[A, t]K^{-1}_T$	PC informs the group.

Threats	*Message 4*	*Message 5*
Masquerading Src.	Prevented by signature.	Prevented by signature.
Eavesdropping	N/A	N/A
Tampering	Prevented by signature.	Prevented by signature.
Replaying	Prevented by time-stamp.	Prevented by time-stamp.

Masquerading, tampering or replaying of both messages 4 and 5 would amount to a forged leave which could be considered a form of denial of service.

Message 6 is an update message (a secure shared object method invocation which changes the object's state):

6.	$A \rightarrow G$:	$[A, \{update\}K_G, t]K^{-1}_A$	A multicasts an update to the group.

Threats	*Message 6*
Masquerading Src.	Prevented by signature.
Eavesdropping	Prevented by encryption in group update key.
Tampering	Prevented by signature.
Replaying	Prevented by the timestamp.

As mentioned previously, updates must be identifiable as coming from one particular principal alone. This is achieved by incorporating the signature into message 6.

Performing an update to the ACL is essentially the same as an update to the shared object's state.

The group update key was introduced in order that updates cannot be interpreted by parties not entitled to observe the shared object state. There must be a mechanism for the pseudo-

participant to install a new group update key. This is necessary in the case where a principal's right to have total read access is revoked, i.e. it does not have the right to perform all read operations any longer. Again, as with principal's public and private keys, it is wise to refresh the group update key periodically in order to reduce the possibility of cryptoanalytical analysis leading to discovery.

7. $T \rightarrow G$: $[A, \{t, K_G\}K_A, B, \{t, K_G\}K_B, \ldots]K^{-1}_T$ PC informs the group of key.

Threats	*Message 7*
Masquerading Src.	Prevented by signature.
Eavesdropping	Prevented by encryption in public keys.
Tampering	Prevented by signature.
Replaying	Prevented by timestamp.

The signature in message 7 is an encrypted digest of the unencrypted new group update key (not the whole message) as a performance optimisation. The new key is encrypted several times, once in every one of the current participant's public keys and is preceded in the message by the intended recipient principal's identity so that recipients can identify which component to decrypt.

4 Performance

Performance tests were carried out on the secure shared objects system in order to establish that the impact of security was not unacceptable. A group of two members was used to establish the time taken between an update being initiated and it being applied to the remote replica. Additionally, a breakdown of this time was established by timing particular tasks in isolation. The measurements are **summarised in Table 1 below.**

Update Breakdown	*Time (ms)*
Un/Marshalling	9
Update De/Encryption	1
Signature Production/Checking	254
Multicast Latency	210
Total Time for an Update	**474**
	Time (ms)
Total Time for Read	**0.1**

Table 1: Performance of Secure Shared Objects. Figures were obtained on a Gateway 2000 PC with a 66MHz Intel 486 processor, running the NeXTSTEP operating system and using 16 byte signatures and 16 byte keys.

The advantage of a replicated architecture is of course that method invocations that do not change the state can be performed entirely locally and are correspondingly fast. Updates on the other hand incur many penalties, the chief two being that of signature production and multicast latency.

The multicast latency could only be reduced by using a different group communication system, although we do consider our measurement to be surprisingly high. The time taken producing the signature is almost entirely taken up by its encryption using slow public key encryption [Rivest78]. This overhead could be substantially removed by using faster shared key encryption. Authentication using this method however, would mean adding one signature for each group member and thus would only be a performance advantage when the members remained below some number *n*. This number can be calculated roughly using the figures obtained above.

For an update with size *i*:

When the time taken for updates to be reflected in replicas using public keys is equal to the time taken using shared keys:

PrivateKeyEncryptionRate + PrivateKeyDecryptionRate = 2.SharedKeyEncryptionRate.n

We assume all other factors are equal. In fact multicast latency will increase as the size of the message increases due to the extra signatures. However we ignore this since Horus literature [vanRenesse94] reports that when message sizes are low, latency less than doubles as the message size increases from zero to 4000 bytes for a total ordering. This size of message should be ample for most updates in our system.

Solving the above equation we get n equal to approximately 120 members. Since n is quite large, then using shared key encryption for authentication is therefore a viable alternative option. However, it should not be forgotten that the time taken for a new group member to become integrated will increase due to the more complex procedure of establishing shared keys with all other members. Consequently in an application where group joining and leaving is a common occurrence, then secret key authentication may be less attractive.

A similar secure shared objects abstraction has been developed at the Digital Systems Research Centre, called Secure Network Objects, which does not utilise replication of the data [vanDoorne95]. The timing figures for this system are displayed side-by-side with ours.

	Reads (ms)	*Updates (ms)*
Secure Shared Objects	0.1	474
Secure Network Objects	0.7	0.7

Table 2: Comparison of Secure Shared Objects and Secure Network Objects. Secure Network Object figures were obtained using DEC 3000/700 workstations, containing DECchip 21064a 225MHz processors. Key and signature sizes are not given.

Although a direct comparison of their performance figures with ours is not possible due to the difference in hardware used to obtain them, it is clear, as expected, that reads performed with the replicated architecture are far more efficient, with updates being more expensive. Since the ratio of reads to updates is application dependant, then so too is the answer to the question of which is the more efficient of the above two implementations.

5 Conclusions

We have presented our implementation of a secure shared object abstraction, implemented (unlike any other known to us) through replication of an object's state, thus bringing the dual advantages of performance and fault-tolerance. Access control is specified at the level of the shared object's methods.

Data comprising the object's state cannot be obtained by any process running on behalf of a principal which is not entitled to invoke all of the object's read methods. Likewise, only principals which are entitled to perform an operation which updates the object's state will have that update reflected in the replicas maintained by other correctly functioning participants. Hence all non-corrupt members will observe a consistent view of the object.

We make this consistency guarantee with one stipulation. This is that corrupt participants, operating on behalf of principals that are permitted to perform methods which change the state are not able to send an update to only a proportion of the participants. This could possibly be achieved through secure loading [Lampson92] of the multicast primitive or an alternative model

where multicasts can only be performed by the more trusted pseudo-participants. All updates by lesser trusted parties can therefore be directed through them.

Performance of the system is of course extremely good for method invocations that do not change the state, since these can be handled entirely locally. However the trade-off is that operations which do change the state are slower and thus our replicated architecture will only be acceptable for use in applications with a sufficiently high ratio of reads to updates.

As explained in the section concerning secure groups, our implementation does not include a secure transfer of ordering information and hence is open to tampering. This was not dealt with chiefly because of time-constraints. A thorough analysis of the requirements of secure group communications in being undertaken and hopefully a complete implementation will emerge. Our basic implementation using Horus has however allowed us to draw useful conclusions as to the validity of a replicated method of providing the secure shared object system.

6 Future Work

The secure shared objects system described here will, as it stands, only allow the specification of rights in terms of simple principals. A typical organisation utilising such a system as this will involve many principals accessing many objects. This complexity results in the procedure of specifying access rights to all objects becoming a laborious task. In order that this system be usable in a large organisation, it must be possible for the secure shared objects system to be extended to encompass the notion of compound principals devised by [Lampson92].

Grouping principals according to their organisational roles is the usual means of simplifying the access specification task. For example, it may be the case that computer science staff should all have the ability to read student record objects. Granting access to all computer science lecturers individually would be time consuming. Additionally, if one member of staff were to leave, then the access rights would have to be revoked at each record object individually. Granting access to all computer science staff as a group would be far simpler. Each of the staff could then be issued with a group membership certificate signed by some authority that could then be presented to others as proof of their entitlement to invoke a method. These certificates should of course incorporate a time-out forcing all principals to refresh them periodically. This would enable the member of staff's rights to be revoked by contacting merely the authority that issues the certificates.

Delegation could also be facilitated by adding the ability to issue certificates. A delegation certificate would be issued to the delegate which can be presented to a third principal as proof of entitlement to take on some of the granting principal's rights. These certificates too would of course contain a time-out.

Both of the above proposed extensions to our system would involve the presenting of certificates containing time-outs to the participants sharing an object. And both therefore suffer from the same problem that caused us to reject capabilities in favour of access control lists (section 3.1). That is that certificates on the point of time-out may be accepted as proof of rights by some participants but not by others, causing replicas to have inconsistent state.

One possible solution to this problem is that ordinary participants do not check the expiration times of certificates at all, this task being left to the pseudo-participant. This trusted process could multicast a revocation to a certificate as soon as it detects that the expiration time has been reached in an update it has received. Of course a non-corrupt process would refresh a certificate long enough in advance that revocation would never become necessary.

The practical design of an extension enabling specification of access rights to complex principals and the possibility of delegation is being investigated further.

The disappointing performance of our system is being addressed as part of the secure group communications work. It is possible that by altering the method of authenticating group multicasts that the reliance upon slow public keys can be removed.

Acknowledgements

Thanks go to George Coulouris and Tim Kindberg for their help with preparing this report and for their contributions in the discussions that fuelled the project.

References

[Achmatowicz94] Achmatowicz, R, Kindberg, T, "Object Groups for Groupware Applications: Application Requirements and Design Issues", proc. European Research Seminar on Advances in Distributed Systems, pp. 269 - 274, 1994.

[Coulouris94] Coulouris, G, Dollimore, J, "A security model for cooperative work", Sixth SIGOPS European Workshop, Dagstuhl, Sept 1994.

[Fischer85] Fischer, MJ, Lynch, NA, Paterson, MS, "Impossibility of Distributed Consensus with one Faulty Process", Journal of the Association for Computing Machinery, Vol. 32, No. 2, pp. 374 - 382, April 1985.

[Lampson92] Lampson, BW, Abadi, M, Burrows, M, and Wobber, E, "Authentication in Distributed Systems: Theory and Practice," ACM Transactions on Programming Languages and Computer Systems", Vol. 10, No. pp. 265 - 310, November 1992.

[Mullender86] Mullender, SJ, Tanenbaum, AS, "The Design of a Capability-Based Distributed Operating System", Computer Journal, Vol. 29, No. 4, pp. 28 - 300, 1986.

[Rivest78] Rivest, RL, Shamir, A, Adleman, L, "A Method for Obtaining Digital Signatures and Public-Key Cryptosystems", Communications of the ACM, Vol. 21, No. 2, pp. 120 - 126, February 1978.

[Ting90] Ting, TC, "Hospital Records Security", Database Security: Status and Prospects, Vol. 3, North Holland, 1990.

[vanDoorn95] van Doorn, L, Abadi, M, Burrows, M, Wobber, E, "Secure Network Objects", TR 385, Digital Systems Research Center, Palo Alto, California, USA, 1995.

[vanRenesse94] van Renesse, R, Hickey, TM, Birman, KP, "Design and Performance of Horus: A Lightweight Group Communications System", Department of Computer Science, Cornell University, 1994.