A Security Architecture for Distributed Groupware

Rowley, Andrew

ISSN 1369-1961

Department of Computer Science

Technical Report No. 764

A Security Architecture for Distributed Groupware

Andrew Rowley

QUEEN MARY
AND WESTFIELD COLLEGE
UNIVERSITY OF LONDON

September 1998

A Security Architecture for Distributed Groupware

Andrew Rowley

September 1998

Department of Computer Science

Queen Mary and Westfield College

University of London

Submitted to the University of London for the Degree of Doctor of Philosophy

Abstract

Groupware applications involve multiple participants collaborating through shared data in order that they can tackle a common task more productively than if there was no computer-supported interaction between them. Many such tasks have security requirements. Measures for securing communication and controlling access to the shared data could be added by the application programmer, however this is burdensome and can result in inconsistencies in the levels of security enforced. This thesis presents the design and implementation of system software for secure group communication and access control specifically targeted at the groupware application programmer.

Group Communication facilitates the fast interactive response times that are essential for productive groupworking. However, Secure Group Communication for Groupware is more complex than simply securing the group from the hostile actions of those outside, which is the approach taken by most existing secure group systems. Group tasks typically involve the multiple participants acting with different rights, i.e. different levels of trust. This is recognition that corrupt activity can also originate from inside the group. The group communication system for groupware presented here recognises and deals with this threat whilst maintaining the speed of other systems.

Although important for efficient groupworking, group communication does not present the most appropriate level of abstraction to the programmer. A shared object abstraction is far more convenient. For secure groupware, shared objects can be protected at the level of their methods by associating with each of them an Access Control List (ACL). However conventional ACLs are relatively static, whereas the security policies of group tasks are far more dynamic. This thesis introduces two techniques for specifying and enforcing dynamic policies. Firstly State-Dependent Access Control allows rights to reflect the current state of the group task. Additionally, the notion of consent is often a feature of group security policies. Secondly therefore, the concept of Backing is introduced as an attempt to

mechanise the collection and proof of consent from some group of other participants.

The complete body of work allows secure and efficient groupware to be more easily built. The resulting application instances can collectively support complex security polices and can communicate both safely and efficiently.

Table of Contents

Glossary

Access Control List	A list of principals associated with methods of an object that describes who can invoke each of the methods.
Backing	Support from one or more principal(s) for another to perform some action, i.e. A gives his 'Backing' for B to do C.
Backing Certificate	A digitally signed certificate containing a single principal's consent to back another. This can then be presented as proof (or part proof) of Backing.
Digital Signature	An electronic version of a handwritten signature used to verify the identity of the originator of a sequence of bytes such as a network message or certificate. A signature is an encrypted hash of the message and appended to it.
Distributor	A uniquely trusted member of a secure group with the special responsibility for distributing multicasts to the other members.
Groupware	Software designed to facilitate interaction and collaboration between groups of computer users.
Group Encryption Key	A symmetric encryption key used for ensuring secrecy of multicasts in the Group Communication System for Groupware.
Guard	Object that protects a Shared Object by intercepting attempts to invoke its methods. Typically the Guard consults an access control list and either accepts or rejects the attempts accordingly.
Individual Member Key	A symmetric key used by the Group Communication System for Groupware which authenticates members to the trusted Distributor (and vice versa).

Model of Trust	A set of assumptions about the behaviour and misbehaviour of components forming part of a secure distributed computer system.
Outstanding Backing Object (OBO)	A repository for requests for backing and any certificates that result. The OBO is a Secure Shared Object and is assumed to be persistent whilst requesters and backers go on and off line.
Participant	A principal that contributes to some group activity.
Practical Groupware Model of Trust	A model of trust appropriate for Groupware applications. One member of the group is trusted above the others to maintain the security of the group.
Role	A generalisation about a principal's organisational purpose. Rights are assigned to generic roles instead of individuals for convenience.
Secure Shared Object	An object conceptually shared by the applications run on behalf of principals participating in a secure group activity. The Shared Object is the means of communication between them and is protected through fine-grained access control that is applied to its methods.
Security Policy	A high-level description of the rights of individuals.
Security Shell	The means by which the user can interact with the security functions of the Secure Shared Object layer, e.g. for changing the access control rights, delegating rights or granting backing.
State-Dependent Access Control	Enforcement of rights where those rights do not just depend upon the principal's identity and the action being attempted, but where rights depend also on the current state of the system.
Vector Signature	A variant of a digital signature used to sign a message that is intended to be verified by multiple participants, but yet still make use of fast asymmetric key cryptographic techniques.

Acknowledgements

I would like to thank my supervisor Jean Dollimore for her considerable effort in guiding me through this research. I am indebted for her advice and encouragement throughout my four years of study. I hope that as her last PhD student before her retirement this thesis represents a body of work of which we can both feel proud.

I am also grateful to George Coulouris and Tim Kindberg for their help over the last four years. The initial inspiration and subsequent development of nearly all of the ideas written down here would not have been possible without them.

Finally I would like to thank Sam and my family for their help and support. They have all assisted greatly in ways that at the time might not have even been apparent to them.

1

Introduction

Some collaborative tasks have security considerations. There are many examples of such tasks that do not involve computer collaboration, for example: the collective preparation of a secret document. There are however not many examples of secure groupware (that is computerised group tasks with security concerns) although quite clearly these tasks too could benefit from computer aided collaboration. There is a lack of system software support for secure groupware and consequently the applications do not exist because the risk of running them on unsecured systems is too great.

1.1 Research Motivation

The security risks of conducting a secure collaborative task on an insecure distributed computer system are two-fold. Firstly wide-area computer networks are fundamentally insecure because of the lack of any centralised control. Communication can pass through many different areas of jurisdiction, some may be trusted by the source and the destination of the communication but some may not. Even if the networks along the way were trusted to securely play their part in passing on the communication, if the message is passed over a broadcast network then all the machines and their users must be trusted not to examine the contents of the messages. Such trust is just not appropriate for the Internet.

Implementations of secure communication abstractions have been developed for the Internet. Typically these employ strong encryption to guarantee the secrecy and authenticity of data. However such 'one-to-one' communication abstractions are not appropriate for group communication which is typically used at the heart of a distributed group activity for replicating data.

Secondly, complex group tasks often require complex security policies to be enforced. Secure communication will keep the task safe from the malicious activities of non-participants, however generally a secure task will involve participants that are trusted to different extents and so are perceived to pose a risk. A security policy exists outside the computing environment and is often devised without reference to the computerised version of the application. In fact in many cases where computerisation is being introduced to aid an existing task, the policy actually predates the computer application altogether. However once computerised, the policy must be accurately represented so that it can be enforced by the system.

Access control lists hold a form of security policy. However they are limited in the policies that they can describe. They were not devised with collaborative group tasks in mind and do not therefore reflect the dynamic nature of rights in such applications.

In short, the lack of secure group communication and a means of representing and enforcing complex policies are preventing secure group tasks from benefiting from computerisation.

1.2 Research Contribution

Each of the two areas of secure group communication and access control have been studied in the context of groupware applications. Therefore there are two areas of contribution:

- **Secure Group Communication for Groupware (with a Model of Trust for Groupware)**

  The trust afforded to both participants and non-participants in a group activity can be complex and varied. Through studies of group tasks and security policies a model of trust for groupware systems has been developed. This forms the foundation for the design of a secure group communication system for groupware, which unlike previous secure communication systems does not rely upon the total trust of all parties involved for its correct functioning.

  Good system performance is essential for group applications to effectively exploit collaboration. The secure group communication system for groupware has been implemented and timed tests conducted which prove that the groupware model of trust can be applied efficiently.

- **Access Control for Groupware**

  Extensions to conventional access control lists are proposed. The extensions reflect the dynamic nature of trust and hence rights within a group task. Many security policies are examined and shown to be expressible using the augmented access control lists. Whilst it is impossible to demonstrate that every policy could be expressed we argue that many policies include common security concepts and that the new techniques cater for these.

1.3 Thesis Outline

The thesis continues in the next chapter with a brief examination of what requirements group applications make of the underlying system in order that they can be both productive and secure. Chapter 3 then continues with a thorough review of existing systems and aims to show that none absolutely meet the requirements that were derived in the chapter 2.

The main body of the thesis is divided into two broad sections concentrating on the two broad areas of deficiency in existing systems: Group Communication and Access Control. Chapters 4 through to 6 present and evaluate the proposed Group Communication System for Groupware and Chapter 7 links this in with Access Control. Chapters 8 and 9 actually present the new ideas for Access Control for Groupware. Finally the conclusions summarise the work and its evaluation and point towards possible further areas for research.

The thesis contains three appendices which allow some of the surplus detail to be removed from the main text. Each is referred to in the appropriate place.

2

Groupware Application Requirements

By looking at a brief but varied array of application types, the aim of this chapter is to establish two key points regarding secure groupware applications. Firstly in Section 2.1, we show that no matter what the nature of any specific application, generally for a collaborative task to be productive then the interactive response time that a user observes needs to be quick. In other words, the time between a user action and the results of that action being observed by the initiator and any other users that it affects must be low.

The aim of Section 2.2 is to introduce the dynamic nature of access rights within a security sensitive group task. Security policies taken mainly from collaborative activities that exist outside the realm of computing are observed. This is necessitated by a lack of secure groupware case studies, this in turn being due to a lack of mechanisms for appropriately securing computer aided collaboration. Providing new mechanisms which cater for dynamic rights will enable secure group tasks too to benefit from computerisation.

The two observations taken together (speed with accurate enforcement of security policy) form the two overriding principles which guided the direction of this research and form the basis of its evaluation.

2.1 Distributed Groupware Applications

Complex organisational tasks nearly always require co-operation and collaboration in order to maximise efficiency. However this is not often aided by any computer application involved in the task, which usually is built on a system that will shield the user from the need to know about the presence of other concurrent activity. This typically leaves any co-ordination of collaboration outside the realm of the application and is to the detriment of task productivity.

Groupware turns this around by making participants aware of the concurrent activity of others.

Ellis, Gibbs and Rein [EGR91] define groupware as:

> *"Computer-based systems that support groups of people engaged in a common task (or goal) and that provide an interface to a shared environment."*

The shared environment (no matter how it is implemented) is the focus of collaboration and is the means of communication between the participants. The purpose of this section is to demonstrate that any user activity that affects this shared environment must be quickly observed by the user and the other participants in order for collaboration to be best exploited.

This case is promoted by looking at three example applications taken from different literature sources.

- **Shared Document Editing**

  A group editor was studied by Ellis, Gibbs and Rein [EGR91] which permitted the simultaneous authoring of a shared document. The editor was used over a period of time by up to six participants in any one concurrent session. The experiences of the users were summarised in the paper and generalisations of key design issues were drawn from the results. Key amongst these results is that synchronous groupware should be sufficiently responsive.

  This need is particularly well illustrated when the editor was used for group brain-storming sessions. It was clear that such sessions relied for their effectiveness upon thoughts being quickly communicated to all participants, i.e. updates to the shared environment must be visible everywhere with low latency.

  It is not only updates that need to be quick. Reading the state of the shared environment must be also. In fact quick reading of the data can be considered even more important than quick updating since typically it is done more often. Users of a shared editor will often scroll up and down the page reading different areas of the document. The smallest of delays would render the application unusable.

- **Real-Time Conferencing**

  Grief and Sarin [GS86] looked at real-time conferencing. Although the transmission of the audio and video data streams does not affect any shared environment, the session management does. In order to keep the conference under control, only one person at a time was permitted to speak, whose identity was chosen by the chairperson. Additionally, real-time voting was used to make decisions.

  The whole value of co-locating (or virtually co-locating) people for a discussion is in the immediacy of their contribution. In a virtual conference the session management too must keep up with the pace of the discussion. Excessive delays between speakers and votes would likely cause the participants' concentration to wane.

- **Group Decision Making**

  Tatar, Foster and Bobrow [TFB91] discuss their experiences with a tool designed to assist remote people agree upon a plan. The system brought together video-conferencing with other shared tools such as editors and work surfaces. Their investigations were aimed at distilling the key implications for the design of groupware.

  One of these was of course response time. They observed that delays and in particular large changes in delays caused actions to become uncoordinated. Just one example arose when a participant was speaking whilst typing into a shared editor. The delay in transmitting the text caused the descriptions of what he typed to become unsynchronised with the text. This caused considerable confusion and frustration. Reducing the delay would of course solve the problem and the authors chose to alter the design of their system to this effect.

There are of course many other examples of synchronous groupware in the literature that could have been used as examples, such as: shared white-boards, meeting rooms [VDN91], shared spread-sheet production [NM91] shared diary or calendar support [GS86] and real-time chat systems. For virtually all examples it is clear that the advantages of distributed collaboration only apply if the system supporting the collaboration is sufficiently responsive.

Indeed this observation is true for any interactive system, not just a distributed or collaborative one. However if the application is distributed the solution is less obvious than just writing more efficient code or obtaining a faster processor. As

Chapter 3 will make clear, the designers of such a system will typically have to consider replicating the shared data.

2.2 Security Policies of Secure Group Activities

Group activities should not be excluded from all the benefits that computerisation brings just because the applications are being used to aid a security sensitive activity. However, this is indeed what happens.

New systems are not introduced because of the increased perceived threat that computerisation together with distribution brings. This is certainly not without reason. A computer offers more discreet access to the shared data (remoteness hides) and adequate security mechanisms do not exist that enable accurate reflection of policy. Any of the example applications from Section 2.1 could be used to aid security sensitive collaboration, but typically they are not.

This section will briefly examine security policy and group activity from both the medical and financial fields. These two areas currently rely on collaboration for productive work to take place, but do not currently support this using distributed groupware.

It will be shown that security policies are often very dynamic in nature. The rights of the principals involved change according to many factors including the state that the task is currently in, the time of day and the changing opinions of other principals involved.

The dynamic nature of rights is usually an attempt by the designers of the policy to ensure that the attempt to do something is in the context of a legitimate activity. At one point in a task it is acceptable to do that something, but at another point it becomes unacceptable. Take for example a doctor who needs to look up details from a patient's medical record. This is permitted if the patient is currently undergoing treatment, otherwise it is strictly forbidden. In other words rights are dependent upon the context in which the access is attempted. Surgeries often have paper records stored in clear view of other staff and this reduces the likelihood of an attempt to make an illegal access. However computerisation more effectively hides the access attempt and makes extra protection necessary. Distribution makes the risk greater still by hiding the perpetrators.

Obviously this investigation of policy cannot be exhaustive, but it does concentrate on some important texts. It is hoped that it is clear that these types of policy are common across a wide range of fields.

- **Security of Personal Medical Information**

  Access to personal information must be controlled strictly. However there are great benefits in having it widely and easily available because treatment is often not confined to one place and to one clinician. Hence groupware could help by bringing remote specialists together.

  Guidelines for the security of computerised medical information are published in a report by Anderson [AND96] in the form of a suggested security policy. The policy has dynamic elements. For example, the report states that some rights should be granted to groups of people and "the group might be any clinical staff on duty in the same ward as the patient". This type of policy could possibly be used to protect against the corrupt prescribing of drugs. Here a clinician's rights are not static, they will change according to the duty rota. If groupware is to be exploited for medical applications then such policies must be applied.

  Draper [DRA96] looks at the collaborative process of mental health care. As a safeguard against corruption, group decisions are always taken in situations where the patient is unable to give consent for treatment. This is another example of a dynamic right since a clinician's right to treat the patient (by updating the patient's treatment instructions) cannot be determined in advance. The group's opinion might change and hence should be re-sought periodically and in any case for every new treatment.

- **Security of Financial Information**

  Protection against corrupt employees is extremely important in securing financial systems. In all fields of financial security the principle of 'segregation of duty' (or 'separation of duty') is used to protect against malicious individuals. The concept demands that the responsibility for an action is shared between one or more other principals. Hence it would take the corruption and collusion of more than one employee to cause an irregularity. The principle of segregation of duty is described by Gray and Manson [GM89] as it is used in financial auditing.

  Kusner's lengthy descriptions of banking procedures [KA81] naturally cover security policies extensively and give many examples of splitting responsibility. Just one example states that a bank teller must seek permission to adjust a special-purpose bank account for reconciling differences between the actual amount of money taken over the counter and the recorded amount (should a discrepancy occur).

> The above texts were written without specific reference to computer systems, but it is obviously necessary that the principle should apply in this setting also. Many financial systems are designed according to Clarke and Wilson's [CW87] model for security. They claim that there are two mechanisms at the heart of fraud and error control: the "well formed transaction" and "separation of duties". An object-oriented system with scope for the enforcement of sufficiently dynamic policies could ensure that both of these mechanisms are achieved.

Anderson [AND96] points out that in a study of medical systems only a very small proportion of corrupt accesses came from external attacks and that all other breaches of confidentiality were from corrupt employees that have legitimate access to the information systems as part of their work. This shows that typically protection against internal corruption using good access control is critical to a secure system that has multiple users.

Access controls that ensure that a principal really is accessing information in the context of a legitimate activity are required. The brief examples above show the two common types of policy that are designed to ensure this.

First, the context of the activity is defined by the state of the task at the time of the attempted access. The state of the task is represented by the state of the very objects that are being protected. For example, the right to update the treatment record of a patient depends upon the state of the duty rota, where both the treatment log and the duty rota are necessarily protected objects. The context is also represented by the exact parameters of the attempted access. For example, even if the clinician is on duty in the ward of a patient it may be a desirable policy to restrict the quantities of a drug that is being prescribed.

The second type of policy is used to protect against corruption when it is impossible or unreasonable to deduce the context from some state. It is only going to be possible to automatically deduce if the context is legitimate from the data that is held within the system. However, often the context is not represented internally. To tackle this, security policies often require that the principal seek the involvement of another. This provides extra protection on the assumption that the corruption of two conspiring parties is far more unlikely.

The prescribing drug policy might more reasonably be stated using a combination of the two types. For example by stating that a clinician can only prescribe on a ward where they are on duty up to a specified maximum. To prescribe over the maximum, the clinician must seek the agreement of another clinician.

Chapter 8 will conduct a more thorough analysis including many more examples of dynamic access policies before introducing more flexible access control mechanisms.

2.3 Summary of Requirements

This section has introduced the two main themes of this thesis. Secure groupware needs:

1. Fast access to shared data.
2. Accurate representation and enforcement of security policies through dynamic access control.

The review of literature in the next chapter will show from experience of other research that providing sufficient performance when accessing shared data typically means that the data should be replicated. However maintaining replicated data means that the communications involving updates must be secured from the malicious behaviour of those external to the task, but who otherwise have potential access to the network.

In order to protect against internal corruption, many real security policies try to ensure that a principal's attempt to do something really is in the context of a legitimate activity. This leads to very dynamic rights since the context can change with every access attempt. This chapter has shown that some policies express the context in terms of the state of the system and the parameters of the attempt. However where this is not possible or reasonable, policies require that the principal seek the involvement of another principal or principals to exclude the possibility of one corrupt individual alone causing damage. It will also be shown in the next chapter that established methods of specifying access rights have little scope for the expression of dynamic policies.

3

Background

This chapter will examine the underlying implementation of secure distributed groupware systems. This will be analysed in the light of the two requirements that were demonstrated in the previous chapter, namely performance and adequate representation and enforcement of policies. It will examine many existing systems and show that although none adequately meets both of the requirements, there are lessons to be learnt about how each can be met separately.

In Section 3.2 it will be demonstrated that where performance is a key issue in a distributed system the problem is usually addressed by locating data close to where it is needed. If the data is needed by multiple parties then this means replication of the data. This however leads to complex problems regarding the maintenance of the data, but fortunately group communication abstractions can ease the burden. It will also be shown that where security and performance are issues then this just adds to the complexity of any solution, but again secure group abstractions do exist.

Security policies exist outside the computing realm and in many cases were enforced before computerisation of a task was even an option. Access control mechanisms are the means by which these policies are represented and enforced inside the system. Section 3.3 will start by looking at Lampson's established mechanisms for access control for distributed systems [LABW92]. Many existing systems build on this work to some extent. The section will go on to look at some of these systems and assess how well dynamic policies of the type introduced in the previous chapter could be expressed. If the mechanisms can merely represent a compromise of the exact policy then this can only be to the detriment of the security of a task.

First however, the next section will put the mechanisms of group communication (for performance) and access control (for adequate representation of policies) into context by examining an established architecture of secure distributed applications. Everything presented in this thesis will fit in with this architecture.

3.1 Architecture of Secure Distributed Systems

This section builds up to an outline for an architecture for secure distributed groupware. The outline is given in terms of the layers of abstraction from which a groupware system is constructed. The description first introduces the notion of a system being built out of multiple abstractions, each forming one layer in a stack. It goes on to introduce secure abstractions and replication.

Layers of abstraction eliminate difficulties of implementing reliable communication in distributed systems. A simple concept such as connected sockets makes applications easier to construct by removing worries about message dropping and uncertain orderings from the programmer's concerns.

Abstractions are built on other abstractions. TCP/IP [POS81a] [POS81b] is built on the Internet Protocol (IP) as its name suggests. Each layer removes some concerns from the layer above.

An abstraction such as TCP/IP relies on certain assumptions holding true. For example the guarantees of TCP/IP could not be met if messages were tampered with en route. To cope with malicious misbehaviour in addition to benign failures a secure version of an abstraction is needed. Secure versions very often offer the same view of the abstraction to the layer above as the non-secure counterpart, however fewer assumptions about the behaviour of the system are made. Secure versions of the TCP/IP socket layer do exist, most notably Netscape's Secure Socket Layer (SSL) [INT95]. Figure 3.1 shows an example stack of layers and its secure equivalent from the network layer up.

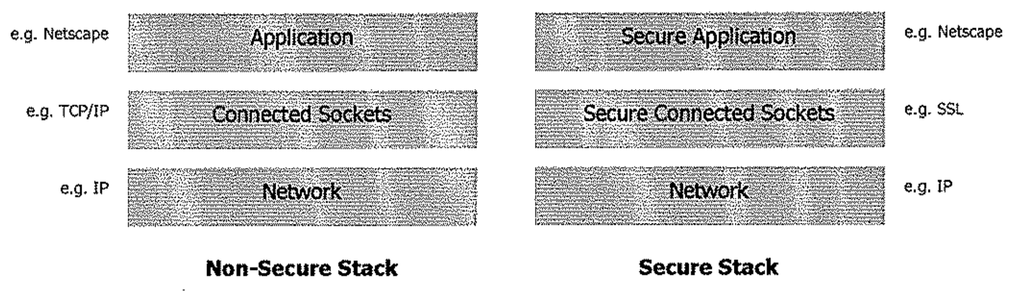


Figure 3.1 Abstractions offer guarantees to the levels above but are based upon assumptions about the behaviour of the system. Secure versions of abstractions rely on fewer assumptions about malicious behaviour.

At each layer of a secure stack the security guarantees are given appropriately for that layer, i.e. in terms of the abstraction. The socket layer is concerned with sending messages between two processes and hence the security guarantees of secure sockets are given in terms of messages and processes, for example: messages are guaranteed to have originated from a process run by a named principal.

Some application programmers use a remote procedure call abstraction which sits in between the application and the socket layer, for example Sun RPC [SUN90]. This is useful because it hides the communication altogether. More recently objected-oriented systems have an equivalent Shared Object layer, for example Sun's Java RMI [SUN96]. Conceptually an object is shared between multiple parties, any of which can invoke its methods. Complexities about locating the object and concurrent attempts to invoke its methods are hidden. Other more sophisticated systems that present this abstraction include Argus [LCJS87] and Arjuna [PSWL95] [SDP91]. This provides the abstraction of a single object shared by many processes. Each can independently invoke its methods as if it was a local object, but any changes to the state are visible to all. As above, security can be introduced at this layer too. This is illustrated in Figure 3.2.

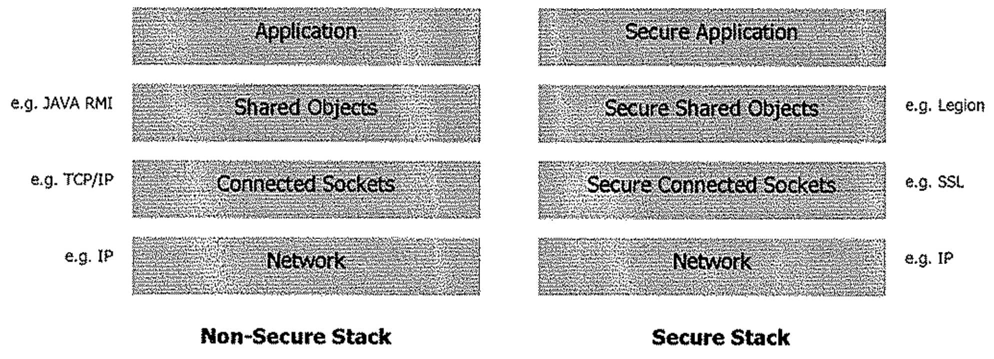


Figure 3.2 A Shared Object abstraction can hide communication (and associated complexities) from the programmer altogether. Secure Shared Object systems also exist.

As was the case with the Secure Socket Layer, the level at which security is presented to the user of the Shared Object layer must be appropriate to the level of abstraction. At the level of shared objects it is appropriate to specify security in terms of which principals can invoke which methods (method-level access control) -there is no notion of secure messages at this level. Examples of systems that protect shared objects at method level include PerDiS [CDKR97] and IBM's CACL [RSC92]. All the authors agree that access control is best suited at this layer because the layers below are too far away from the user's view of the system, whereas incorporating access control in the application potentially leads to inconsistencies in enforced policies if different applications share the objects.

Just as each layer relies on the communication guarantees of the layer below in order to maintain its own abstraction, so too each secure layer relies on the security guarantees of the layer below. For example, despite the fact that the level above is not aware of messages being sent, method-level access control can only be enforced if the messages containing the remote invocations are protected from tampering and replay. These are two of the guarantees of the Secure Socket Layer below (in Figure 3.2).

As will be shown later in this chapter, the key to good performance in groupware is replication of data. If data can be copied into the same address space as the application that uses it then access will be as fast as possible. However maintaining replicated data is complex.

For interactive groupware, updates made by one participant must be quickly communicated to the other replicas. Many systems use replication for other reasons (fault-tolerance or disconnected operation for example) and not all give much emphasis to communicating updates quickly. The gossip architecture [LLSG92] for example is used to enhance the availability of services. Updates are sent occasionally in batches. This simplifies the implementation and reduces the network load, but such an architecture would be unsuitable for interactive groupware.

The immediacy of communication needed by groupware can be supplied by multicasting updates to the replicas every time some participant initiates a change to the shared state. These updates are sometimes referred to as 'eager updates'. However updates to the state from multiple and potentially simultaneous sources must be consistently applied to the replicas.

Again an appropriate abstraction can remove these concerns from the programmer. A Group Communication layer (See Figure 3.3) can replace the point-to-point socket layer. Typically such a layer will offer the layer above a process group abstraction and mechanisms for consistently multicasting updates to the members. Additionally it might deal with processes joining and leaving the group and copying replicated state to new members. Notable examples include ISIS [BJ87] [BSS91] [BIR93] and Horus [RHB94] [RBFHK95].

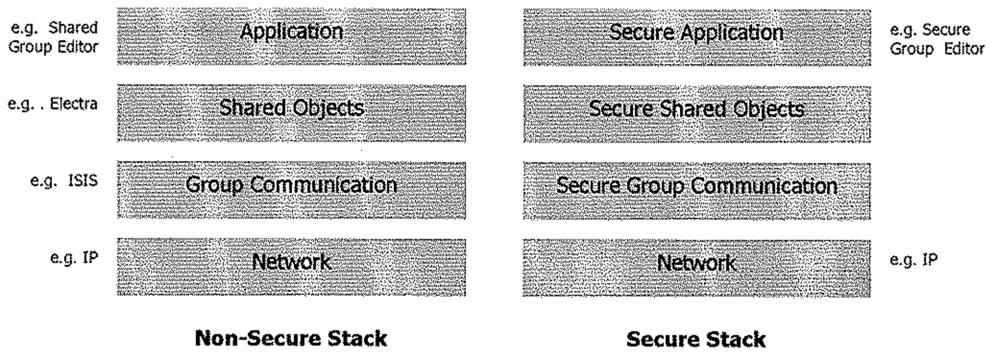


Figure 3.3 A Process Group abstraction can relieve the programmer of concerns about maintaining replicated data. Secure versions do exist.

A shared object layer can still make sense above a Group Communication layer. The concept as perceived by the application programmer is precisely the same, the difference being only in the underlying implementation. Now state can be locally replicated and hence access to it can be fast. This is a preferable solution for many reasons. Not only does replication lead to enhanced performance, but also such a system is more lightweight because there is no need for a server to maintain the object. This in turn means such a system is a more scalable system. Replication also allows for Fault-Tolerance. Examples of systems that provide a shared object abstraction with an underlying replicated architecture include Electra [MAF94] and Shared Objects [AK94] and these will be studied later on in this chapter. Likewise a secure shared object system could be implemented which provides method-level access control.

If existing applications are to be reused then security might not be present in the application layer at all. However, as long as it is built on top of secure layers then security policies, such as the ability to invoke methods upon objects, can be enforced at these lower levels of abstraction. In a well-designed application the top-level application objects and methods will mirror the real life counterparts that are referred to by any security policy. There must of course be a way of bypassing the application layer in order to inform the lower levels of security policy. A mechanism for this (a Security Shell) will be discussed in Section 3.3.2.

However there is no reason for security not to be built into an application from the outset if this is intuitive. For example the application could provide options to the user that ultimately change the underlying protection of methods and objects.

The two main themes of this thesis: providing fast secure groupware and providing dynamic access control are dealt with by the middle two layers of Figure 3.3 which are, as depicted in the diagram, notably void of examples. Initially the speed issue is tackled with group communication for quick multicasting of updates to replicated state. Then dynamic access control is handled at the level of controlling access to the methods of shared objects.

3.2 Secure Replication for Groupware

Replicating data locally at the machines of participants involved in a group task is the key to getting the desired level of performance for groupware. However if the group application is being used to aid a task that is security sensitive then it is important that any group communication software that is being used to implement the application is sufficiently secure. For example, updates to the data must be communicated in a secure manner that may involve encryption and authentication techniques being applied.

Replicating data is also used to provide fault-tolerance and it is with this goal in mind that most of the group communication systems (such as ISIS [BIR93] for example) are designed. So it is not surprising that the secure group communication systems that do exist were not designed with concern for groupware. This is a problem because the type of security that is needed for groupware is more complicated than simply protecting a replica group as a whole from outside malicious efforts.

Typically the participants in a group application have different rights from each other. This is necessitated by the different levels of trust that each holds. For example, security policy typically states that only certain participants can update the replicated state that represents the application. The exact nature of this Groupware Model of Trust will be studied in Chapter 4. For now however it is sufficient to bear in mind that communication (multicasts) that could contain updates to shared state must be identifiable as coming from a particular participant in order to be suitable for a secure group application.

Therefore secure groupware needs three things: replication, security and a design aimed at the groupware trust model. Unfortunately not many such systems exist. This section will review existing systems that meet at least two out of the three requirements and conclude that although there is much to be learnt from studying them, none of them as they stand is suitable for highly interactive secure groupware.

3.2.1 Replication and Groupware

Data can be replicated for three reasons:

1. Availability: A replicated service for example provides a choice if some should be slow or faulty.

2. Fault-Tolerance: Distributed systems with replication can cope with the loss of replicas as well as Byzantine type failure. The remaining correctly functioning replicas can continue to operate.

3. Performance: It is often not desirable to wait for data to be transferred from a remote location. If the data is replicated locally, obviously access will be faster.

It is performance that is the reason of interest for groupware. This section gives some examples of systems specifically aimed at groupware applications. The designers of these systems have all recognised that replication is the way to deliver the required levels of interactive performance. All examples provide the single shared object abstraction to the software layers above, whereas in reality the object is replicated locally at each participant's machine.

Shared Objects [AK94]

The Shared Objects system provides the groupware application programmer with the convenient abstraction of using a single object that is shared by all the participants in the group activity. In reality what the programmer instantiates and treats as the Shared Object is a proxy to a replica group.

Invocations that inquire the state of the Shared Object can be satisfied entirely locally through the local replica. Invocations that update the state (which are assumed to be far less frequent in applications of this type) must of course be communicated to all the other replicas in the group.

The system is built on top of the group communication system Horus [RHB94] [RBFHK95] that provides the necessary ordering to the updates as they are received. Many applications require an expensive total ordering of updates if it is essential that every participant is to observe the Shared Object progressing through exactly the same series of states. However some applications do not have such strong and hence expensive requirements. Take a shared phone book object for example, it is clearly not necessary for the participants to see new entries appearing in the same order. It is only important that all entries do eventually appear. A shared diary on the other hand might have a stronger ordering requirement.

Additionally the group communication system facilitates the transfer of state to new participants so that they may form their own replicas. Joining the group also involves communicating the new members' identity to all existing members in case this is important for the application, as it is with a group conferencing system for example.

Javanaise [HL97]

Hagimont and Louvegnies also recognise in their Javanaise implementation that groupware benefits from local replication of data and also that the groupware programmer's task is made easier if that replication is hidden behind the convenient shared object abstraction.

The objects are kept persistent on a Javanaise server. The application code that uses them is also stored on the server and both are transferred as Java mobile code and state to a participant's machine when required. Shared objects are grouped into clusters for efficiency and so are replicated at that granularity. Replication is in the form of a cluster cache that is maintained on each participant's machine.

This approach means that updates to shared object state initiating from an instance of an application are not necessarily communicated immediately to other replicas. Rather, replicas of caches are invalidated when an update occurs and only when an application instance spots this is a new consistent version fetched from the server. Javanaise would therefore benefit from group communication and thus eager updates. This would then make it more suitable for more interactive group applications.

Groupkit [RG97] [GR96]

Greenburg and Roseman's Groupkit toolkit is an extension to Tcl/Tk that allows real-time distributed groupware to be rapidly developed. The toolkit provides many common elements of groupware applications, most important amongst these being the 'shared environment'.

A shared environment is essentially a collection of objects that can be shared amongst the multiple participants of a groupware session. The shared environment can be centralised or replicated locally at the machines of the participants. The toolkit follows the Model-View-Controller paradigm of Smalltalk and so essentially the shared environment is the model, with each application instance having its own view and controllers.

Updates to a replicated shared environment are immediately communicated to all other replicas and so, unlike Javanaise, the toolkit would well support highly interactive distributed groupware such as shared whiteboards for example.

There are other examples of systems that provide the shared object abstraction: the Electra toolkit [MAF94], Mushroom [KIN96] and Orca [BTK90] all support replication. However, none of the examples mentioned in this section as yet provide any security options and so would not be suitable for use with a security sensitive task.

3.2.2 Security and Replication

Section 3.2.1 was introduced with three reasons for replicating data in a distributed system. Performance is a major benefit to groupware and all the systems outlined in the previous section took advantage of this. Fault-tolerance, although obviously important to distributed systems, is not a key issue to be dealt with here, however it is interesting to think about the parallels between fault-tolerance and security. Turn and Habibi [TH86] as well as Meadows [MEA] say that providing security is just coping with another class of failure. Fault-tolerant systems are designed with a model of failure in mind: that is assumptions about the behaviour and misbehaviour of the system. A security model of trust is the same idea but with fewer assumptions. Now it is assumed that components that are external to the system can exhibit misbehaviour also and with both internal and external components the failure can be Byzantine, i.e. malicious.

Replicated data can be easily maintained with group communication systems such as ISIS [BIR93] [BJ87] [BSS91]. Secure applications need secure group communication and so in this section two examples are studied: first a proposal for a secure extension to ISIS and second a system that uses a significantly different trust model. It will be shown that providing an appropriate trust model at the same time as maintaining performance is not straightforward. Both systems outlined here facilitate replication with fault-tolerance in mind and so together with security they provide for highly robust applications.

Secure Replicated Services [RBG92] [RB92a] [REI93]

As with both examples in this section, Reiter and Birman's studies into designing a secure extension to ISIS were aimed at providing replication for secure and available fault-tolerant services rather than for providing performance. The trust model employed is that of a set of trusted servers, i.e. a 'trusted island' of processes providing their service despite the malicious efforts of other external processes (which could include clients).

The architecture concerned is that of a set of component servers containing replicas of data. The service provided to one particular client is not provided however by one of the components alone. A single interaction with the service that does not change any state will involve contacting many components and comparing the results. This allows a proportion of components that have failed benignly to be tolerated. However use of replication in this way conflicts with local replication for performance reasons.

The secure group system gives the following guarantees as regards authentication:

- Group Multicasts are authenticated as coming from a group member.
- The group can authenticate new members. Permission to join must be granted by one existing group member.
- New members are able to authenticate the group.

Note that multicasts are authenticated as coming from a group member and not from any member in particular. This is achieved by signing communication with a group key possessed by all members and known by no parties outside the group.

Because of the potentially different rights of participants in a secure group task, authentication of communication for secure groupware however must guarantee that the source is a particular principal and not just a participant in general and this renders this system unsuitable for such uses.

This work carried out at Cornell was derived from work at Cambridge [GON89] directed at providing a secure and available authentication service. Independently, Kerberos [KNT91], a widely used authentication service, introduced replication of client key information to improve fault-tolerance. Since authentication is something that must occur at least at the start of every secure session then it is evident that the availability of the service is of paramount importance. It is clear that communication between components of the service must be secure since they contain security keys.

Rampart and Distributed Trust [REI94a] [REI94b] [REI94c] [REI96]

Rampart was also designed and implemented by Reiter. However Rampart utilises a more complex trust model than the previous work. Components of the replicated service are now no longer assumed to be incorruptible. The group communication system however is designed to allow the service to operate normally despite the corruption and hence Byzantine behaviour of less than a third of its components.

In particular the authentication guarantees are as follows:

- Group multicasts are authenticated as coming from a particular member.
- The group can authenticate new members. Permission to join must be granted by a proportion of existing group members.
- New members are able to authenticate the group.

This trust model and the authentication guarantees are appropriate for groupware, however allowing for a proportion of corrupt group members is expensive in terms of messages. A multicast is delivered to the application only after several rounds of messages have been sent to the whole group. This is to ensure that the same update message was sent to every member, since no assumptions can be made about the behaviour of the source. Additionally each of these messages must be authenticated with expensive public key signatures.

Rampart gives a multicast latency of about 73ms (on a SPARC 10) for a reliable multicast with a rather small 300 bit (modulus) RSA key. No figures are given in the literature for atomic multicast that is a likely requirement for many groupware applications. Nevertheless, at the very least an atomic multicast will take twice as long as a reliable multicast and in reality is likely to be greater still because ordering information for sets of multicasts are sent out only periodically. Unfortunately such long latencies are likely to render Rampart unsuitable for groupware.

Other systems that tackle the issue of security of group communication include Transis [DM96], SCOM [COO96] (a communication security layer for Horus), LSGC [MHP98] (whose main contribution is to performance by multiplexing groups) and Caelum [ACDK97]. However none of these systems offer anything other than the 'trusted island' model of trust.

3.2.3 Security and Groupware

All of the systems studied here provide security of network communication specifically with the needs of groupware in mind. However none are absolutely suited to the needs of highly interactive groupware where the participants act with different rights.

Secure Network Objects [DABW95]

The Secure Network Objects system provides secure communication of remote method invocations and results. Objects are created and maintained by processes run by the object's 'owner'. Specifically the guarantees of method invocation provide:

- Authentication of objects by clients.
- Authentication of clients by object server (owner).
- Protection of communication from eavesdropping, tampering and replay.

Clients and owners authenticate each other through trusted authentication agents that run on each machine. The agents set up a shared key that is used by both parties to sign method invocations (and the results returned) in order to prevent tampering. The possibility of replay is removed by including with each invocation message a unique sequence number that is returned by the owner with the results. If secrecy is required then the whole message is encrypted using the same key.

Just as the remote access is transparent, so too are the security mechanisms that are completely hidden from clients by the setting of a proxy in the clients' address space (here called a 'surrogate').

Lotus Notes [LOT] [HUT]

Lotus Notes is a groupware system in that it facilitates shared information for the purposes of collaboration. In fact Notes offers mechanisms for communicating with groups and coordinating group activities. At the heart is a document database that is held in servers. Documents are shared by clients and can be updated simultaneously by others.

A Notes system can employ replication in two forms:

- Server-to-Server: Notes is designed to be used across wide area networks and hence multiple copies of the document databases can increase access times as well as the availability of the data. It is possible to control the frequency that updates are sent to other replicas and so reasonably synchronous interaction can be supported.
- Client-to-Server: This is intended for disconnected operation, for example a salesperson who pays frequent visits to customers can take a copy of part or all of a database and access it whilst away. When she returns, updates made

to her copy are applied to the server copies and vice-versa. Notes flags any conflicting updates for manual resolution.

Neither of these forms of replication would be very useful for highly interactive groupware though. In fact the document structure of the data within databases renders interactive applications (such as shared whiteboards for example) difficult. Although fields within documents can contain structured objects, Notes only permits sharing down to the field level, i.e. two users cannot simultaneously edit an object within a field.

Notes also provides security of communication across networks. This of course includes the securing of all internal messages sent regarding the maintenance of replicated data. Messages are authenticated and made tamper proof through digital signatures and their contents can also be encrypted. Users and servers initially authenticate themselves to each other using RSA public key encryption supported with X.509 certificates [ISO88].

Persistent Distributed Store (PerDiS) [SKR97] [CDKR97] [CDR98a] [CDR98b] [CD94b]

The PerDiS system is currently under joint development by a consortium of European research and commercial institutions. The system is aimed at supporting group applications particularly but not exclusively in the CAD domain. For example the system might be used to support a group application for collaborative architectural design. The project aims to tackle a number of areas such as persistence, distribution, concurrency, fault-tolerance and security in a transparent and scalable way.

The system presents a shared memory abstraction to the application programmer. This is a broader idea than the shared object abstraction that enabled a wider range of existing applications to be easily converted for use over PerDiS.

For availability, fault-tolerance and performance, it is necessary to replicate data that is currently being worked on by some participant. This results in the familiar problems of maintaining replicated data that have been demonstrated by other systems in this chapter. Replication in PerDiS is local to the clients, i.e. on their machines, however the replicas are not updated after every update applied by other users. Only the local updates are reflected immediately. Occasionally the new state of a segment of memory is brought up to date, not by distributing updates, but by distributing a new version of the segment.

Security of communication is also implemented. When data is copied to form a new replica, the data can be authenticated as coming from a legitimate source. Optionally the data can be encrypted for its passage across the network. When updates are made to the state, they too must be authenticated and optionally encrypted by the source of the change. Other participants currently working on the same shared memory data then only apply the new version to their replicas if the source is a legitimate participant and not some malicious party attempting to apply false updates to the secure state.

PerDiS can also control access to the shared memory. Section 3.3.2 will return to this system again in order to study its more fine-grained access control mechanisms.

Enclaves [GON96]

The Enclaves group communication system developed by Gong at Javasoft provides a secure group abstraction intended for use by groupware applications. An Enclave is a 'protected virtual subnet', or in other words a 'trusted island' of participant processes.

Groups are initiated and maintained by a group leader. This leader is responsible for authenticating prospective new members and for communicating their arrival to the rest of the group. The leader in fact communicates all 'essential' group information to the others such as keys and the initial state. However this does not include multicasts to updates to the state, which come from the members themselves and are authenticated with a shared group key. Consequently there is no basis for building on access control as it is impossible to securely establish from whom within the group an update originated.

The Caelum [ACDK97] group communication system is another example that uses this model for groupware. As already stated, the 'trusted island' model of trust is not suitable for secure groupware where the participants have different rights. Nonetheless access control is important for many secure tasks.

3.3 Access Control for Groupware

The groupware performance requirement discussed in the previous section is provided for primarily at the communication layer. This section concentrates upon the next layer up, which in an object-oriented system is the Shared Object layer (see Figure 3.3).

This is the level at which access control is naturally implemented. There is no notion of fine-grained operations at the communication level below and it is desirable to alleviate the application programmer from the burden of implementing access control mechanisms. Indeed, if existing applications are to be reused and made secure then separating access control is essential.

As with the communication layer, security at the Shared Object layer is presented at a level consistent with the communication abstraction. Hence access control is expressed in terms of the ability to invoke methods upon objects.

First Section 3.3.1 explores the conventional methods of specifying rights as defined by Lampson. Many secure distributed systems use these techniques as their foundation. In the following section some systems that have more sophisticated methods of specifying rights are examined. However despite their various innovative features, none of them offer total support for the kind of dynamic policy introduced in Chapter 2.

3.3.1 Established Methods for Specifying Rights

Lampson originally set out his 'access control model' in 1974 [LAM74]. Since then, the ideas have been widely adopted in many systems. Later in 1992 the ideas were updated in order to be more relevant in a distributed context [LABW92], but the overall concepts remain unaltered.

The access control model for distributed systems embodies the following entities:

Principal:	The source of actions and the unit of trust. Principals can be people or system components such as services.
Process:	Principals are represented in the system by processes. Generally a process acts on behalf of one principal.
Request:	An attempt (originating from a principal) to perform some action.
Protected Object:	The resource under protection, for example: files, devices and services.
Access Control List:	(ACL) The form in which the security policy is represented in the system. It is a list of principals associated with each operation of an object.
Guard:	The component that enforces access control.

The model encompasses the notion of a trusted computing base (TCB). This is a set of processes that are assumed always to operate according to their specification. As will become clear, all systems must be built with a trusted computing base since it is impossible to build trust out of nothing. In general however, making it as small as possible minimises the risk of the incorruptibility assumption breaking. Figure 3.4 illustrates how the entities of Lampson's model are applied to a secure object-oriented server.

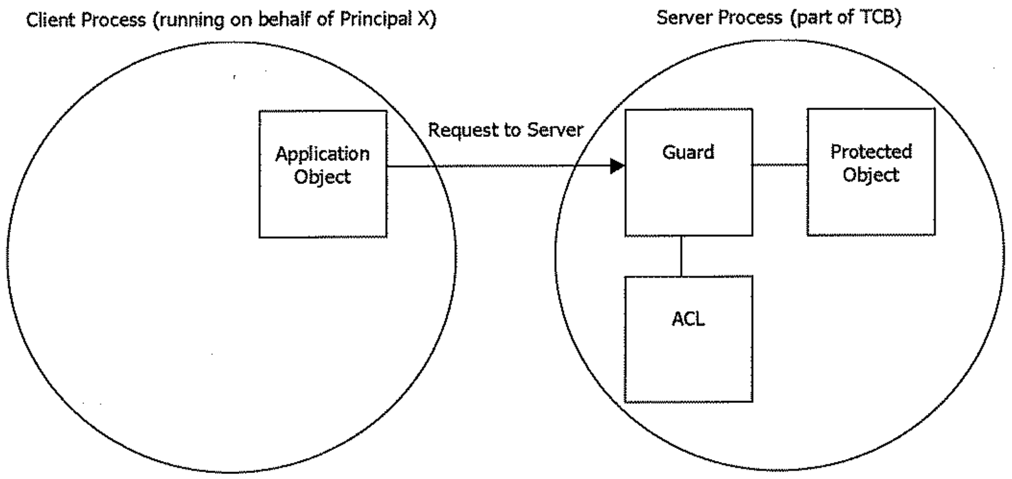

Client Process (running on behalf of Principal X)
Server Process (part of TCB)
Application Object
Request to Server
Guard
Protected Object
ACL

Figure 3.4 The Guard intercepts requests from clients. It consults the Access Control List before handing on the request to the protected object.

The specification of access rights could be a laborious task if the number of principals and objects is large (as is typically the case in distributed collaborative applications). Lampson's updated model introduces methods of simplifying the task of specifying rights.

Simplification of rights specification is accomplished by allowing generalisations about principals. Principals can be grouped according to the role that they play in some task and then rights are assigned to the group. For example all doctors could be given the same right to access a medical record. Later it will be shown that other schemes have simplified rights specification further by allowing for generalisations to be made about objects.

The security policy can only be expressed as accurately as the access control list permits. A simple list of principals or roles can only describe very static policies such as "Principal X can do Operation Y". As Chapter 2 made clear, policies are often more dynamic. Lampson's model does include the notion of delegation. This allows one principal to empower another with some or all of their rights dynamically, without intervention from some authority.

Most multi-user operating systems use the access control model to some extent. The UNIX file system for example, although not allowing the specification of access rights on an individual basis, allows rights to be granted on a group basis. Lotus Notes studied in Section 3.2.4 uses a combination of system-defined roles and access control lists specifying rights to invoke generic document oriented operations (read, write, delete and compose).

The TAOS operating system [WABL94] implements the entire model. The system provides an API that can be used by the programmer of a service for example, to check the authenticity of a principal and verify rights claimed through delegation or role membership. Principals must be provided by the application with a way of delegating and specifying role membership and access rights. This does mean that the system could not easily be introduced into an existing application. The following section will introduce a solution to this problem also.

Rights are packaged with the necessary proof (signed certificates) into credentials which are a data type used as arguments in the security API. Access control lists are also represented as data types with functions for checking a principal's presence on the list and of course for modifying them. These functions allow the implementation of principal naming to be kept hidden and to allow the procedures of authentication and checking of access rights to be transparent to the user of the application. All communication between machines running TAOS is authenticated through the use of secure channels implemented using shared keys to sign digital signatures [NS78].

3.3.2 New Access Control Techniques for Groupware

Lampson's access control model offers little if the policies are dynamic in any way, that is rights cannot easily be changed without intervention as some collaborative task proceeds (delegation is the exception). Changes in rights will require a change to be made to the access control list and this is only achieved through the intervention of some authority. This is inconvenient and probably far from immediate and consequently not ideal for interactive groupware which relies upon fast responses to actions for its productivity.

There are some systems that offer more flexible access control however and this section will explore some of them, namely: Legion, CARDS, Intermezzo and PerDiS. In particular these systems are aimed at group applications that cater for a greater degree of dynamic policies to be expressed.

Legion [LG95]

Like some of the systems studied in the previous section, the Legion system also offers the application programmer a shared object abstraction. However Legion is intended to be a fully functioning and practical system rather than merely a research project. Consequently issues such as scalability, object location, object naming, persistence, fault-tolerance and access control all feature to some extent in the implementation.

When not in use by any application, Legion objects exist in persistent storage somewhere. An application wishing to access the object must obtain a reference that the Legion communication layer can use to locate a host that maintains the object. Once a connection has been established, communication of remote method invocations and results can start.

Access control is specified consistently with the shared object abstraction, i.e. rights are specified in terms of the ability to invoke the methods of shared objects. Legion takes a very simple and open approach to rights specification by ensuring only that a specific boolean method (called 'MayI') of a shared object is called before access is granted. The application programmer implements these methods and it is their returned value that determines whether the attempt to invoke a method succeeds or not.

This approach means that the potential policies that can be specified are wide and rights can obviously be more dynamic since the 'MayI' method can refer to such a variety of criteria including system state. The down side of the Legion approach however is two-fold. First it places more burden on the application programmer because he is probably forced to implement an access control list himself. Secondly and more seriously, a change in security policy or even possibly an individual's change in rights will result in application code having to be altered and rebuilt. The flexibility of Legion access control is good, however not having it tied in with application code would result in a more maintainable system.

Context Authentication Service for Role Based Access Control in Distributed Systems (CARDS) [HT95]

Most secure systems give more assistance to the programmer and supply the implementation of access control. The CARDS system is an authentication service and not a layer as such in a secure system. Applications can invoke the service on behalf of a principal and gain signed statements of their rights. These statements can then be presented across the network (to a service for example) in order to gain access to some resource. The rights obtained are not static however as they would be if they were read directly from a static access control list, i.e. successive claims from CARDS could result in different rights being returned. The rights returned to clients can reflect the current state of a collaborative task and therefore can more accurately express need-to-know policies by more concisely restricting the circumstances under which a principal can do something.

Tasks are divided into phases and separate rights can be assigned to principals in each of these phases. Assigning a principal's roles to a specific phase and then only permitting the principal to take on this role during that phase facilitates this.

Intermezzo [EDW96]

Edwards presents a specification language for defining policies using a system called Intermezzo. These policies are more flexible than simple access control lists because they are evaluated at run-time. This means that what users can do changes dynamically as the collaborative task evolves.

Intermezzo is not designed to be a highly secure system, rather, Edwards is just one of many authors that recognises that collaboration needs to be controlled in order to be effective. He gives the example of a shared drawing tool -the scope for interference is great and if interaction is not constrained in some sense then users will be "*overwhelmed by the task of baby-sitting the environment rather than getting work done*".

Amongst others, the following two policies are given as examples:

> *"Don't let anyone bother me when I'm working on my thesis. Unless it's my advisor of course."*
>
> *"I need to share my workspace with others during demo days."*

These both represent policies that couldn't be expressed in a conventional access control list since both refer the current state of an activity, i.e. what is currently being worked on and whether it's a demo day or not. These policies can only be evaluated dynamically. Policies are expressed in a language that allows the state of objects to be consulted. Object methods can be called directly and the results returned used in the evaluation of rights.

Access to objects cannot however be controlled on a per-method basis. Access is granted only to the generic operations Read, Write and Exist (to test if an object exists). This is perhaps strange in a system whose contribution is to widen the range of policies that can be expressed.

It is easy to see that the specification language could be useful in a secure system. The example policies could easily be security policy if in the second example, the workspace contained private information at times other than demo days.

There are many other systems that use roles and policies to control potentially destructive collaboration, such as Suite [SD92], MPCAL [GS86] and PREP [NKCM90]. Also there are other systems that recognise that for accurate enforcement of security policies, the state of the system must be taken into account when evaluating a principal's rights. Apart from CARDS and Intermezzo, such systems include Schumann Security Software's SAM [HIL97] and work by Moffett and Sloman [MS91]. Additionally work by Hayton on the Oasis system [HAY96] also recognises that rights are dynamic. As well as introducing a language that allows state dependent control of rights, the system also extends the notion of delegation to recognise that the concept is broader than just the transfer of existing rights, but that new rights can also be created as they are in elections for example.

Persistent Distributed Store (PerDiS) [SKR97] [CDKR97] [CD94b]

The PerDiS system (already studied in the previous section) doesn't offer anything beyond Lampson's access control model in terms of ability to specify and enforce more dynamic security policies. However it is interesting because it is aimed specifically at groupware applications and it offers a very immediate way in which participants of a group activity can change rights.

The layers in a secure system offer security at a particular level of abstraction and offer an appropriate interface to their security functions to the layer above. For example a secure communications layer offers security in terms of secure messages. A layer above uses this to offer its secure abstraction that will typically offer a completely different model of security to the layer above it, for example protection of shared object methods. Finally the application layer will offer its own view of security to the users which might be completely different again.

This presents a problem with securing existing applications that do not currently offer any security functionality. They cannot easily be rebuilt on top of the secure equivalents of the underlying layers because new functionality will need to be built into the application code in order to be able to control the security features.

The PerDiS system was specifically designed with existing non-secure CAD applications in mind and offers a solution to the above problem. The system includes the notion of a security shell that is a means of bridging the gap between the secure shared object layer and the user and thus bypassing the application which can therefore remain security ignorant, but secure nevertheless. All security commands such as changes to access control lists, delegations and role-taking are dealt with by the security shell. See Figure 3.5.

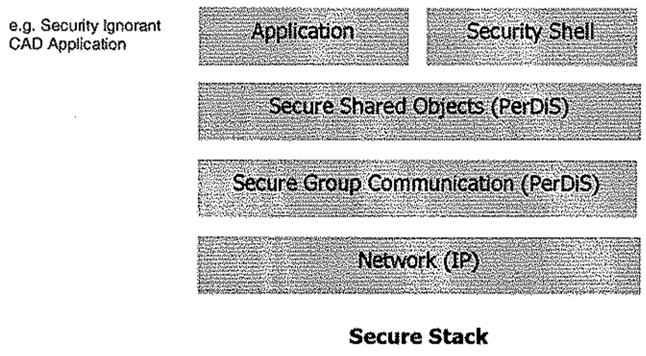

e.g. Security Ignorant CAD Application
Application
Security Shell
Secure Shared Objects (PerDiS)
Secure Group Communication (PerDiS)
Network (IP)
Secure Stack

Figure 3.5 The Security Shell of PerDiS relieves the application from supplying an interface to security features, thus allowing existing 'unsecured' applications to be reused with minimal modification.

It does mean of course that the user has to deal with security and specify rights at the level of shared objects, i.e. controlling access to methods. However a well designed application should have intuitive objects names that mirror those of the real-life counterparts and the object methods are assigned to generic categories that have user-understandable names such as 'read', 'write' and 'edit'.

PerDis also includes the notion of 'tasks' that are used to partition the scope of access control rights and objects in order to make the system more scalable and ease the complexity of assigning many rights to many people for many objects. The task concept is drawn upon by the work presented in this thesis in Chapter 9, Section 9.4.2 and is described in further detail there.

3.4 Summary of Groupware Security

This chapter has attempted to demonstrate that although many systems exist that address security and groupware issues, there is none that entirely addresses all the needs of highly interactive secure groupware, particularly where participants act with varying rights.

Replication and eager updates can address the speed needed for effective collaboration and the systems studied in Section 3.2.1 (Shared Objects, Javanaise and Groupkit) proved that this can be achieved whilst hiding the replication from the application programmer behind a convenient shared object abstraction. However none of them addressed security as an issue.

Secure replication has been addressed by other systems, two of which were covered in Section 3.2.2 (Secure Replicated Services and Rampart). However neither of those were specifically aimed at interactive groupware and so were inappropriate in the trust model they assumed, which is a little more complex for groupware (Chapter 4 will explore a model of trust for groupware further).

The four systems outlined in Section 3.2.3 did address security for groupware specifically, however each of those too was inappropriate in some way. Secure Network Objects didn't use replication, Lotus Notes and PerDiS didn't use replication appropriately for highly interactive applications and Enclaves again assumed an unsuitable trust model.

At a higher level, access control mechanisms have existed for a long time and have been adapted for distributed systems. However the range of policies that it is possible to express using an access control list is limited. Some schemes have been proposed for making access controls more dynamic and thus allow for a greater range of policies. Section 3.3 examined some of these (Legion, CARDS and Intermezzo). However none of these were aimed at the specific needs of groupware.

These gaps will be filled in the remainder of this thesis. Chapters 4 through to 6 will deal with providing secure group communication for groupware and Chapters 7 to 9 will deal with dynamic access control for groupware.

4

Secure Group Communication for Groupware - Design

The previous chapter showed that the group application programmer's task can be made easier by using system software that provides a shared object abstraction. Several examples of secure abstractions were given such as Secure Network Objects [DABW95]. However none of them took advantage of replication in order to make them fast enough for really interactive group applications. The chapter continued by demonstrating that locality of data and hence replication of data is at the heart of performance.

The facilitation of replication and hence the shared object abstraction can be far more easily implemented with the help of a group communication system. Some secure group communication systems that facilitate replication have been discussed but it was argued that none absolutely suited the needs of groupware. Some systems made inappropriate assumptions about the nature of trust within a group of participants (for example Secure Replicated Services [RBG92]) whilst other approaches were simply too slow for groupware (such as Rampart [REI96]).

This chapter offers a solution to the problem of making secure interactive groupware fast enough for constructive work to be done. It presents the design for a secure group communication system that is intended for secure groupware applications. The intention is that this system could be used to provide replicated state to a group of participants and facilitate the immediate communication of any updates to that state to the replicas, the updates being securely multicast to the group member processes.

Section 4.1 starts with the observation that all secure systems are founded upon some trust assumptions holding true. No system can operate in the absence of all trust. This trust takes the form of processes that are assumed always to operate according to their specification, in other words they never become corrupted. These processes form the trusted computing base [LABW92].

Replication in groupware is different from other types of replication because data is replicated upon the machines of ordinary users, i.e. the participants in a group application, rather than replication in servers. This in turn means that a group communication system is being used in a different trust environment. The secure group communication systems that were studied in the previous chapter are intended for the implementation of secure services, where the replicated data remains safely under the control of trusted processes that provide the service. Placing data under the custody of ordinary participants' machines means that the trust assumptions made by the above group communication systems are no longer valid.

Section 4.1 is concerned with discovering what are appropriate trust assumptions. It will be shown that there are compromises to be made since the ideal is actually impossible to implement.

Section 4.2 will make concrete exactly what is the full set of requirements from a secure group communication system for groupware. Most obviously there are communication requirements. These shape the abstraction that is presented to the application. There is no reason why this should be any different from the process group abstraction of existing systems that have been shown to effectively support the implementation of distributed systems that replicate data. Then there are some security requirements such as authentication of updates to replicated data and of course it is required that the abstraction remain intact despite some potential malicious activities. Finally there is the all important performance requirement.

4.1 A Model of Trust for Groupware

When designing a secure system it is important to be clear about exactly what type of malicious behaviour the system will withstand. This is in line with the approach used by designers of a fault-tolerant system. For fault-tolerance, assumptions are made about which processes in a system will adhere to their specification and which can deviate from their specification and in precisely what way.

These entities and assumptions form a model that can be reasoned about. A system is designed and implemented according to the model. It will function as expected in any situation in which the assumptions of the model are valid. The same approach can be used to design a secure system. The basic entities of the access control model (processes, principals, requests, etc.) are taken and assumptions about their behaviour are made. For example "Guards always function correctly". These form what will be referred to here as a **Model of Trust**.

There is a difference between a failure model for fault-tolerance and a model of trust however. In a model of trust it is essential to include every principal that can have any potentially malicious effect on the system. This includes principals whose processes aren't even part of the system. This is a vast number if the system is to be implemented over a wide area network such as the Internet. In practice the assumptions will be blanket statements covering many principals such as "all those involved in the system are assumed to behave according to their specification" and "all those not involved are not assumed to behave in any way".

When considering whether a process can be trusted (assumed always to execute according to its specification) it is necessary to recognise that a process will possibly be running software from many different sources. In fact even the hardware could be exposed to sabotage. Trusting a process and any communication from it is not merely trusting the principal upon whose behalf it is running, it is trusting all the software and hardware as well. The level of trust given to a process must take all these factors into account. Typically the level of trust afforded would be the lowest common denominator.

Thankfully the least trusted component is typically the user and so for the models of trust described here a process is afforded the same level of trust as the principal on whose behalf it is running.

Finally it is important to realise that trust is always from someone's point of view. It is not something that can always be applied to a system from above. This is especially true of groupware systems that could involve cross-organisational collaboration. Participants in one organisation might trust each other completely and the others not at all. Whereas those in another organisation trust themselves but not the first group. Hence the assumptions must make clear where the trust is coming from.

4.1.1 Example Models of Trust

Most existing secure systems are designed to very simple models of trust. For this reason the model is not often explicit in the literature. For example the model that the Secure Socket Layer is designed in assumes that the two communicating processes trust each other, i.e. each assumes the other is uncorrupted (see Figure 4.1). All other processes are not trusted at all, with the possible exception of a certification authority. This mistrust includes all the processes involved in transmitting the message such as gateways and bridges. Any system that employs secure channels is adopting this model.

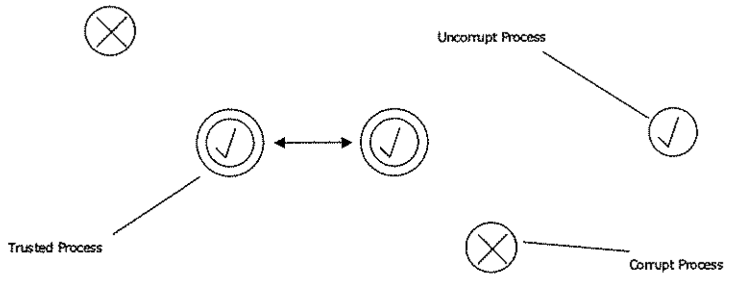


Figure 4.1 The Secure Channel Model of Trust. The two communicating processes trust each other but no other process. The communication abstraction is maintained despite the potential corruption of the other processes in the network.

Of course just as with fault-tolerance, if the assumptions become invalid then the guarantees of the system are likely to fail. If the Secure Socket Layer software becomes corrupt then messages may not be transmitted securely.

Chapter 3 introduced some secure group communication systems that maintained a group abstraction securely, even in the presence of some malicious behaviour. These examples employed different models of trust. The Secure Replicated Services system [RBG92] employed the simplest model of trust. This was a simple extension to a group, of the model used by the Secure Socket Layer. In this 'trusted island' model all processes that are members are trusted equally and totally by each other (see Figure 4.2) and all other processes are not trusted at all.

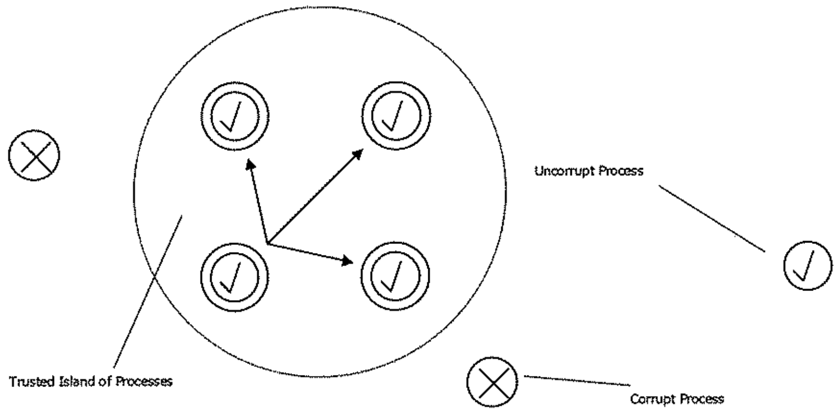


Figure 4.2 The Trusted Island Model of Trust. All member processes trust each other, but do not trust any other processes that are assumed to be potentially corrupt.

If just one group member becomes corrupt then the abstraction can fall apart for all members. All other external processes can misbehave in any way (by for example attempting to masquerade as legitimate members) but cannot disrupt the group abstraction.

4.1.2 An Ideal Model of Trust for Groupware

It is straightforward to implement a secure group communication system in the simple 'trusted island' model of trust introduced above. All that is needed is for communication between group members to be authenticated. As long as a recipient believes that a message does originate from another member then the message (an update to replicated state for example) can be acted upon without further checks, because all other members are assumed to be trustworthy.

This model is probably appropriate for groups of servers that run on machines that don't host user sessions directly. These machines can bar interactive logins from all but trusted administrative staff and they can be physically located in locked rooms. Other similar measures can increase the probability of the assumptions holding and therefore the processes forming the group remaining uncorrupted.

It is not necessarily that simple for a group application. For a start the processes that are group members are located on users' workstations. This is necessary to get replicated data located in the same address space as the application and hence achieve the maximum performance gain. These processes will obviously be more open to corruption than the server processes mentioned above. This 'trusted island' model seems unacceptable when bearing in mind that it takes just one member to become corrupt in order to destroy the whole abstraction for the others.

Secondly and more fundamentally, the principals of a secure group application are likely to be acting with different authorities. Several studies analysing secure group tasks (both computerised and non-computerised) have shown this to be the case [SD92] [CD94a] [EGR91] [GS86]. Very often in situations where it is important to adhere to some policy, some people are given more rights than others. Just a few of many examples are: the chairman and the board of directors; the manager and other staff; consultants and junior doctors; lecturers and students.

Applying a general model of trust to the infinitely diverse array of different levels of trust afforded to members in a diversity of possible applications can be difficult or rather restricting. It is not possible to assume that any member is totally trusted, i.e. has all rights. So the best that can be done that is generally applicable to all possible convolutions of trust is to make no assumptions about trust at all. This is what will be referred to as the 'ideal' model of trust for groupware and is depicted in Figure 4.3 below.

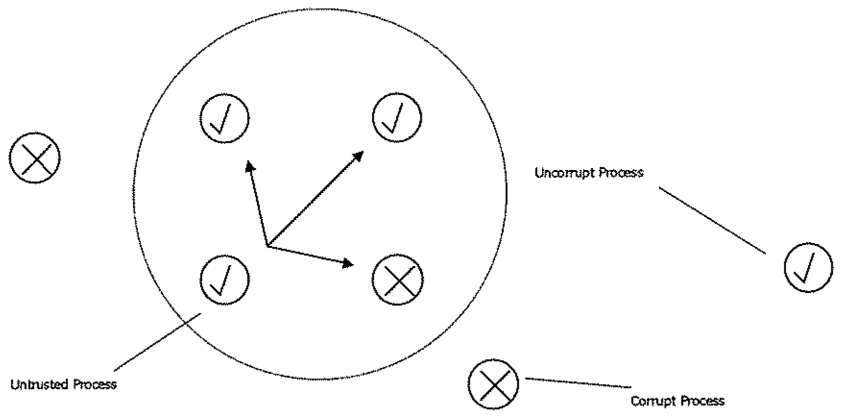


Figure 4.3 The Ideal Model of Trust for Groupware. No assumptions about trust of group members are made. It is considered possible that some group members will become corrupt.

If no assumptions about any processes' behaviour are made then the possibility of all the member processes becoming corrupt must be catered for. Clearly it is impossible to maintain any sort of abstraction under these circumstances. In fact the situation of providing reliable multicasts in the presence of arbitrary behaviour is an incarnation of the Byzantine Generals problem [LSP82]. Lamport et al. have proved that this is impossible if a third or more processes become corrupt. Hence the ideal model of trust for groupware makes it impossible to implement the system.

4.1.3 A Practical Model of Trust for Groupware

Clearly if the ideal model makes the system impossible to implement then it will have to be compromised in some way if the goal of replication for groupware is to be realised. Some group member or members must be assumed trustworthy. There are two clear ways forward.

The Rampart [REI96] secure group communication system introduced in Chapter 3 takes one possible approach. This is to assume that at least two-thirds of the group members are uncorrupted and functioning according to their specifications. Under these circumstances it is possible to maintain the group abstraction to these trusted processes.

Rampart like the other group communication systems studied previously was designed for the implementation of a replicated service and not for groupware specifically. For this reason, the performance of the system was not an overriding issue.

Rampart implements reliable multicasts in the described model of trust through 'echo multicasts', each of which involves three sets of messages. First the data intended for distribution to the group is sent to everyone by the originating source. Following this everyone echoes the message with a digital signature back to the source. When the source receives enough echoes it sends all the signatures back to everyone. This proves to all that everyone else got the same multicast and the source didn't try to disrupt the consistency of replicated data by sending different updates to different members. This is a distinct possibility since the source of multicasts (the group members) is not trusted. At least two-thirds are trusted, but it's not known which ones these are.

The implementation in this model of trust makes Rampart very slow. The high numbers of messages together with the need to digitally sign each with slow asymmetric keys makes Rampart unsuitable for interactive groupware. In fact the description above is only for the reliable multicast implementation. Totally ordering them is even more complex and doubtless slower still although no timing figures for totally ordered multicasts are given by the author. Reliable multicasts take in the region of 73ms to be delivered to the application (this is for a 1KB multicast). A totally ordered (atomic) multicast would take much longer than this (at least double) since the sequencer type architecture means that separate ordering multicasts (themselves reliable) are sent out.

The group communication system for groupware applications presented here takes a different approach and employs a new more practical model of trust. Rampart is expensive because the source of a multicast cannot be trusted to send the same message to everyone. If however the source can be trusted then this problem is eliminated. If a multicast can be authenticated as coming from a trusted source then all group members will maintain their replicas consistently.

This does not necessarily mean trusting the ordinary group members. An ordinary group member wishing to initiate a multicast can first forward the data to the trusted member of the group. This member actually emits the multicast which will be (under the practical model of trust for groupware) correctly formed and emitted. This is shown in Figure 4.4 below.

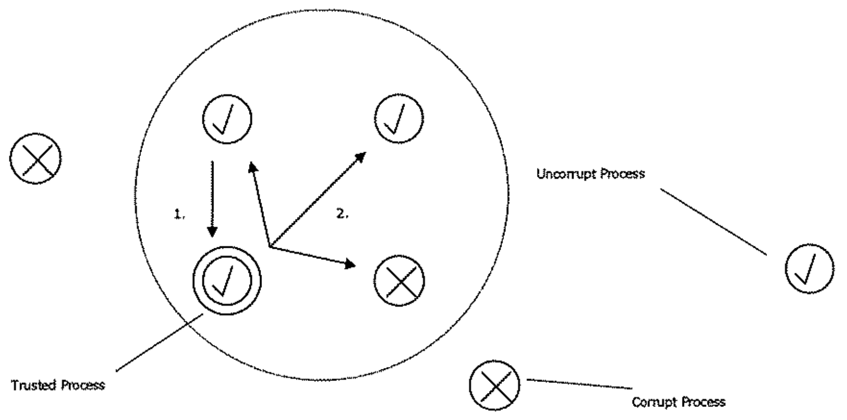


Figure 4.4 The chosen Practical Model of Trust for Groupware. A specific group member is assumed always to remain uncorrupted.

Obviously the model is weaker than the ideal, since the corruption of the one trusted process could result in the collapse of the abstraction. Therefore it is vitally important that this member is protected from corruption. Since all malicious corruption ultimately stems from human interaction then this process could run on a machine that does not allow interactive logins. It would be a group member that maintains replicated data in the same manner as its untrusted counterparts, but it wouldn't interact with a user directly. Additional physical steps could be taken to further reduce the possibility of the incorruptibility assumption breaking.

The model is stronger than the 'trusted island' model because a system implemented under those assumptions could not withstand the corruption of even a single process, no matter which one. Also as will be demonstrated, a system implemented in the practical groupware model of trust would be fast and appropriate for data replication at users' workstations and hence appropriate for interactive groupware.

4.2 Requirements of Group Communication

This section deals with the communication, security and performance requirements that secure groupware demands from a group communication system. Another set of requirements that would ordinarily be necessary for a practical groupware system would be for fault-tolerance. Given that the group members in the system are processes running on users' workstations it should be deemed likely that member processes will fail. However this research is directed at assessing whether the security requirements can be met in a sufficiently speedy implementation, as a consequence benign process failure is not addressed. Secure group communication systems such as the Secure Replicated Services system [RBG92] use communication time-outs to determine when a process has failed. Such implementations prove that fault-tolerance can be built into the abstraction. Further these implementations show that, with the exception of Rampart, fault-tolerance does not impact severely on the performance of the system when it runs normally. Only when a process fails is there a performance degradation. Even faster implementations of group communication do exist, however such systems do not typically offer the same levels of fault-tolerance. Amoeba [KT91] is one such example.

4.2.1 Communication Requirements

The communication abstractions offered by most group communication systems are very similar. Most offer means of joining and leaving groups, receiving a replica of state from an existing member and of updating the state by multicasting to the group.

However the details of the communication guarantees do differ subtly, particularly in relation to the order that multicasts are delivered to an application. A network does not generally provide any guarantees that the order in which multicasts are made is the same as the order in which they are delivered. Indeed two processes on the Internet could emit a multicast containing an update to replicated state at exactly the same point in time. A distributed application may require that these updates be applied to the replicas in the same order everywhere. Such a guarantee is not necessarily cheap and if the application is dealing with secure data then security measures may slow the implementation down further.

One of the most widely used group communication systems is ISIS [BJ87] [BSS91] [BIR93]. This system offers a choice of abstraction. Each choice differs according to the ordering guarantees that it provides, the intention being that the programmer chooses the cheapest one that satisfies the needs of a particular application.

The most expensive of the ISIS abstractions is known as Close Synchrony. This provides an approximation of an actually synchronous system where group multicasts and membership changes happen atomically and are observed in the same order everywhere. It is a simple execution model for the programmer, but providing the total ordering of multicasts and membership changes is expensive. ISIS Virtual Synchrony is similar but does not guarantee the total ordering of events. Instead a cheaper causal ordering is implemented.

ISIS has been shown to be widely useful for the implementation of distributed systems and as a consequence the group communication system for groupware gives guarantees similar to ISIS Close Synchrony. The communication guarantees are therefore as follows:

C1. When a process joins a group it receives state and the current membership list.

C2. Following this it receives updates to the state that are consistent with the original state and are totally and causally ordered with respect to the updates received by all the other members. All correct processes receive all updates.

C3. When a process receives notification of another process joining or leaving the group, this is consistent with the notification received by the other processes, i.e. notification of membership changes are also totally and causally ordered with respect to the multicasts in between.

ISIS would also guarantee that the failure of a process is also communicated to the group. However as already stated, this fault-tolerance is not the key purpose of this implementation and so process failure will not be catered for. The implementation will however cope with communication anomalies such as dropped or repeated messages.

Another more subtle difference between the communication guarantees of the system presented here and that of ISIS close synchrony is the impossibility of ensuring that a process delivers and acts upon all the messages that it should. ISIS would guarantee that a member process either successfully delivers a multicast or its failure is reported back to the group. This is necessary for applications that use the group to distribute work to member processes for example. Since in the system presented here it is not assumed that an ordinary member is functioning correctly, it is not possible to say whether it processed a message or not. However for applications that use group communication merely to distribute updates to replicated state (as is the case for groupware) then this guarantee is not important, i.e. if a member chooses to ignore or incorrectly process updates to state then this behaviour can only affect itself.

4.2.2 Security Requirements

The practical model of trust for groupware has shown that the participants in a groupware system are not generally completely trusted. The group application might be enforcing some access control policy and this in turn implies that the participants are afforded different levels of trust. In order to be generally applicable therefore the group communication system should not make any trust assumptions about the participants. In other words the correct functioning of the system should not rely upon the correct functioning of an individual participant's process.

This gives rise to two types of security requirements. Firstly, one that ensures that the group abstraction remains intact despite the malicious behaviour of ordinary member processes (excluding the trusted member). Secondly those requirements that the layers above the group communication system need to enforce some access control policy.

The first security requirement is simply:

S1. The communication model (C1 - C3) will remain intact at all member processes that are uncorrupted.

It is obviously impossible to make such a guarantee to a process that has become corrupt, for example one that is running a corrupt version of the software providing the abstraction. However this is not a concern. It is only desirable that the correctly functioning processes (i.e. those that adhere to their specifications and hence adhere to the group communication protocol) see a view of the group that is consistent with the other non-corrupt processes.

Requirement S1 obviously implies that certain steps be taken in the implementation in order to prevent a malicious process from disrupting the group abstraction. These steps are in addition to ensuring that one group member always functions correctly. Such mechanisms include preventing group messages from being replayed or tampered with and manifest themselves in the implementation as digital signatures attached to messages. The implementation is described in full in the following chapter. The remaining requirements described in this section are to support any attempt by an application to enforce access control.

Different participants may have different rights. It is therefore necessary to be able to identify the exact source of a communication to the group. This is in contrast to group communication systems that are built according to the 'trusted island' model of trust in which it is only necessary to identify a communication as coming from a group member in general. Hence S2:

S2. Multicasts can be authenticated, i.e. not tampered with or replays of earlier communication and identifiable as coming from a particular group member.

It is further necessary to give guarantees to the new member regarding the authenticity of the group that it is joining. Additionally the group must be aware of who is joining.

S3. A prospective member can authenticate the group that it wishes to join.

S4. A group can authenticate a prospective member and refuse admission.

Requirement S3 ensures that the new member really joins the group that it is intending to join and not some malicious party's attempt to masquerade as the group. S4 provides the same assurance to the group so that admission can be refused. First thoughts may deem this unnecessary since S1 requires that no matter who gets into the group they cannot disrupt the view of the other correct members. However group members receive state and this state might be secret such as personal medical or financial information.

Admission policy is application dependent and so we assume that the group communication system can obtain details of which new members are permitted to join a group from the level above. A secure shared object layer for example might be able to make a decision of whether or not to allow admission based upon an access control list. It would be sensible to disallow admission if the participant did not have rights to invoke all the methods that inquire state, since a successful join would result in state being transferred to the participant's machine.

Given that applications can enforce access control then it is possible for rights to change. Hence a current member of a group may lose its right to be a participant so it follows that there should also be a means by which a member can be excluded. Hence the next security requirement.

S5. A group can evict an existing member.

In our system there should be no assumptions made about the processes that run on behalf of principals outside the group. In practice using existing broadcast network technology it is infeasible to prevent any machine from viewing an update message. If collaboration is taking place across the Internet then machines such as gateways that are involved in the transmission of messages are typically outside the jurisdiction of an organisation. If malicious these could glean secret information from a message. This must be prevented, hence the secrecy requirement S6.

S6. All state transfer and multicasts are useful only to members (i.e. encrypted).

This secrecy requirement may not always be necessary for every application and therefore will be an option in order to avoid the expense of unnecessary encryption.

4.2.3 Performance Requirements

Performance is why the group communication system is used in the first place. Consequently the performance requirement is the most fundamental and needs the least additional explanation. Chapter 2 has already covered the reasoning behind this requirement. However for completeness:

P1. Group communication must be sufficiently efficient for interactive groupware.

4.3 Summary

This chapter has presented a practical model of trust for secure groupware systems, together with communication, security and performance requirements. The group communication system for groupware must be implemented in the groupware model of trust. Hence all of the requirements must be met in a potentially malicious environment and one in which no assumptions are made about the behaviour of ordinary participants. The next chapter explains how the requirements are realised in the implementation.

5

Secure Group Communication for Groupware - Implementation

The previous chapter introduced the practical model of trust for groupware. It was a compromise of the ideal model because that was impossible to implement, however it does allow for a fast implementation. Every group has a trusted member process. As long as this one member remains uncorrupted the implementation described in this chapter will consistently present the group abstraction to all of the correctly functioning (non-failed, non-corrupt) members. This is despite the potential corruption and Byzantine behaviour of any proportion of them. Because of the role of the trusted member process in distributing multicasts, this member will be referred to in this description of implementation as the **Distributor**.

In this implementation, every principal is initially authenticated to others using an asymmetric key pair [RSA78]. It is assumed that there is some authority (or authorities) that principals trust to create certificates for these keys. These certificates could for example be X.509 certificates [ISO88] or of the Simple Public Key Infrastructure (SPKI) of Lampson et al [LYRFET98].

The members of the group are processes. It was stated in the previous chapter as part of the requirements that every member must know the identity of the other members (requirement C1 from Section 4.2.1). This implies that members must be uniquely identified. Other group communication systems use operating system process identifiers coupled with the machine name to uniquely identify the members. Whilst this is also the case in this system, recipients of group communication need also to be able to identify the source of the multicast in terms of the principal that sent it (requirement S2 from Section 4.2.2). It is not desirable to exclude the possibility that a principal have two or more member processes in a group, consequently a principal's identity alone will not uniquely distinguish the member. Hence the group members are processes that are identified by a process identifier, a machine identifier and the identity of the principal on whose behalf the process is running.

In Section 5.1, secure multicasts are examined and individually-identifiable multicasts (II-Mcasts) are introduced. These are used as the basis of group communication and are in turn used to implement secure group membership.

Section 5.2 discusses some of the details of ensuring secure group membership, particularly the authentication of the new member and the group it wishes to join. The prospective member can easily be identified using conventional asymmetric cryptographic authentication techniques. How to authenticate the group however is less obvious, but can be achieved using group authentication certificates. Additionally because of the secure nature of the applications that the system will be used to implement, there has to be a way of restricting group members to those who are entitled to work on the collaborative project. How decisions are made regarding who to admit into the group is a topic discussed in Section 5.2.2.

Finally the last section shows how the system is divided into layers in order to make the various semantics optional. This keeps the functionality of a group to the minimum required by an application that it supports, thus making it as light-weight and hence as fast as possible.

5.1 Secure Multicasts

The security requirement S2 (Section 4.2.2) states that a multicast must be identifiable as coming from a particular named principal. This is in contrast to most other group communication systems that merely identify the source as being a group member in general. This requirement is in tune with the practical model of trust for groupware.

This section will describe the implementation of Individually-Identifiable Multicasts (II-Mcasts). Despite the potential corruption of the source of the multicast and other group members (apart from the Distributor), the multicast will come with guarantees of:

Authenticity:	Its Stated Source will be proved using strong encryption (S2).
Reliability:	All correctly functioning (i.e. non-corrupt, non-failed) processes will eventually receive the multicast (C2).
Total Ordering:	Multicasts will be delivered to the application in the same order to all correctly functioning processes (C2).

Optional Secrecy: The data contained within the multicast will be encrypted (S6).

Obviously it is impossible to make any guarantee that a corrupt member will deliver the multicast. This isn't a matter of concern however, for as long as the sub-group of correct members all observe the same multicasts in the same order then they can all see their shared data progress through the same states.

The reliability guarantee means that this implementation will deal with communication failures, i.e. dropped messages, but not process failures (most notably the benign failure of the Distributor). Although as stated in the previous chapter, the existence of other group communication systems such as ISIS prove the viability of recovering from a group member failure.

5.1.1 The Distributor

An II-Mcast consists of two messages. All multicasts are initially unicast to the Distributor. The Distributor then IP Multicasts the information to the rest of the group. This ensures that every group member receives the same message. This is ensured because the Distributor is trusted by all the other group members to do its job properly.

The Distributor also caches every multicast that it forwards. This means that it can re-send the data to any member that did not receive it because of a communication failure.

Every communication (both unicasts and multicasts) contains a sequence number which serves three purposes:

1. Sequence numbers serve to order the messages. In this way the method of ordering is similar to that used by Kaashoek for the Amoeba system [KT91]. Rampart [REI94c] and ISIS [BIR93] also use sequence numbers for totally ordering messages, however this is achieved in slightly different ways.

2. Sequence numbers allow for the detection of dropped and repeated messages due to network anomalies.

3. Sequence numbers allow for the detection of maliciously replayed messages.

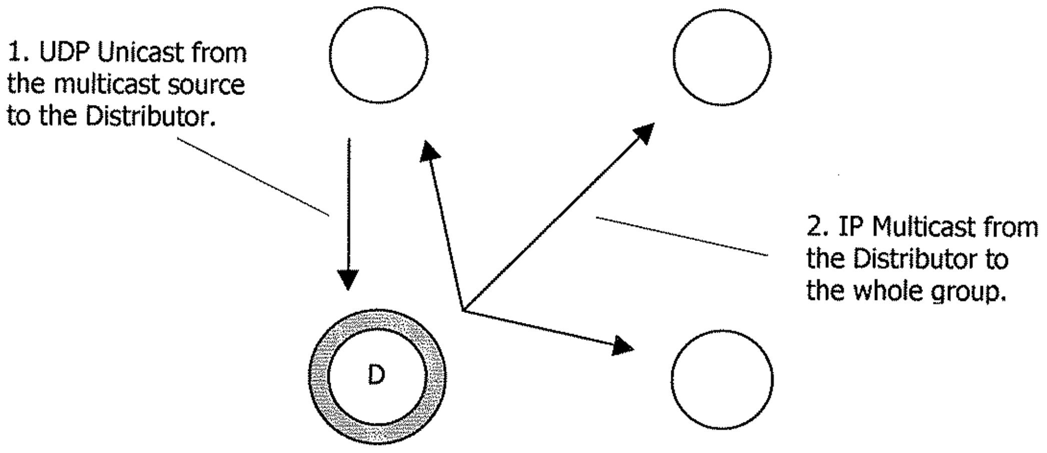

1. UDP Unicast from the multicast source to the Distributor.
2. IP Multicast from the Distributor to the whole group.
D

Figure 5.1 An II-Mcast consists of two messages. It is initiated by the source with a UDP Unicast to the Distributor (D). This is then forwarded in an IP Multicast from the Distributor to the rest of the group (and back to the source).

The Distributor includes the identity of the original source of the II-Mcast in the message (message 2 in Figure 5.1). The rest of the group will only accept and deliver multicasts that come from the Distributor, i.e. only ones that form part of a valid II-Mcast. In order for the recipients to believe the original source and that the II-Mcast really comes from the Distributor, both unicasts and multicasts are authenticated. The mechanisms for authentication are discussed in the following two sections.

5.1.2 Authenticating Unicasts

Unicasts are simply authenticated using a symmetric shared key that will be referred to as an **Individual Member Key**. Every member establishes a different Individual Member Key with the Distributor as part of the group join protocol. The protocol for this is described in detail in Section 5.2. Group communication through II-Mcasts are far more frequent than group membership changes. Hence using symmetric key encryption for authentication of multicasts is far preferable to slow asymmetric key encryption. The Individual Member Key is used to digitally sign the multicast initiation message (message 1 in Figure 5.1).

Periodically it is wise to refresh the key in an attempt to reduce the possibility of a successful cryptographic attack. A new key can be invented by a member and this is piggybacked (but incorporated into the signed body) in a multicast initiation message to the Distributor. This suggestion for the new key is itself encrypted in the Distributor's public key for privacy.

5.1.3 Authenticating Multicasts

Individual Member Keys are established between the Distributor and every other process in the group. This effectively establishes n - 1 secure channels where n is the number of processes in the group. These channels are also used by the Distributor to forward the multicast the group.

Perhaps it would be more obvious to authenticate the multicast (message 2 in Figure 5.1) using a signature signed by the Distributor's private key. This however would slow the II-Mcast down unnecessarily. Faster symmetric key encryption can be employed if the Individual Member Key is used for authentication. This doesn't necessarily mean that multiple messages must be sent out to the group members. Instead of attaching signatures to individual messages signed with the Individual Member Keys, all of the signatures can be attached to one multicast message. This will be referred to as a **Vector Signature**.

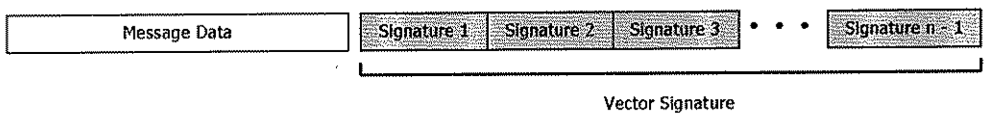


Figure 5.2 A Vector Signature consists of n - 1 components. Each component is a conventional message digest encrypted with one of the Individual Member Keys.

The Vector Signature (see Figure 5.2) has n -1 components (where n is the number of members in the group), each of which is a standard encrypted message digest of the message data. Each component is encrypted by the Distributor using one of the Individual Member Keys and attached in turn to the message. The whole message is then sent in one IP multicast to the group. Each recipient then extracts the component signature that was signed using the key that it shares with the Distributor. It verifies this signature using the key and ignores the other components of the Vector Signature.

Timed tests were conducted on this scheme to prove how much faster than conventional asymmetric key authentication the Vector Signature scheme really is. These tests established that the expense of asymmetric key encryption was out-weighed by the smaller expense of multiple symmetric key encryptions and longer messages in the Vector Time Stamp scheme for group sizes of up to approximately 1800 members. Group sizes of this magnitude are unlikely for groupware applications that use one group for every shared object. Chapter 6 has full details of this and other timed tests conducted on the system.

5.1.4 Encrypting Multicasts

Requirement S6 (from Section 4.1.2) stated that multicasts must be encrypted if required by the application. The Individual Member Keys cannot be used for this purpose since they are not known by all of the members and neither should they be as this would defeat their purpose. Hence a different key is used for encryption. This will be referred to as the **Group Encryption Key**.

The key encrypts the whole message except for the Vector Signature. The Group Encryption Key is also established as part of the group join protocol which is described in Section 5.2.

5.1.5 Multicast Protocol

This section describes the exact content of the two messages comprising an II-Mcast (depicted in Figure 5.3). Each process in a group is assumed to run on behalf of one named principal and so the members are referred to in the descriptions by that principal. The descriptions use the following notation:

Members	
M1, M2, M3, etc.	Ordinary Group Members.
D	The Distributor.

Keys	
PuM1, PrM1	The Public and Private Key Pair of M1.

M1 ↔ D	The Individual Member Key Shared Between M1 and D.
GEK	The Group Encryption Key.

Messages	
M1 → D: M	M1 sends Message M to D.
D → G: M	D sends Message M to the Group.
{ M } K	Message M is Encrypted with Key K.
[M] K	Message M is Digitally Signed with Key K.
[M] D ⇔ G	Message M is Vector Signed with all the Individual Member Keys that D shares with the members of Group G.
SN	A Sequence Number.

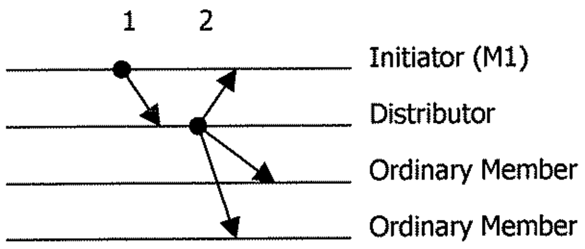


Figure 5.3 Two messages comprise the group II-Mcast. One from the initiator of the group communication (M1) to the Distributor and one a multicast from the Distributor to the group.

II-Mcast 1: M1 → D: [{ IIMCAST1, M1, data, SN } GEK] M1 ↔ D
Some group member (in this case M1) initiates the II-Mcast with a message sent to the Distributor (D). This message contains a sequence number in order that replays can be detected. Replays from a previous period of M1's membership are not possible because a new key M1 ↔ D is established at every join.

II-Mcast 2: D → G: [{ IIMCAST2, M1, data, SN } GEK] D ⇔ G

The data is then relayed to the group (G) by the Distributor (D). It is authenticated using the attached Vector Signature.

The sequence numbers in II-Mcast 1 and II-Mcast 2 are different since every source maintains individual counters for communication to a particular destination. In addition the Distributor maintains a separate counter for multicast messages to the group. A new member obtains the current value of the multicast sequence number from the Distributor as part of the group join process. This is explained in the next section.

5.2 Secure Group Membership

Individually Identifiable multicasts only account for some of the communication and security requirements from Section 4.2. Additionally there must be a method for securely joining a group, receiving state and informing the existing members of the new addition. This is all achieved with the secure group membership protocol that is described in this section.

5.2.1 Group and Prospective Member Authentication

When a process joins a group it is important for that process to be able to authenticate the group in order that some malicious party could not dupe the process into joining some spoof group. In the practical model of trust for groupware the Distributor is always trusted. Hence authenticating the group amounts to no more than identifying the Distributor. In the first message of the group join protocol, the new member effectively challenges the Distributor to authenticate itself by signing the return message with its private key.

This just leaves the problem of tying the Distributor's public key with a group identity. This is achieved by the certificate depicted in Figure 5.4. This is very similar to a public key certificate, the main difference being that the group certificate contains additional details about the group as well as the public key of the Distributor and the group name. The group details allow the new member to be sure of the correct configuration and semantics of the group that it is joining. This is discussed further in Section 5.3 as it relates to the layered structure of the group system. The name of the group can be any string that uniquely identifies the group to the members.

Group Name	IP Multicast Address	Group Details	Public Key	Signature

Figure 5.4 A Group Certificate contains the group's identity, its associated IP multicast address, details of the group's structure, the public key of the Distributor all signed with an authority's private key.

When a group is first created the authority that initiated it designates a Distributor and produces the certificate (note that a single process can act as a Distributor for many groups). It signs the certificate with the authority's private key and distributes the certificate or makes it available in some way to all who might need it. It could for example be placed in a certification service.

Authentication of the new member is achieved by the Distributor checking that the prospective member has possession of the private key of the principal that it claims to be. The second message that the new member sends in the group join protocol contains a signature.

Every group must have its own IP multicast address so that group communication can be confined only to interested parties. The group's IP multicast address is included in the group certificate to enable a new member to locate the group. Before the prospective member has any initial contact with the group it cannot possibly know the location of any of the current members. Hence the IP address extracted from the certificate enables the first contact to be made.

5.2.2 Joining and Leaving the Group

A new member that is permitted to join the group becomes party to information that is only available to group members such as the Group Encryption Key and any state that is transferred. There must therefore be a mechanism in place for restricting membership of the group. These decisions are based upon any security policy that is in force and hence cannot be made by the group communication system because access control does not belong at this level. The decision is deferred up to the software layer above that is closer to the application and hence closer to the level at which security policy is specified. Additionally security policy might change. This means that the instruction to evict a member must also be passed down from the layer above.

However the question remains of which group member should make these decisions. Secure Replicated Services [RBG92] for example allows any existing member to admit another, whilst Rampart ensures that over two-thirds of existing members all agree before the new member is admitted. These requirements are different because of the different models of trust that each system is built to exist in. The obvious correct answer for the practical groupware model of trust is that the Distributor should make the decision to admit the member. However this decision does deserve some further explanation.

Every member already in the group has a version of the group state. They could potentially give it away to anyone at any point. Given that becoming a member of a group only gives the new member state and not necessarily any ability to update it, then why shouldn't any member be able to admit another? Part of the task of admitting the new member is informing the rest of the group of the new member's arrival (requirement C3) and this is a crucial part of maintaining the abstraction and therefore should not be under the control of ordinary members. The whole group join protocol is described in the following section.

5.2.3 Group Join Protocol

The join protocol consists of four messages. Upon completion the new member has received the current membership of the group, any replicated state, has authenticated the group and is ready to start receiving multicasts. The group in turn is aware of the new member and has authenticated it. Figure 5.5 captures the four messages that comprise a group join.

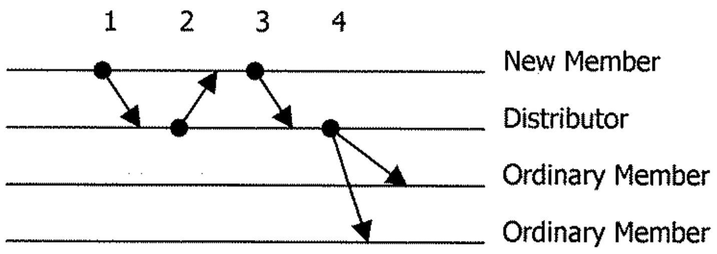


Figure 5.5 Four messages comprise the group join protocol. Three messages are exchanged between the new member and the Distributor resulting in message 4 which is a multicast to the group informing the other members of the new member's arrival.

JOIN 1: A → D: JOIN1, A, Nonce

The first message of the join protocol of course comes from the prospective member and is directed to the Distributor. The nonce is a randomly generated number that will prevent some malicious party from replaying the join messages from the Distributor and hence duping the prospective member into thinking it has joined the group when in reality it hasn't even communicated successfully with it.

JOIN 2a: D → A: [{ JOIN2a, Nonce, SN, M'ship, State, A ↔ D, GEK } PuA] PrD

The second message is the Distributor's response to the join request. It contains all the information that the new member needs in order to start receiving multicasts. This includes the sequence number of the next multicast, the current membership list, the Group Member Key (A ↔ D) that is invented by the Distributor and known only by the two parties. All this is encrypted in the new member's public key so that it can only be usefully interpreted by the intended recipient A. The message is authenticated by the inclusion of a signature signed with the Distributor's private key. The group membership key is changed after any member leaves the group. In fact it must also be changed as a new member joins (see NEWKEY message below), in order to stop that member from decrypting previous group communication exchanged before it became a member. That member's rights could have been different then and information could have been exchanged that was never intended to be seen by the member.

JOIN 2b: D → A: [JOIN2b, Nonce] PrD

If the Distributor has to reject the request to join, then the authenticated rejection above is sent as an alternative to 2a.

JOIN 3: A → D: [{ JOIN3, A, A ↔ D } PuD] PrA

The third message is an acknowledgement of the shared key proposed in the previous message. This message is necessary in order for the Distributor to believe that the prospective member has received the key. It is also the message that authenticates the new member to the Distributor since JOIN 1 is not actually signed at all.

JOIN 4:   D → G: [{ JOIN4, A, SN } GEK] D ⇔ G
Finally the Distributor informs the group of the new member and includes the new member's identity in the message. The message must be totally ordered with respect to multicasts and so has the same format as the II-MCAST 2 message, i.e. vector signed with a sequence number. The existing members include the new member in their views of the group from this point onwards and the new member will receive all future multicasts.

NEWKEY:D → G: [NEWKEY, { newGEK } A ↔ D, { newGEK } B ↔ D, etc., SN] D ⇔ G
As part of integrating the new member the Distributor must inform the group of the new Group Encryption Key. The format of the New Key message is shown above although when the key is being installed as part of the group join procedure this message can be piggy-backed onto JOIN 4. The key is included in the message several times, once for each member, each encrypted with the appropriate Individual Member Key. Upon receiving this message, members will discard the old key and start decrypting future multicasts with the new key.

LEAVE 1: A → D: [LEAVE1, A, SN] A ↔ D
When a member decides to leave the group, it informs the Distributor with a leave message. This has the same basic form as the II-MCAST 1 message.

LEAVE 2: D → G   [LEAVE2, A, SN] A ⇔ G
The Distributor informs the group with a vector signed multicast. LEAVE 2 also has a NEWKEY message piggy-backed onto it. As in JOIN 4 and NEWKEY, this message has the same format as the II-MCAST 2 and hence the sequence number in the LEAVE 2 message comes from the same counter as the Distributor's multicast sequence number (as does the sequence number in EVICT below).

EVICT:	**$D \rightarrow G$ [EVICT, A, SN] $A \Leftrightarrow G$**
An eviction message is identical to the LEAVE 2 message, where in the case shown, A is the member being evicted. This is also how the evicted member learns of its fate. Of course EVICT has a NEWKEY message piggy-backed onto it which does not contain an element for A.	

An analysis of this protocol using the logic of authentication devised by Burrows, Abadi and Needham [BAN90] is included in Appendix A. This proves that the group join protocol meets the authentication goals that were set out in Section 4.2.2 on system Security Requirements.

5.3 A Flexible Group Communication System

All of the member processes run the same code, the only difference with the Distributor member is that the code runs on a more trusted machine in a more trusted environment. The code has been implemented in layers, with each layer adding part of the total semantics of the group. This follows the approach of the Horus group communication system [RHB94] and is used in order to make the system more lightweight. The intention is that only the communication semantics that are needed by the application are included. This is extended in the secure group communication system for groupware by allowing the application layer to choose only the security layers that are required by the application. This is achieved by removing or adding different layers, which allows the practical model of trust to be changed from the groupware model to a cheaper alternative (such as a 'trusted island' model) if this is appropriate for the setting in which an application is being run.

The system when configured to run for the practical model of trust for groupware, with all the security and communication semantics described in Section 4.2, consists of seven layers. Figure 5.6 shows all the layers and can be considered an expansion of Figure 3.3.

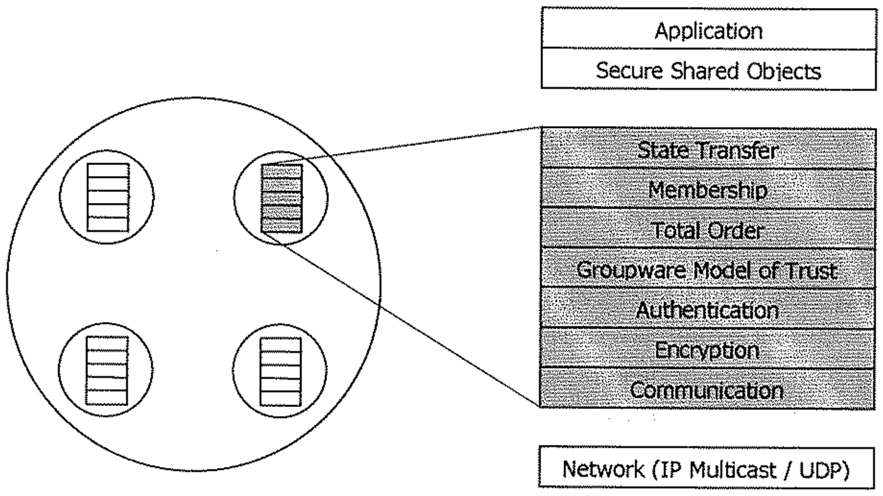


Figure 5.6 The Group Communication System for Groupware is layered (layers shown shaded). Each layer adds something to either the communication or security semantics of the whole.

The layers can potentially fit together in any order because all share a common interface of down-call methods (which move down from the application end of the stack) and up-calls (which move up from the network end). However not all configurations make sense. The layers form a partial order because the semantics of some of the layers rely upon guarantees of layers below. For example the Membership layer that delivers the membership changes to the group members relies upon a totally ordered reliable multicast which are the guarantees of the Total Ordering layer. Every layer relies upon the Communication layer at the bottom that handles the group multicast address and translates the incoming communication into up-calls and the down-calls into outgoing messages.

Each down-call has the message (as it exists at that stage) passed down to it from the layer above. Each layer has the opportunity to add to the message. The message constructed by the Join 2a down-call for example would have the membership list placed in it by the Membership layer and the Group Encryption key would be inserted by the Encryption layer. When the message reaches the bottom of the stack the Network sends it out as either a UDP or IP Multicast message. In a complementary fashion, after the message is received at its destination and converted by the Communication layer into an up-call, each layer is responsible for removing (and interpreting) from the message the elements that were inserted by that layer at the message's source.

Security is represented in three layers: Groupware Model of Trust, Authentication and Encryption. The model of trust layer is responsible for diverting the multicasts to the Distributor and hence ensuring that the same multicast is delivered to all incorrupt members despite the potential corruption of some of the group.

Unsurprisingly the correct functioning of this layer relies upon the Authentication layer .

The Authentication layer is responsible for maintaining the members' public keys and calculating and verifying message signatures. In the Distributor the Authentication layer calculates Vector Signatures. During a group join phase, this layer invents the Individual Member Key. In the Distributor this layer would store all the keys that are shared with the ordinary members. In the processes of those ordinary members there is only the one key to keep track of.

The Encryption layer is chosen if an application is dealing with sensitive information that must be kept secret whilst being communicated. This layer in the Distributor creates the NEWKEY message and in all members this layer responds to this message when it is received. This layer invents the Group Encryption Key and inserts it into the Join 2a message during a group join.

Obviously all members of the group must have the same stack of layers as every other member of that group. For this reason the group certificate that was introduced in Section 5.2.1 contains the sequence of layers that the group is using. This is contained within the "Group Details" section of the certificate in Figure 5.4.

Further details of an implementation of this system written in Objective C over the NeXTSTEP operating system are contained within a technical report [RD97a]. Additionally there is a paper that was presented to the Second European Research Seminar on Advances in Distributed Systems [RD97b]. This whole system has also been implemented in Java and it is this version that was used to obtain the timing figures that are presented in the following chapter.

6

Evaluation of Secure Group Communication

The security and communication requirements of the secure group communication system for groupware have already been demonstrated in Chapters 4 and 5. Hence the main task in evaluating the effectiveness of the system for implementing interactive groupware applications is assessing the remaining requirement. That is the performance of the system.

This chapter is divided into two sections. Section 6.1 will present the timing figures for the system, measured for a range of group sizes. It will be shown that the multicast latency is indeed of the right order for groupware. Section 6.2 will examine more detailed timing figures that allow an evaluation into the effectiveness of Vector Signatures to be performed. The original set of timing figures are compared against figures obtained from a version of the system that was modified to use more conventional asymmetric key authentication. It is shown that for all reasonable group sizes the Vector Signature scheme out performs conventional public key authentication.

Finally Section 6.3 discuses two other important issues that might need to be addressed before the system could be used in a fully practical setting, namely: fault-tolerance and scalability.

6.1 The Speed of Multicasts

When measuring the performance of the group communication system, it is the times for multicast latency that matter. Participants in a group task will tolerate a small delay when joining a group. However multicasts containing updates to replicated state are far more common and as has been shown already in the examples discussed in Chapter 4, delays in receiving updates will damage the productivity of collaboration.

In this section multicast latency is measured and assessed. This is the time that elapses between the initiation of an Individually-Identifiable Multicast (II-Mcast) and its delivery to an application. Every II-Mcast that is generated must pass through the Distributor, hence the multicast can be timed entirely at one machine even though it is being delivered to as many machines as there are group members. The clock is started when the message is handed to the group communication system and stopped when the system delivers it back. In this time the system will have generated and sent two network messages (II-Mcast 1 and II-Mcast 2), one from the source to the Distributor and one IP multicast that travels back to the original source and to the group as a whole.

Figure 6.1 presents multicast latency for a range of group sizes. Two graphs are shown: one for a zero data size and one for 1000 byte data sizes. The data size represents the amount of data given to the system by the layer above. The actual message size will be longer due to the additional information that the group communication system appends to messages (such as sequence numbers and signatures for example). A group size of two is the lowest sensible size from which measurements can be taken, as a singleton group will not result in any network communication.

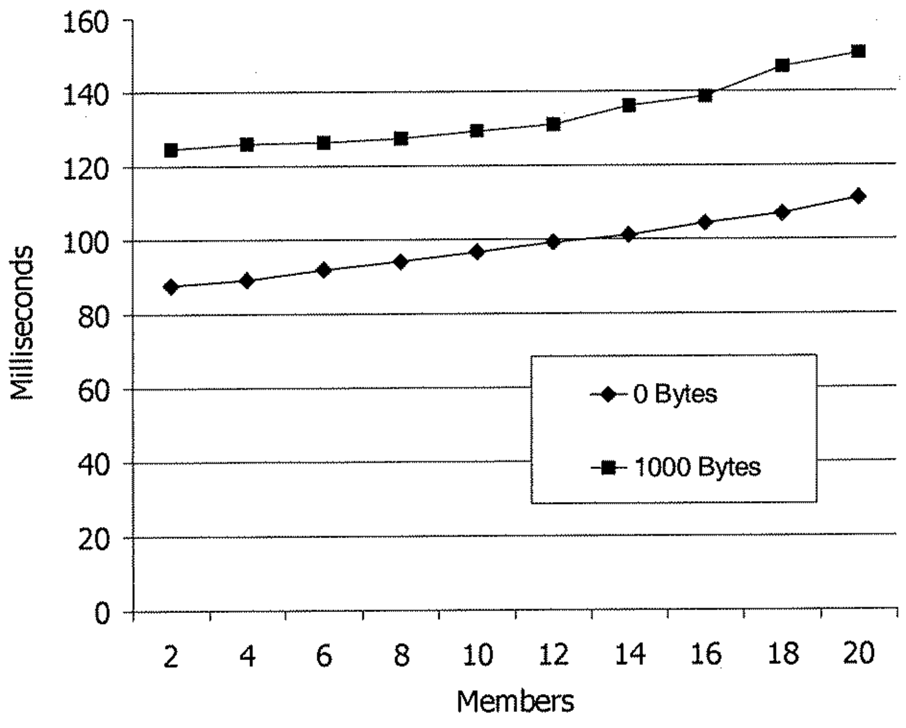


Figure 6.1 Multicast latency timing figures for zero and 1000 byte data sizes.

The latency of multicasts does not increase dramatically over the range of group sizes depicted, with all measurements being in the order of one tenth of a second. This is sufficiently low for interacting group participants not to experience noticeable delays. The maximum measurement, taken with a group size of 20 for a 1 Kbyte data size is only 150 milliseconds.

The increasing times as the group size increases are due mainly to the production of the Vector Signature, although the increased communication time of the message due to the longer signature does have a small effect.

The figures were obtained using 120 MHz Pentium PCs running the Windows NT operating system and communicating over a 100 Mbit/s Ethernet network. The whole system is implemented in Java and for encryption the system uses the Cryptix Java cryptographic package implementation of the IDEA symmetric key algorithm that has a 16 byte key length. The system was run using the SUN JDK v1.1.4 'just-in-time' Java virtual machine.

The system timed was a modified version of the actual system, adapted for timing purposes. For accuracy and reduction of fluctuations in operating system and network performance, the figures shown are an average value of 500 II-Mcasts. Each subsequent multicast was sent only after the previous one had been completely delivered.

The following section contains a more thorough examination of the Vector Signature's contribution to the timing figures.

6.2 The Performance of Vector Signatures

The time taken from a multicast being initiated to it being delivered at another group member process can be divided into two portions: the time taken to construct the Vector Signature and the time taken to construct and send the message. If the data size is constant then the proportion of the time devoted to constructing and sending the message remains constant also.

The construction of the signature is a significant proportion because it involves expensive encryption and decryption. In fact, in the case of Vector Signatures this involves many such encryptions: one for each of the other group members. Hence as the group size increases, so too does the proportion of the time spent constructing the Vector Signature.

This is depicted in Figure 6.2, where the increasing proportion of the time spent on constructing and verifying vector timestamps can be clearly seen. The proportion rises from 25 per cent of the total latency for a group size of two, to 37 per cent for a group size of 20. Obviously this proportion continues to rise as the group size increases beyond 20. These figures were obtained with a zero data size.

The alternative to using Vector Signatures is a conventional asymmetric key authentication mechanism. Under such a scheme latency does not increase substantially as the group grows becuase the work involved in the production of the signature remains constant at one asymmetric encryption of the message digest.

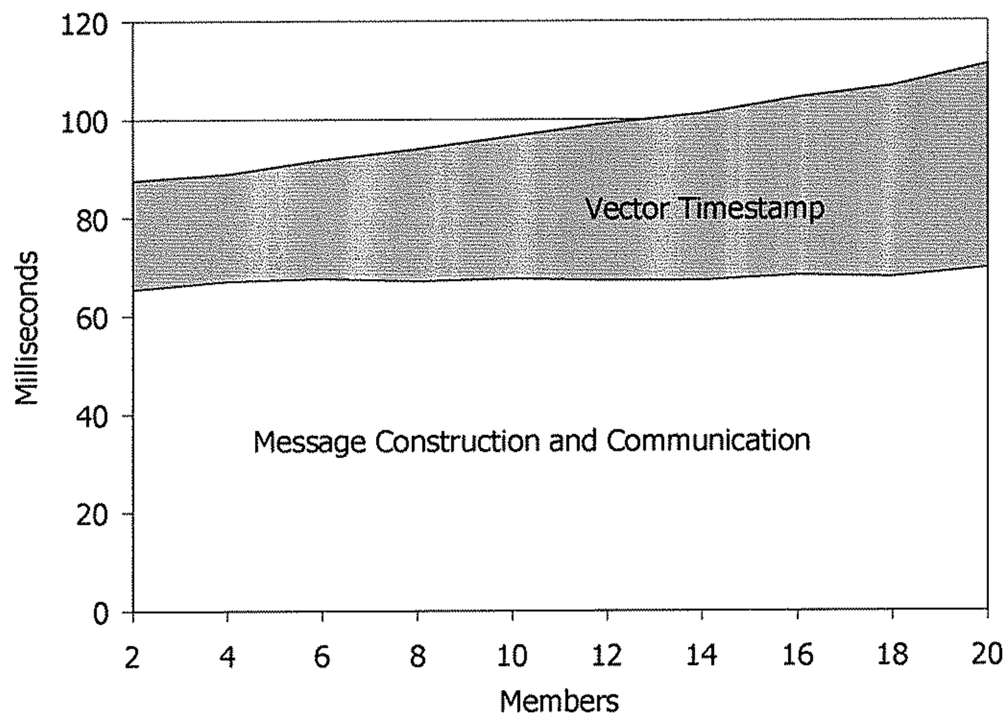


Figure 6.2 This chart shows the increasing proportion of the time that is spent constructing the Vector Signature as the group size increases. The figures presented were recorded with a zero byte data size.

These observations mean that inevitably there will be a point at which the Vector Signature scheme becomes inferior to the asymmetric alternative. This point can be approximately estimated by extrapolating the graph of timing measurements to very large group sizes. The time to produce a vector signature is assumed to be linear as the group size increaces. This is appropriate because the additional work as each new member joins is the same for each member. It consits of one extra signature calculation (one encryption) and a fixed addition to the message length. The resulting graph is depicted in Figure 6.3.

The crossover comes at approximately 1800 members: a figure that is far larger than can be productive for any group.

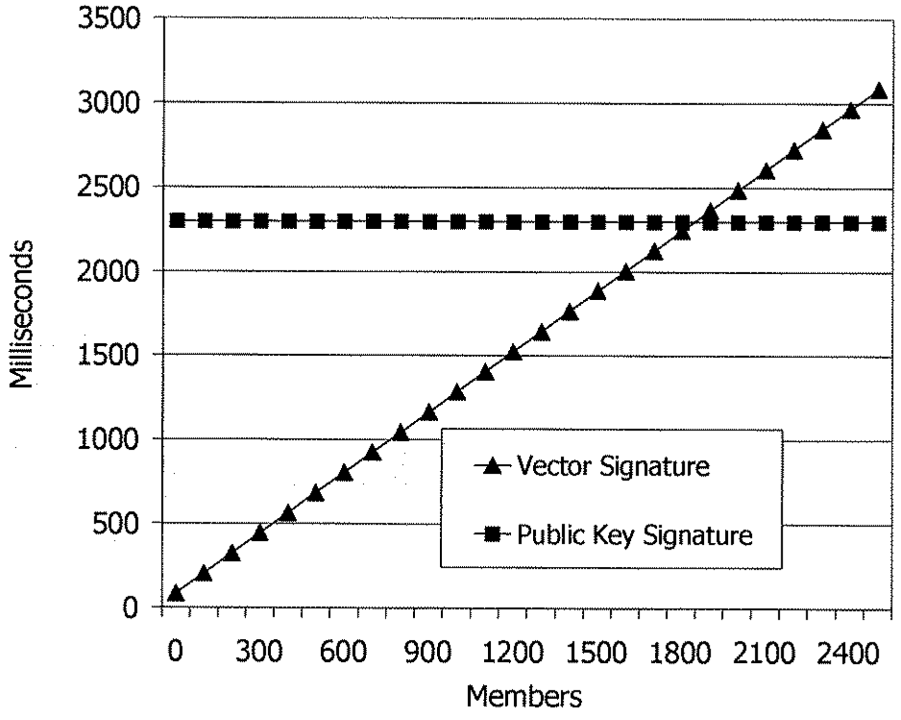


Figure 6.3 An extrapolation of timing figures comparing authentication using vector timestamps and a conventional asymmetric key scheme. Both figures were obtained using the Cryptix cryptographic package for Java. Symmetric key encryption used the IDEA algorithm and asymmetric encryption used RSA with a 512 bit modulus.

6.3 Other Evaluation Issues

The system was implemented in order that the speed of multicasts could be assessed and consequently fault-tolerance was not addressed. If the system were to be used in a practical setting then a scheme for recovering from a member failure and in particular a Distributor failure would need to be implemented. However the existence of fault-tolerant group communication systems such as ISIS show that this can be a reality.

It is important for the value of this work however to be able to dismiss the potential types of failure as security threats. It is clear that the failure of a normal member cannot disrupt the group in any way since these members do not have any direct contact with the other members. The failure of the distributor of course will inevitably mean the demise of the group, however since group communication is driven from the distributor, updates will merely cease to be distributed and the group will remain secure. The members will just behave as if there are no new updates. Updates initiated by themselves will of course be lost, but this is not a problem from a security point of view.

Denial of service attacks against the group cannot cause a security breach either. An attacker with malicious intentions could for example repeatedly attempt to join a group for which it has no rights. This would result in the distributor returning the encrypted state of the group (Join 2A) if the joiner falsely claimed that it was a principal legitimately entitled to join. The state would be returned to the malicious party, however it is of course encrypted in the legitimate principal's public key and can therefore only be decrypted by them. Any other messages either replayed or formulated by the malicious party will be ignored because of the authentication and anti-replay features of the protocol. The most serious result of such actions can only be the hanging or crashing of the distributor (discounted as a threat in the previous paragraph).

Scalability is another important issue. The centralised role of the Distributor in maintaining the group abstraction means that the system as it stands would not scale well as the group size increases. Again however this is an issue that can be addressed. As the group size increases then so too does the workload of the Distributor. Therefore if there was a mechanism for introducing replicated Distributors then the work could be shared. This would also provide a big step towards making the system fault-tolerant as the currently fatal failure of the Distributor could be managed if there are others to take over the task.

However introducing multiple Distributors would mean that not all multicasts to a group would pass through the same point and this in turn would mean that the current implementation of a total ordering would no longer work. However this problem is not necessarily insurmountable. It would just mean that the Distributors would have to communicate when assigning sequence numbers for the purposes of ordering and reliability.

The Distributors would form a group in themselves that collectively maintain replicated data, i.e. the sequence number. However this group is different from the whole group because the Distributors are all totally trusted and thus would be trusted by each other and so could communicate under the cheaper ‘trusted island’ model of trust employed by other group communication systems.

The major lesson to be learnt from the group communication work presented here is that the factors that are all important for secure distributed groupware are in contention. This system has shown that fast secure group communication is possible, but at the expense of scalability. Catering for scalability, whilst possible, would decrease performance. Scalability and performance can be realised together, as several of the examples in Chapter 3 showed, but at the inevitable expense of security, in the shape of an inappropriate trust model.

Therefore any of the apparent possible solutions is to some extent a compromise. The best system is perhaps the most flexible that allows the balance to be tipped towards whatever application and trust environment exists. This adds weight to the layered implementation of the system described here and elsewhere (notably Horus [RHB94] [RBFHK95]).

7

Secure Shared Objects

The background chapter made it clear that replicating data close to where it is used is the key to good interactive performance for groupware. This is widely accepted and has been demonstrated in many systems and applications: PerDiS [CDKR97], Caelum [ACDK97] and Javanaise [HL97] were studied in Chapter 3. However, replicating data is complex. Fortunately group communication systems can relieve a programmer of many of the problems associated with maintaining replicas. Often though communicating through sending messages is not the ideal abstraction.

An abstraction more suited to the programmer's view is the Shared Object. The programmer can be presented with a view of the distributed system that consists merely of conceptually shared programming level objects. These objects can be mapped into the address spaces of any applications that currently have users participating in a group activity. Communication between the participants happens transparently as Shared Objects' methods are invoked. Some of the examples studied in Chapter 3 offered such an abstraction, for example Javanaise [HL97].

If an application has some security requirements then the same layers of abstraction of course remain useful, however security guarantees must be built into each of them. Figure 7.1 below is an adaptation of Figure 3.3:

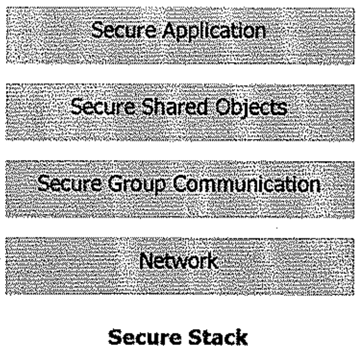


Figure 7.1 Layers of secure groupware. Access control fits in at the level of Secure Shared Objects. Replication is dealt with at the Secure Group Communication layer.

The security guarantees introduced at each level will only make sense to the programmer if they are made at an appropriate level of abstraction. Hence it will not make sense to offer guarantees of message authenticity at the level of shared objects.

Security at the shared object level is best stated in terms of objects and the ability to invoke their methods. Chapter 3 also gave some examples of secure shared object abstractions: such as Secure Network Objects for example [DABW95]. However these did not make use of replication as they were server-based. Take a group editor as an example application: if every time the user scrolled to a new paragraph, the application had to invoke a method of the object in the remote server, then editing would be slow and consequently less productive. Replicating the document's state at every application instance solves the problem.

This chapter introduces a Secure Shared Object abstraction that uses replication to provide the necessary performance for interactive groupware. The system builds its abstraction upon the Secure Group Communication System described in Chapters 4 through to 6. Security guarantees are given in terms of many principals' abilities to invoke methods upon the Shared Objects. In other words access control is applied at object method level.

Every Secure Shared Object can have an access control list for its methods. This is merely a list of principals that are entitled to invoke each method. The access control model is entirely compatible with Lampson's access control model [LABW92] and so does not exclude the possibility of delegation and role membership outlined in Chapter 3.

This chapter is intended to join the two main bodies of work of this thesis. The essential features of the Secure Shared Object system are described. For a more intricate description of an Objective C implementation of the system the reader is referred to the technical report describing the system [RD95]. The system described is in fact a secure extension of a non-secure Shared Object system developed by Achmatowicz [AK94].

7.1 The Programmer's View of Shared Objects

The Shared Object system gives the programmer the illusion that there is only one version of each Shared Object and that this single object can be mapped into the address space of all participants. When one participant does something that causes a method of a Shared Object to be invoked then any changes to the object state that result are accessible to all of the participants, in that the new state is reflected in the results of any invocations that they make.

In reality each Shared Object exists at every machine that needs to access it, i.e. every participant has a replica. Updates to the object state that occur as a result of a method invocation are communicated to the group. The updates are distributed using the group communication system that ensures that they are applied in the same order to every replica. Hence every participant's view of the Shared Object actually passes through the same sequence of states. Each replica is in effect a state-machine as described by Schneider [SCH90] and used in that case to provide fault-tolerance.

The programmer can treat the Shared Object as if only one copy of it exists. However because of the delays in network communication, the reality is that replicas may not actually even exist in the same state at exactly the same instant in time. However this does not affect the programmer's view and does not have any bearing upon the way he writes an application.

7.2 Using Proxies to Implement Shared Objects

Local replication of data and hence group communication is the key to adequate performance for groupware. However communication abstractions are not the ideal interface for the programmer. As stated above, the programmer's job can be made easier if communication is made transparent.

The Shared Object layer presented here builds upon the secure group communication layer described in earlier chapters. Every Shared Object in effect has its own underlying process group. The group certificates (Section 5.2.1) are used to name and locate objects, state transfer is used by a new member to obtain the state of a Shared Object and group multicasts (II-Mcast: Section 5.1.5) are used to keep members informed with updates to the state. All this communication however is hidden from the application programmer behind the illusion of a single Shared Object.

Proxies are the key to hiding the communication in the Shared Object abstraction. In the programmer's view, the proxy is the Shared Object. The programmer treats the proxy as if it is the Shared Object. He writes code to invoke its methods, and transparently the other participants observe any changes that the invocations cause when they invoke methods upon their proxies.

In fact each proxy maintains a replica of the Shared Object. Upon receiving an attempt to invoke a method from the local application, the proxy does one of two things depending on whether the method invoked alters the state of the Shared Object or not:

1. A Method call that does not alter the Shared Object's state (read method) can be handled entirely locally by invoking the method upon the replica. The results of the call upon the replica are returned as the results of the call upon the proxy. The caller does not need to be aware that the object that actually services the call is not actually the Shared Object.

2. A Method call that does change state is converted into a message together with any argument values and distributed in a multicast message to all the processes that are maintaining a replica of this object, including itself of course. The proxy object in that process receives this message. Upon receiving such a message each proxy unwraps it and reconstructs the original method call complete with arguments and applies it to its replica. In this way all of the replicas can be made to pass through the same sequence of states.

This implementation of course relies upon the communication guarantees of the group communication system that it is built upon. The update messages need to be reliable and ordered for the replicas to maintain some useful consistency. For the replicas to pass through exactly the same sequence of states, a total ordering of update messages is needed. These requirements are of course met by the secure group communication system described in Chapters 4 through to 6.

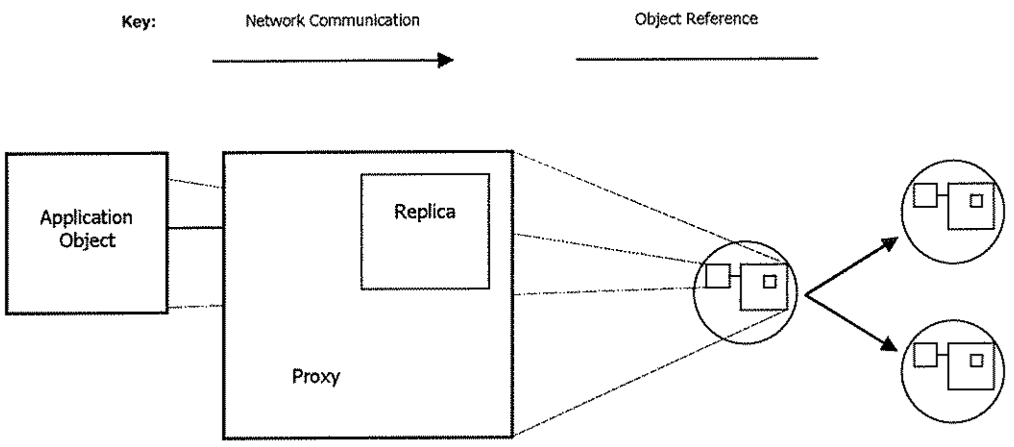


Figure 7.2 The group of processes shown on the right each contain application objects. Each application process maintains a reference to a proxy object for each Shared Object that it needs to access. The proxy in turn maintains a replica.

The internal structure of the proxy is represented by Figure 7.2 and this is how it is implemented by Achmatowicz [AK94]. The application creates the proxy when it wishes to access the Shared Object state.

The programmer of a class that he wishes to be shared must implement certain methods in that class. These methods enable instances of the object to be marshaled and unmarshaled. Additionally the programmer must supply a method that when called, distinguishes between the methods of that class that can potentially update state and those that only read state. This method is called by the proxy when making the decision about whether to distribute the method invocation or not.

7.3 Naming and Locating Shared Objects

Every different Shared Object in a system has a different underlying process group. Any application that needs to access the methods of an object must first instantiate a proxy. This proxy then joins the appropriate group. In this way updates to state are confined to just the set of applications that are actually using the object.

Since every object has a unique group, then naming the object is the same as naming the group and hence is achieved with the group certificate that was introduced in the Chapter 5 (Section 5.2.1). Recall that the certificate also contains the IP multicast address of a group and hence serves as the means of locating a group and consequently locating a Shared Object.

Once the appropriate group certificate is obtained then the group can be joined and hence the state associated with the group obtained. This state obviously forms the replica maintained by the proxy. Subsequently any updates caused by other group members invoking methods that change state are multicast and applied to replicas by the group members.

7.4 Protecting Objects with Guards

Merely building the Secure Shared Object system over the Secure Group Communication system ensures that communication across the network is safe from the usual threats (eavesdropping, replaying etc). This section explains how the objects are further protected by extending the Shared Object system to include protection of methods. This too is straightforward: simply a Guard (that is added to the proxy) intercepts all attempts to invoke the Shared Object's methods. The Guard has a reference to an access control list that contains principals' rights. The Guard can consult the ACL and either allow or disallow the invocation to proceed. Figure 7.3 depicts the internal structure of the proxy for Secure Shared Objects.

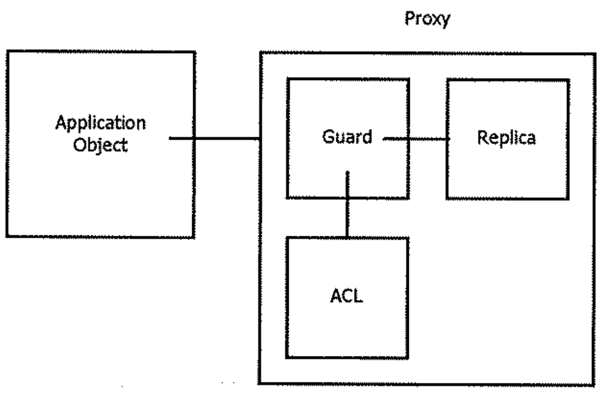


Figure 7.3 The Guard intercepts attempts to invoke the methods of the Shared Object. It consults an ACL before allowing the invocation to succeed or not.

7.4.1 Access Control Lists

The access control list (ACL) associated with the methods of a Secure Shared Object is conceptually a list of principals that are entitled to invoke each method. Each entry in the list can be a simple named principal (as defined by Lampson [LABW92]). Chapters 8 and 9 will expand on this however and show how far more elaborate security policies can be expressed and hence enforced.

In the implementation, the ACL is in fact an object with its own methods. Most obviously there is a method for checking whether a principal appears on the list for a method. Also there are methods for updating the list. Since the ACL is itself replicated inside every proxy, then any updates that occur to it during the life of a shared object must also be communicated to the group. These are included in multicasts that are totally ordered with updates to the shared object.

An ACL object is not a shared object in its own right however because if it were it would have its own ACL and an infinite recursion problem would result. Issuing an update to an ACL must of course be controlled though. Only trusted authorities should be able to change rights. Hence in effect the ACL has itself got a Guard protecting it. The recursion can be resolved by having an immutable ACL protecting the shared object ACL that allows updates to come only from an authoritative role. The role could be called 'SecurityManager' for example. Membership to this role can then be altered accordingly without the need to distribute any updates.

7.4.2 Getting First Access to a Secure Shared Object

Joining a group results in the new member becoming party to state associated with the group, in this case the state of the Shared Object. Hence access to the group should be restricted. Recall from Chapter 5 (Section 5.2.2) that decisions regarding admitting new members to the group are deferred by the group communication layer to the layer above, which in this case is the Secure Shared Objects layer (this only happens in the process that is acting as the Distributor). So how is this decision made?

In this system, every group member has a complete copy of the state comprising the Shared Object in the form of the replica maintained by the proxy, i.e. every user has a copy of the state in the address space of a process belonging to them. It is not therefore feasible to expect that the user cannot get access to this state. Another way of looking at this is that every member of the group effectively has the right to invoke all of the methods that merely make inquiries of the object's state.

For this reason it makes no sense to allow a prospective member to join and gain access to the Secured Shared Object if the principal does not have the right to invoke all of the object's read methods. Hence this is the criterion used (by the process acting as the Distributor) to reach the decision allowing membership. This seems to be quite an extreme decision, but is an inevitable consequence of the chosen model of trust and the performance requirement (which resulted in the decision to replicate data on the machines of participants).

Resorting to a more relaxed model (where we trust members not to trawl their address space for unauthorised data) would remove the problem. Alternatively selectively controlling access to read methods could be achieved by compromising the performance of the system. Such principals could be given access to state through a trusted custodian process that runs on a different machine. The custodian would hold the entire state of the protected object and pass only the results of the permitted read methods back to the invoker. The technical report [RD95] elaborates on this point.

7.4.3 Invoking a Method of a Secure Shared Object

Invoking a method of a Secure Shared Object results in the following five actions by the proxy:

1. The proxy first performs a local check against the ACL to see whether the principal does actually have the right to perform the operation. The security of the Shared Object does not rely on this check being performed correctly because the practical model of trust for groupware does not make assumptions about the behaviour of individuals. The check merely allows useful feedback to the user if something illegal is attempted and the action can be halted at this stage without burdening other processes.

2. If the method invocation is legal, then any arguments provided with the call are marshaled into a message together with a description of the method signature. This is then multicast (II-Mcast) to the group and consequently is received by all the other proxies maintaining replicas of this object. Recall that the group communication system will ensure secure ordering, authentication and encryption of the message (if required), all without making any assumptions about the behaviour of the source.

3. Once received by the proxies, the method call and its arguments is reconstructed by each of them.

4. The proxies then perform their own access control check by consulting their copy of the ACL. In this way everyone merely relies upon themselves for the security of their own version of the Shared Object.

5. If a proxy deems that the invocation is legal then the method call is invoked upon their version of the replica. In this way, every uncorrupted proxy will see the Shared Object progress through the same series of states.

7.5 Summary

The practical groupware model of trust derived for the Secure Group Communication System for groupware is one in which no ordinary group participants are trusted to do anything in maintaining the group abstraction. In particular ordinary group members are not even trusted to run the same system as correctly functioning members. In other words the group abstraction will remain intact despite the corruption of some of its members.

This means that the only responsibility that a principal (and hence the code he runs) has in maintaining the abstraction is to himself. Participants look after themselves and do not rely on others, except of course the trusted Distributor.

This underlying model of trust is carried upwards into the Shared Object level. Each participant runs proxy code that maintains his own version of the replica of the Shared Object. As long as he runs the correct code and hence adheres to the correct protocol then the version of the Shared Object that is maintained will be consistent with the versions maintained by all the other correctly functioning participants, i.e. they will pass through the same sequence of states.

The system as it stands can enforce only the simple security policies that conventional static access control lists can describe. As Chapter 2 pointed out, real-life policies tend to be more complex and dynamic, particularly those that apply to interactive group activities. This thesis now goes on to examine this shortcoming in greater detail and also to offer solutions. These new dynamic access control ideas can be integrated with the Secure Shared Object system described in this chapter.

8

Access Control for Groupware - Design

Chapters 4 and 5 have discussed the design and implementation of a secure group communication system designed specifically for the needs of groupware applications. At the group level of abstraction, access is controlled to the group at a coarse granularity. Principals are either permitted to join or they are not (this being decided by the level above). The system allows group membership to be controlled. However, any reasonably sophisticated security policy cannot be expressed in these terms.

This chapter covers the subject of access control for groupware applications. Access control does not conveniently fit in with communication abstractions. Security policies are described in terms of real world objects such as medical records and financial data and not often in terms of messages. Hence access control is best built in at a higher layer. Chapter 7 showed how a shared object layer is far more convenient for specifying access control since a well designed object-oriented application will have models of real world objects as its highest level programming objects. The methods of these objects represent real life operations such as reading or writing to a medical record object.

Access control is not always associated with groupware, however in one sense access control is only applicable to groupware. This is because access control is only needed when the actions of multiple parties can affect each other. Multiple parties affecting each other can be seen as a definition of groupware. This is similar to the broad definition of groupware given by Ellis et al [EGR91] and others [RB92b]. This notion is particularly obvious in role-based access control: as Hilchenbach [HIL97] points out, a role is after all essentially a group of principals related by some rights that they share.

However conventional access control techniques only cater for a range of groupware that is relatively unsynchronised, where participants' concurrent activities are shielded from each other. In these areas the access control model of Lampson [LAM74] [LABW92] is often employed. This model was outlined in

Chapter 3 and an implementation of the basic model (excluding delegation and role taking) as it is used to protect shared objects was outlined in Chapter 7. The model is built upon of a view of the system consisting of objects protected by Guards. The Guards intercept attempts to invoke the methods of the protected objects by principals. The Guards make the decision of whether to accept or reject the attempts by consulting the access control list (ACL) for that method. The ACL contains only the names of the principals that have the right to invoke the method. The model also caters for compound principals which encompasses the concepts of delegation of rights, conjunctions of principals and for granting rights to groups of principals: principals who all shared a common organisational role.

Chapter 3 showed how static a conventional access control list is. Typically a change to a principal's rights would involve the intervention of some authority in order to change the access control list. However, often rights are dynamic, even when the security policy does not change. A driver's right to pull away from a road junction changes when the traffic lights change, but the policy remains the same. It would be more convenient if changes to an access control list were necessitated only when policy changes. In other words what is needed from the access control system is for the expression and enforcement of security policies and not merely rights, since policy is relatively static whereas rights are dynamic.

Other research has attempted to tackle this problem by observing that rights change as some task changes. The Background chapter reviewed some examples. Edwards [EDW96], Holbein [HT95], Hayton [HAY96] and Moffett [MS91] all explored the possibility of having rights depend upon the state of objects in the system. Section 8.2 in this chapter builds on this with the additional observations that rights also depend upon other factors such as the exact nature of what is being attempted and environmental factors such as the time of day.

The main emphasis of this chapter is on providing access control for more synchronous and interactive groupware. Applications of this type offer participants new ways of interacting and so not surprisingly there are new requirements for controlling this interaction. Groupware can facilitate interaction and negotiation between participants that often mimic real life interactions. However very often in secure real life situations, access rights are one of the topics that are being negotiated. Further, rights can be modified as a result of the discussion and negotiation. Delegation is one example, however this chapter will show that in groups this exchange of rights between participants without the involvement of some authority can go further.

Section 8.3 introduces the concept of **backing** and explores its consequences. The concept captures the situation where one or more participants back another participant to do something. It is different from delegation because before the action of backing no one in the system necessarily had such a right. In effect the right is distributed amongst the group of backers.

State-dependent access control and backing often go hand-in-hand and that is the reason for them both being tackled together in this chapter. Often a security policy makes a principal's rights depend upon state in an attempt to very tightly control the circumstances under which the protected action is successfully completed. Such types of policy include 'need-to-know' policies where an attempt is made to define the exact situation under which a principal really needs to know (or do) something. However tightly controlling access in this way can often be a bad thing if there is no way around the controls. This is the case because the designers of security policy cannot preempt every possible scenario. Hence policies often provide for an alternative circumstance under which the principal can execute the operation and this very often involves getting the backing of one or more other individuals. This interrelation between state-dependent access control and backing is also explored in Section 8.3.

In summary, the aim of this access control work is to allow access control lists to contain descriptions of security policy and not just a list of rights. This means that the policy descriptions which should be confined solely to the access control list should allow for the dynamic nature of rights found in interactive groupware.

This chapter starts with a closer examination of rights within group tasks by examining real security policies drawn from various literature sources.

8.1 Group Security Policies

This section introduces some examples of security policy that could be applied to group activities. To the author's knowledge, none of the actions described in the policies have actually been accurately specified and enforced in a computer system. This is largely because of the nature of security surrounding the data under protection and the lack of existing mechanisms for enforcing the policies. None of the policies listed below could be conveniently described in a conventional access control list. Conventional access control lists can only express policy in terms of objects, actions (requests to execute object methods) and principals. However the policies described here depend upon a wider variety of variables.

A conventional access control list cannot refer to the state of objects. Many examples of policies that refer to state can be taken from the medical field. The following policy taken from a list of dynamic security policies in Appendix B gives an example of an attempt to restrict the circumstances under which a clinician can access personal medical data by restricting access to an eligible group:

P1. *"Some extra restrictions may be needed in defining groups; for example, the group might be any clinical staff on duty in the same ward as the patient."*

This policy depends upon the state of a duty rota. P1 is a dynamic policy because rights can change even though the policy stays the same. One possible way that this policy could be expressed and enforced in an access control list would be to add the name of the doctor to medical records of patients when a doctor changes his duty. Alternatively there could be a unique role created and granted access to every record of a ward and the clinician assigned to that role when they change duty. Both these solutions are obviously impractical, particularly if the number of factors that rights could depend on is large. This point is made by Hilchenbach [HIL97] when considering how best to enforce a policy that gave probationary bank tellers fewer rights than more experienced staff. Whether or not the teller was on duty or not was just one of five factors that could effect rights and each of these five variables had an average of five values. Consequently this lead to 5^5 (3125) possible roles for bank tellers alone!

Some security policies are dependent upon more than just the state of objects. They depend upon the state of the environment. Typically such policies refer to the current date or time of day. P2 is taken from Anderson's suggestions for security of medical records [AND96]. The report contains nine principles that were prepared on request for the British Medical Association. One of them expresses rights in terms of the current date:

P2. "No-one shall have the ability to delete clinical information until the appropriate time period has expired."

Rights here depend upon the current date. Enforcement of this policy using a conventional access control list would require a change to the list every time that the critical time period expires for every record. This is clearly impossible to achieve in any practical situation.

Some policies rely upon the exact nature of the operation being attempted rather than the state of objects or the environment. Policy P3 is taken from the financial field. Kusner and Anterpol's lengthy explanations of banking procedures [KA81] includes the following security policy:

P7. "Bank tellers should not be able to process transactions that involve themselves."

This policy relies upon information that can only be gleaned from the parameters of the attempted operation, in this case the identity of an account holder. A medical example might state that clinicians could prescribe only certain quantities of drugs.

Often security policies dictate that more than one person should be involved in an activity. Groups are used to increase security by distributing trust. Such policies are designed to reduce the possibility of corrupt activity by making it impossible for one corrupt individual alone to do damage. A bank safe might have two keys for example. Both key-holders must agree before the safe can be opened and as a result the safe cannot be opened if just one of the key-holders had malicious intentions. It would take the corruption and collusion of both to commit damage.

Policies of this type will be referred to here as 'backing' policies, because one person is required to seek the backing of one or more individuals prior to an attempt to do something. Financial applications provide many further examples of these policies. Security literature often refers to 'segregation of duties'. Gray and Manson [GM89] describe this technique in their descriptions of financial auditing. The technique is applied to tasks and is concerned with ensuring that no one person can complete the task without the involvement of at least one other individual at some stage. Often work is checked by some authority and a hand written signature is applied before the employee can continue.

Kusner's descriptions of banking procedures also contain examples of this kind of policy. The following example requires that someone senior back a bank teller:

P14. *"A teller must seek permission to adjust a special purpose bank account for reconciling differences between the actual amount of money taken and the recorded amount (should any discrepancy occur)."*

Note that although simple delegation could well give the teller the necessary right to adjust the account, it would be unwise to grant the right for any period of time. In fact it is probably important that the teller only be able to adjust the account once without seeking further backing.

Usually there will be more than one potential backer and these form a group. In all the examples shown here the group is formed by all the principals entitled to take on some organisational role. Managers for example, or Senior Clinicians. Often the backing of more than one is required. Some policies don't specify an exact number of backers that a principal needs. Instead the required level is expressed in terms of a proportion of a group of role members, for example the majority of the board of directors.

Draper [DRA96] discusses the treatment of mental health patients as being a collaborative process. The backing of some proportion of a group is required to support decisions. Decisions are taken collectively as a safeguard in cases where the patient's consent cannot be obtained. An example policy could be:

P9. *"A clinician must obtain the backing of a majority of a patient's carers in order to prescribe new treatment."*

In summary, security policies often refer to more than just a principal's identity and the action attempted. Rights can be:

1. Dependent upon the state of objects in the system.

2. Dependent upon environmental factors such as time.

3. Dependent upon the parameters of the action being attempted.

Additionally, security policies can often require that a principal receive the backing of another principal or principals. Backing can come from:

4. A specific number of individuals.

5. A proportion of a group.

This chapter continues by analysing the resulting issues that state-dependent access control (Section 8.2) and backing (Section 8.3) raise.

8.2 State-Dependent Access Control

The previous section found that rights in a secure activity could be dependent upon three types of state: the state of the task; the state of the environment (usually time) and the state of any parameters of the action being attempted. The following three sections explore the consequences and limitations of letting rights depend upon these types of state.

8.2.1 Task State-Dependent Policies

The current state of a group task as represented inside the system is of course held within the shared programming level objects that applications create. Hence some state-dependent policies must be enforced by allowing the access control system to test the value of object state.

This is best and most obviously achieved by allowing object methods to be called as part of a Guard's decision process. An alternative would be to allow the Guard access to the instance variables of objects. However this is not in-keeping with object oriented notions of data abstraction.

In the access control model, security policy is stored in access control lists. As a consequence it is here that object method calls must be specified. Chapter 9 will tackle the implementation issues and the exact form that the extended access control lists take. It will be shown that boolean expressions that include object method calls can be evaluated as part of rights evaluation. However there are two obvious limitations to this approach: what happens if the required state (object) is not in the system and what if there aren't suitable methods belonging to the objects that are there?

Policies that Rely upon State that does not Exist Inside the System

Policy could depend upon other state; state not held within a system, but obviously such access could not be enforced, as the Guards cannot access everything they need to make such a decision. In fact, very often in group activities this is where a requirement to obtain backing is introduced. If it isn't possible to decide if the exact circumstances exist under which a principal can do something, then the backing of others can be sought in order to verify that the attempt is legitimate. The backer can then possibly check state that is external to the system (paper records for example) as part of his decision processes. In other words a solitary activity can be made a group activity for security reasons. The interrelationship between backing and state-dependent access control is discussed further in Section 8.3.

Policies that Rely upon State that is not Accessible Through Existing Methods

If security policy refers to state that is not directly accessible through a protected object's existing methods, then adding new methods for access control purposes would probably not be so inconvenient. The main purpose of this work is to enable access control lists to contain descriptions that are closer to security policies and not merely a list of rights. This is in order to reduce the level of interference necessary from authorities when rights change. Adding object methods especially for the purpose of access control would very likely be a once-only job that required no further intervention at the programming level. For example, a method might be added to a medical record object that returned the value of a patient's age, so that this could be used to enforce a security policy that depended upon a patient's age. If that policy changed so that a parent's consent was needed if the patient was under 16 rather than 18, this would not necessitate any further changes to programming objects, just changes to the access control list.

The Possibility of Covert Channels

Another possible worry about having rights depend upon object state surrounds the possibility that covert channels may open up. This worry occurs because even an unsuccessful attempt to invoke an operation might enable the principal to infer something about the state of protected objects inside the system. Examining an example policy can alleviate these worries to some extent.

Suppose that like policy P1, in order for a clinician to update a medical record, she must be on duty in the ward that the patient is in. This policy is designed to protect against a clinician's malicious actions, for example making unnecessary prescriptions. The clinician's right depends upon the state of the duty rota. A clinician could infer something about the state of the duty rota if she attempted to update a medical record even if she wasn't on duty in the ward. However, the clinician could only infer that she is not on duty. This is not a problem because all clinicians have the right to look at the state of the duty rota anyway. In fact, even before any computerisation of the medical record system was in place, it must have been possible for clinicians to observe the state of the duty rota in order for them to evaluate for themselves if they are allowed to update a record.

Another example taken from the list of security policies in Appendix B is P7. This policy has already been introduced in Section 8.1:

P7. [KA81] *"Bank tellers should not be able to process transactions that involve themselves."*

In this example a bank teller attempting to conduct a transaction could only infer that they themselves were connected with the transaction in the event of a failed attempt at an invocation. This is something that is probably perfectly obvious to the teller anyway.

The point is that most sensible security policies would only make a principal's rights depend upon something that the principal has a right to know, this has to be so in order that a non-malicious principal can evaluate for themselves whether or not to even attempt the action.

The worry of accidental covert channels can be further reduced by ensuring that the system, i.e. the Guard ensures that any methods that are invoked as part of rights evaluation are invoked with the original principal's authority and no more. State dependent access control inevitably results in an unsuccessful invocation revealing something about the state of the system. However subjecting secondary invocations to the same checks as any other invocation means that at least this is not information that the principal is not entitled to obtain by other means.

8.2.2 Environment-Dependent Policies

Rights often change with the time of day. As an example taken from outside the computerised world, a manager might only have the ability to open a safe inside office hours because it has a time lock. In the computing realm, a doctor's right to access medical information might depend upon whether or not he was on duty at that moment. An example of such a policy is P2:

P2. *"No-one shall have the ability to delete clinical information until the appropriate time period has expired."*

The calculation of the predicate referring to whether the current time period has expired relies upon the current time. Therefore, the access control system enforcing such a policy must be able to take the time of day or date into account. This is not of course possible for conventional access control lists.

The solution is a straightforward extension of what has already been covered in this chapter, i.e. environmental inquiries can be made by the Guard through normal method calls. A Guard can gain information about the current time from the operating system and appropriate methods can be supplied as part of the access control system. Chapter 9 deals with the implementation and how such calls can be included in an access control list.

However this has one important security repercussion. It is important that the means by which the operating system gets the current time is secure. If the Guard exists on a secure machine whose clock is set by a system manager then this may be acceptably secure, if not highly accurate. However if the clock is set using a time service then the security of this service should be ensured. Secure time servers have been developed because of the general havoc that a corrupt server could wreak upon a system [MAR84] and also because authentication protocols often depend upon an accurate system clock for their security. Kerberos [SNS88] [KNT91] is a prominent example.

In all of the example policies investigated as part of this research, the current date and time have been the only environmental factors upon which security policy depended. Nevertheless, other information could be gained from the computing environment such as available disk-space, and it is conceivable that rights could depend upon these as well.

8.2.3 Parameter-Dependent Policies

Conventional access control recognises that rights vary according to what is being attempted, i.e. what method a principal is trying invoke. Often as has been shown already in this chapter the security policies demand a finer granularity of protection than can be afforded merely by guarding accesses at the object method level. Many policies differentiate between different invocations of the same method according to the parameters of the attempted invocation. For example, it might be acceptable for an architect (working with others on a shared design) to move the location of a wall by up to 30 centimeters, whereas greater movements would require greater authorisation[1]. This example shows that the principal's ability to do something depends upon the magnitude of one of the parameters of the operation.

Take the policy P6 as another example:

P6. *"Tellers can't cash personal cheques"*

A teller's right to invoke the method associated with cashing cheques does not depend only on who he is and the fact that he is attempting this particular activity, but also upon the details of the actual cheque involved, which will presumably be represented as the parameter to the method. Accordingly an access control system that deals with protection at this level would need to be able to refer to the values of parameters dynamically as methods are invoked.

It would of course be possible to change the implementation of the protected methods so that the test is performed by the object itself. However this would mean that security policies are contained within the object. One aim of this work is to have the description of policy confined to one place: the access control list. Chapter 9 will show how such policies can be represented in access control lists and how they can be enforced by the security system.

[1] This policy actually came from one of the PerDiS partners IEZ (CAD software retailers) and was reported to the author verbally.

8.3 Backing

Sometimes it is the case that some actions are too security sensitive to be under the control of one person alone. Splitting the responsibility between two or more principals, so that the consent of all or some proportion of them is required for the action to be taken, is a common way of protecting against corruption. Security policies that specify a need to obtain backing ensure that the collusion and corruption of more than one principal is required before malicious damage can occur.

At the start of this chapter, the example of a manager requiring the backing of another in order to get into a safe was given. Also the principle of 'segregation of duty' that is taken from the financial domain has been mentioned. This principle requires that no one person sees through a security sensitive procedure from start to finish without the intervention of another.

There are many other examples, such as P9 that requires that a clinician does not act alone when making decisions that concern mental health patients. The consent of other carers is important for such patients who are not always able to consent themselves:

P9. *"A clinician must obtain the backing of a majority of a patient's carers in order to prescribe new treatment."*

In another example taken from commercial law that restricts certain types of decision made by partners of a company:

P11. *"Unless the agreement provides to the contrary, all partnership decisions will be made on the basis of a simple majority."*

Backers must assess the merits of each individual request for backing. As all the examples show, any backing given is granted for a one-off action. The implementation must ensure that the backing from some source is used only once.

One way of understanding the backing security concept is by considering its parallels with fault-tolerance. In fact the concept of backing and distributing trust amongst a group is to security what replication is to fault-tolerance. The idea is that trust is doubled (or trebled etc.) in order to tolerate a breakdown in part of that trust, i.e. in order to tolerate some degree of corruption.

Backing is an interactive security mechanism. It requires the negotiation of rights between parties and hence it sits closely with collaborative applications and groupware. The collection of backing is often, but not always a quick and largely synchronous process. The required level of backing might however come in slowly over a period of hours or days. This poses a question about whether or not the cumulative level of backing ever actually existed at one moment in time and whether it actually matters if it did. This issue along with others is discussed in Section 8.3.3. It is certain that in some instances backing must be collected quickly, for example as a vote during a group conference. For this reason any implementation must be sufficiently fast so as not to disrupt the flow of other collaboration.

8.3.1 The Interrelationship Between State-Dependent Policies and Backing

Backing naturally follows on from state-dependent policies because often there is an interrelationship between the two. Such situations are particularly well highlighted by medical applications. State-dependent access controls are an attempt to tightly constrain the situations under which a principal can do something. However, often it is impossible or unreasonable to cater for every situation under which the action is acceptable. In these circumstances the backing of others must be sought. Hence security policies often use a combination of the two techniques. The following example combines a parameter dependent policy (pregnancy younger than 24 weeks) with a need to obtain backing (from two other registered medical practitioners):

P18. *"A person shall not be guilty of an offence under the law of abortion when termination is performed by a registered medical practitioner and two registered medical practitioners have formed the opinion in good faith that the continuance of the pregnancy would involve risk, greater than if the pregnancy was terminated ... subject to the pregnancy not exceeding its 24th week."*

Additionally, this example also shows that gaining the backing of others could well be a legal requirement. Hence if remote collaboration is being supported by a groupware system then backing must be non-repudiative. The implementation discussed in Chapter 9 will obviously therefore involve strong encryption and digital signatures.

Backing is in fact a solution to many access control problems that can't be implemented because immediate decisions regarding rights are deferred to agents outside the system. It means that policies that cannot be expressed conveniently do not necessarily go unenforced.

8.3.2 Backing and Role Membership

Most often the source of required backing in a security policy is not an individual principal but some description of a group. Invariably the group is the group of principals that are entitled to take on some organisation role, the familiar concept from role-based access control, e.g. Managers, Clinicians, the Board of Directors, etc.

Two distinct forms of backing were introduced in Section 8.1: backing that comes from a specified number of backers and backing that comes from a proportion of the group. When proportional backing is required, the proportion is usually a simple majority. In this particular form backing can be likened to elections, although the whole concept of backing is wider of course.

This second type of backing where the consent comes from some proportion of a group can cause a problem. This arises because of the period of time that elapses whilst a principal attempts to procure backing. The problem is that whilst the collection process is underway, the role membership might actually change.

For example, recall policy P9:

P9. *"A clinician must obtain the backing of a majority of a patient's carers in order to prescribe new treatment."*

If the procurement of backing took several days, it is possible that the patient gains a new carer during this period.

The group of principals entitled to take on some role is transitory and so what constitutes a majority (or some other requisite proportion) from the point of view of the principal collecting it, may not be deemed sufficient by the Guard.

The possibility of the Guard rejecting what the principal believes to be a valid attempt to invoke a method isn't really a problem. It is the state of the role membership at the point that the Guard evaluates the attempt that should be taken into account. Simply, the principal must accept the Guard's decision. In any case it is an unlikely event, particularly where the backing is being collected over a short period of time.

8.3.3 Semantics of Backing

It may seem obvious what the semantics of backing are, but in fact what exactly the backer consents to is not so straightforward. Because the backing of an individual is going to be used to gain access to something secure, it is very important that the backer be absolutely aware what they are backing. This section attempts to make that clear. Hence in this section several requirements of any implementation of backing will be presented.

Backing Must be Provable

Obviously it is important that the backing that one principal gives to another can be proved to the Guard. Public keys and role membership etc. are all proved using asymmetric cryptographic techniques and hence it comes as no surprise that a backer's intention to consent needs to be stated in a signed certificate. This is the first requirement of the implementation.

BR1: Backing should be provable: i.e. contained within a digitally signed statement of consent.

The details of the collection of backing will be given in the next chapter, however it is easy to anticipate that the implementation of backing consists largely of sending out a request for backing to all the potential backers and then waiting for them to reply with the signed certificates. Once the requestor deems that sufficient certificates have been collected then the method invocation can be attempted.

The Guard checks the validity of the collection of certificates and allows the invocation to proceed if the collection constitutes the necessary proof of backing.

Backers that Change their Minds

In general there should not be any limit on the time that it takes to collect backing. Some more synchronous applications might allow the requestor to quickly collect the necessary consent, however there seems no reason to restrict the concept of backing to highly interactive groupware. This results in a problem due to the elapsed time between the first backer consenting and the final presentation to the Guard of the backing certificates. Specifically, during this period, backers could change their opinions resulting in the potential absence of the required level of backing at the time of successful method invocation.

Revocation of capabilities and certificates are well-known problems. Certificates usually contain a time-out to tackle this issue. After a certain period the certificate must be reissued. A similar approach could be used to deal with the possibility of a backer's change of opinion. However at first it seems unsatisfactory to live with the possibility that at the moment that the requestor succeeds in invoking the protected object method, there may not actually be the necessary consent amongst the backers.

Perhaps a better solution can be obtained by considering more carefully exactly what the backers take into account when they agree to consent. A backer might have a particular view of the world when he decides to back. For example, a Manager may agree to back a Trader's attempt to purchase 10,000 shares because the price was below £2. However the price may rise before the Trader collects the rest of his required backing. Perhaps some criteria could be captured within the certificate and checked by the Guard to reaffirm that the predicates still hold true at the point of method invocation.

Whilst it certainly seems possible to include a summary of object or system state within a certificate, unfortunately it seems impossible to include all possible criteria. A backer might have consented for a potentially infinite number of reasons, not all of which have any representation inside the system. Backing could have been given because of a 'gut feeling' or simply because it was a nice day!

For this reason forcing the backer to include an expiry date and time in the certificate seems the best all-round solution. This must be made clear to the backer when consent is being sought and the backer therefore accepts that if he has a change of mind after granting his backing then there is nothing that can be done about it. This sounds harsh, but is exactly what happens for delegation in other systems. In the TAOS operating system [WABL94] for example it is impossible to immediately revoke a delegation.

Therefore the second backing requirement is as follows:

BR2: Backing once granted, must be limited in duration to a specified period of time.

The lifetime of the backing certificate would depend upon the exact nature of what backing was being sought. For example backing for a bank teller to adjust the discrepancy bank account (P14) would likely only last for a matter of minutes or possibly hours, certainly shorter than a day. However the backing of a doctor for sanctioning new treatment (P9) is likely to last longer. In the implementation described in the following chapter, the period of time that backing is valid for is limited by specifying a time-out in the backing certificate. This period therefore is kept under the control of the backer.

What Exactly the Backer Consents to

In all of the security policies presented, backing is sought on a one-off basis to perform some specific named action. At the Secure Shared Object level of abstraction this translates into invoking one named method upon a named object once and once only. The certificate of backing therefore must reflect these requirements:

Backing once given must only be used:

BR3: To execute some specific named method with specific arguments;

BR4: Upon some specific named object;

BR5: To execute it once only.

This does raise the question of how exactly methods and objects are uniquely named. A method in most programming languages is uniquely defined in its class by its signature: that is its name, the number and types of its arguments and the type of the value it returns. This therefore needs to be captured inside the certificate of backing.

How a particular instance of an object is uniquely referred to is individual to the implementation of the object. In the case of Secure Shared Objects introduced in Chapter 7, objects are given a unique string name. This corresponds to the name of the underlying process group.

Note that backing is given for a principal to invoke a method with specific values for the arguments (if the method takes arguments). There is a great difference between for example, a principal asking for backing to transfer £10 as opposed to transferring £1,000,000. A potential backer will certainly want to differentiate between these two. Therefore a request for backing will contain proposed values for any arguments and these are then included by the backer in the certificate.

All backing certificates when constructed by the backer are intended to be used once only. There are no examples in the policies observed that demand otherwise, so there must be some way that the implementation can ensure that certificates are only used once. Further, it must be possible to ensure that distinct sub-sets of the entire set of backings obtained are not used separately, i.e. it is not enough simply to be able to spot that a certificate has been presented to the Guard before.

Take as an example a policy that requires that two members of a particular role back a principal to perform some method. Suppose that there are ten possible backers and all give consent for the method to be performed. In order to ensure that the method is only performed once it is not sufficient just to recognise that a backing certificate has been used before. If this was the case, the principal could make five repeated attempts to perform the operation with different pairs of certificates (each from different backers). This must be prevented, i.e. once the operation has been performed, all the backing certificates relating to the same request must be unusable.

Making the Backer Aware of the Semantics

Asking a potential backer to make up his mind about giving consent for a principal to invoke a method upon an object may not be realistic. This is because method invocations upon programming objects are not in his view of the system. It is not really reasonable to expect the participants in the collaborative task to make a decision about whether to grant backing in response to a request which looks like a method call, for example:

> *"can Principal A perform updateRecord("Dose", 5)?"*

This might not mean much to the person. In order for the potential backer to be able to reach a reasoned decision he must be sure exactly what backing is being requested for, it is necessary to bridge the gap between the shared object level of abstraction at which objects are protected and the user's view. Hence backing requirement BR6:

BR6: Requests for backing when presented to a user should be consistent with the users' view of the system.

The best and most obvious way of communicating what backing is being sought is if the potential backer is presented with a natural language statement. For example:

> *"can you back Principal A's request to update the Medical Record of Patient X with a Dose of 5 units?"*

This complicates matters however, as there now exist two descriptions of the nature of the backing required: the natural language request as presented to the user and the formal description of backing as described in the ACL. This means that there must be a secure relationship between the two since it is obviously essential that the authentic natural language request is presented to the potential backer and not some bogus request presented maliciously. Chapter 9 will show how this can be achieved in an implementation.

8.4 Summary

This chapter has introduced in a more complete way two types of security policy that are commonly found outside of the computing realm. If computer supported security sensitive group tasks such as those dealing with medical or financial information are to be properly protected from malicious activity then the same level of protection needs to be applied inside the computer system as is currently afforded outside. This means that mechanisms for state-dependent access control and the concept of backing must be designed and implemented.

This chapter has covered some of the key issues that need to be addressed before an implementation can be considered. Such an implementation can now be discussed in the following chapter.

9

Access Control for Groupware - Implementation

This chapter will cover some of the more interesting points surrounding any implementation of the new access control concepts that were introduced in Chapter 8.

As the system runs, the access control list (ACL) is consulted when necessary by a Guard. The Guard assesses whether principals have the right to do what is being attempted. Section 9.1 will show that extending the Guard's actions to allow the consultation of state is straightforward. Hence the implementation of state-dependent access control does not raise many further implementation issues. The implementation of backing is not so clear cut.

In addition to the ACL, Guards typically base access control decisions on the results of evaluating credentials. These often take the form of signed certificates of role membership (and possibly delegation). Assessing whether a principal has accumulated the required degree of backing is a similar job. Consequently a principal's intention to back is also contained within a certificate. Section 9.3 examines in greater detail how principals can initiate requests for backing and the exact structure of the backing certificates that are returned.

Of course it is the intention that the access control concepts described here all be integrated with a system of protected shared objects that are in reality replicated. Such a system was described in Chapter 7. It was implemented using the group communication system for groupware described in Chapters 4 and 5. However the fact that object data is replicated does not on the whole affect the implementation of the access control concepts. This is because access control belongs at a higher level of abstraction than the level at which replication occurs. The layers were depicted in Figure 3.3. The relevant part of this diagram is reproduced below in Figure 9.1:

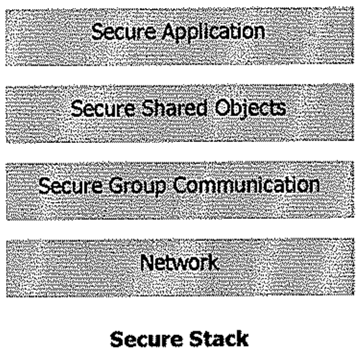


Figure 9.1 Layers of secure groupware. Access control fits in at the level of Secure Shared Objects. Replication is dealt with at the Secure Group Communication Layer.

This separation of replication from access control means that the issue of replication is largely irrelevant in this chapter. However it is important to bear in mind that the performance requirement of groupware necessitates replication in the underlying levels.

Another important issue is addressed in this chapter. This is the issue of making easier the specification of the rights of large numbers of principals to access large numbers of objects. The work here was designed to be incorporated into a task-oriented framework for access control in groupware. The task concept limits the scope of shared objects, access control and hence any associated communication. Section 9.4.1 describes the task framework of the PerDiS project and Section 9.4.2 extends the concept to incorporate the ideas of state-dependence and backing.

Finally Section 9.5 attempts to evaluate the access control concepts presented by justifying the work further with the observation that the procurement of backing can often be a legal requirement.

9.1 A Guard's Access to Shared Object State

In the Shared Object system, every object potentially has access to every other Shared Object. Since Guards are themselves merely objects, then allowing them to inquire the state of other objects is not a problem. Figure 9.2 demonstrates this.

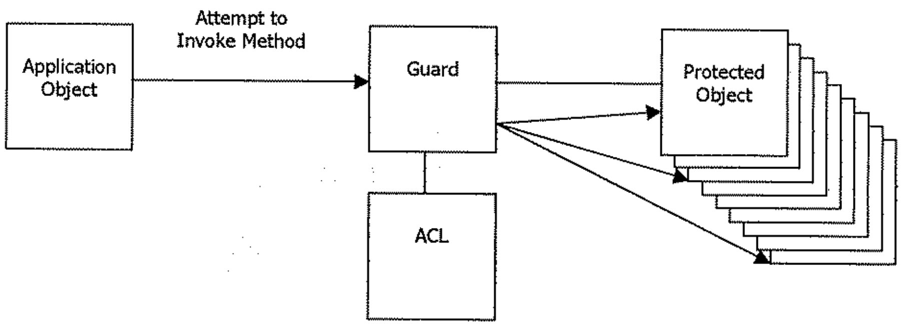


Figure 9.2 In the access control model, it is perfectly feasible for the Guard object to initiate invocations of protected object methods as part of rights evaluation.

As the following section shows, the ACLs contain the references to objects and the method that should be called in order for the access control decision to be made.

9.2 Expressing Dynamic Security Policies in Access Control Lists

In order to enforce state-dependent policies and policies that require principals to obtain backing, it must first be possible to express such policies within the system. The mechanism explained here extends simple access control lists in order for them to contain calls to access control methods. The Guard invokes the access control methods whenever it consults an entry in an ACL that contains such a method. The results returned by the method are then used in evaluating the principal's right to invoke the protected method.

An access control method can be any existing method of any Shared Object in the system since security policy might rely upon any state from within the system. For the expression of policies that require backing to be obtained, special-purpose access control methods are introduced. A formal description of the syntax of the access control lists described here is given in Appendix C.

Before continuing it is necessary to say something about naming. Obviously the access control lists need to be able to refer to the objects they protect as well as the principals and the roles that they are entitled to take on. Additionally in our extension to the ACLs, it is necessary to be able to refer to the methods of objects as well. No convention for naming is necessary for the functioning or understanding of the ACLs described here. However it can be assumed that the protected objects are the Secure Shared Objects as described in Chapter 7 and are uniquely referred to in the manner described in Section 7.3, i.e. with unique string names. References to principals and roles must also obviously be unique. A suggestion as to how this is achieved in a scalable way is discussed in Section 9.4.2. Uniquely referring to object methods is tackled by the object oriented language that implements them. All example references to object methods given here and the complete syntax presented in Appendix C follow the syntax of the Java language.

9.2.1 Expressing State Dependent Policies

In this section it is shown how the syntax of ACLs can include calls to object methods so that the results returned from those invocations can be used in the evaluation of a principal's rights. This in turn allows a more dynamic set of policies to be expressed. The first step is the observation that access control lists can be seen simply as boolean expressions. Take the following list of principals that could be associated with some method of an object as an example:

PrincipalX, PrincipalY, PrincipalZ

This could be rewritten in a manner that resembles a boolean expression in a programming language. P is the principal attempting to invoke some method of an object. P is granted access if:

(P == PrincipalX) or (P == PrincipalY) or (P == PrincipalZ)

The notion of roles can be easily incorporated. The following expression captures a requirement that principal P must be entitled to take on the role of a Doctor or a Manager:

($P \in$ Doctors) or ($P \in$ Managers)

Lampson [LABW92] also introduced the notion of conjunctions in access control:

($P \in$ Doctors) and ($P \in$ Managers)

Stipulating that the principal P attempting to invoke the method must be a Doctor and a Manager in order to succeed.

The expressions above demonstrate the obvious parallels between a boolean expression found in a programming language and entries in an access control list. However specifying security policy in this format is a little verbose and lacks the obvious simplicity and simple semantics of a list. For that reason the list structure is retained in the access control lists described below (formally and more completely specified in Appendix C). Therefore a comma in the ACL is equivalent to boolean 'or' and principal and role identifiers appear on their own rather than with equality or set membership operators as above. All ACL entries from this point on conform to the syntax specified in Appendix C.

A programmer writing a boolean expression would expect to be able to include calls to boolean methods in any conditional statements. Here this idea is adopted in order that ACLs can refer to methods in the same way. Hence an ACL entry can now reflect a state-dependent policy. Hence the following example ACL entry for a hypothetical appendNotes method:

```
void appendNotes(String s):

        Physician and MedicalRecord.patientOverEighteen ( );
```

where *MedicalRecord* names some object which is the target of the *patientOverEighteen ( )* method call. The entry expresses the following policy that could be used to protect some object method:

> *"Physicians can append notes to a medical record if the patient is over eighteen"*

Alternatively, if there is a method that returns the age of a patient, then the same policy can be expressed as follows:

```
void appendNotes(String s):

        Physician and ( MedicalRecord.patientAge ( ) >= 18 );
```

Environmental policies can be expressed in a similar fashion because the implementation of methods that are called have potential access to time and other details maintained by the operating system.

How the target object of the method calls are named within an ACL is deferred to Section 9.4. The solution integrates the new access control concepts with an existing 'task oriented' approach to grouping and naming objects that form part of a shared environment.

9.2.2 Expressing Parameter Dependent Policies

Expressing policies that refer to the value of the parameters of an attempted invocation can also be achieved by treating ACLs as boolean expressions. In this way expressions involving the parameter names and boolean operators can be incorporated.

Take the following entry for a moveWall method. It expresses the policy that was introduced in Section 8.2.3 taken from the PerDiS designers that restricts Engineers from moving a wall more than 5 centimeters:

```
void moveWall (int distance):

        Engineer and (distance <= 5);
```

Another similar policy might state the same restriction on engineers moving walls, however with the added addition that managers can move the wall any distance. This would be expressed as follows:

```
void moveWall (int distance):

        Manager, Engineer and (distance <= 5);
```

Note that in these ACL entries a comma has been used to represent the logical operator 'or' in order to preserve the list nature of the expression. This is just aesthetic and not an important issue. Obviously the commas have the highest precedence of all the operators.

9.2.3 Expressing Policies that Require Backing

Two forms of backing policy were identified in the previous chapter. These were, firstly: backing from a specific quantity of backers and secondly: backing from a proportion of potential backers. In both cases the potential backers come from the group associated with some organisational role.

A requirement to obtain backing can be stated with two method calls: 'exactly' and 'atLeast' representing specific quantity and proportional backing respectively. Hence the following access control list entry for some object method:

```
Doctor and exactly ( 2, Doctors );
```

For expressing the specific quantity backing policy: 'A doctor with the backing of two other doctors'. Also the following entry:

Chairman and atLeast (1/2, Directors);

For expressing the proportional backing policy: 'The chairman with the backing of half or more of the board of Directors'.

Obviously these are not methods that the application programmer implements. These form part of the access control system. In fact it is not necessary to know the target object of the method calls in order to write and understand an ACL entry that comprises backing. Conceptually however, the methods can be thought of as being implemented by the Guard object, since it is the Guard that is responsible for verifying that the necessary level of consent has been procured.

Recall from the previous chapter that every backing statement has associated with it a natural language description of the backing being sought. This is because the method and object names that form the invocation that is actually being backed are not in the user's view of the system. This association between the method and its natural language description must be a secure one, because the backer's decision to back is based upon it. The ACL therefore is the sensible place to form this relation. For every method that could be protected with an ACL entry requiring a principal to obtain backing there is an associated string describing the method. Hence a complete ACL for a method for deleting a medical record might look as follows:

```
void deleteRecord (String patient)

"delete the medical record of patient $patient":

        Doctor and exactly ( 1, Doctors );
```

The $patient placeholder will be substituted for the value of the argument when the request is presented to a potential backer. It will appear upon the potential backer's screen as:

"Request from 'Doctor X' for backing to delete the medical record of patient K. Jones"

This appears together with the name of the Shared Object that the method belongs to. Appendix C contains the formal description of the syntax for ACLs.

9.3 Proof of Backing

Being able to express a need for a principal to obtain backing is only part of the solution. It must also be necessary for the principal to be able to collect backing and subsequently prove it. Recall the six requirements of any implementation of backing from the previous chapter:

BR1: Backing should be provable: i.e. contained within a digitally signed statement of consent;

BR2: Backing once granted must be limited in duration to a specified period of time;

BR3: A specific instance of one principal's backing of another must be limited so that it can only be used to execute some specific named method with specific arguments...

BR4: upon some specific named object...

BR5: and to execute it once only;

BR6: Requests for backing when presented to a user should be consistent with the users' view of the system.

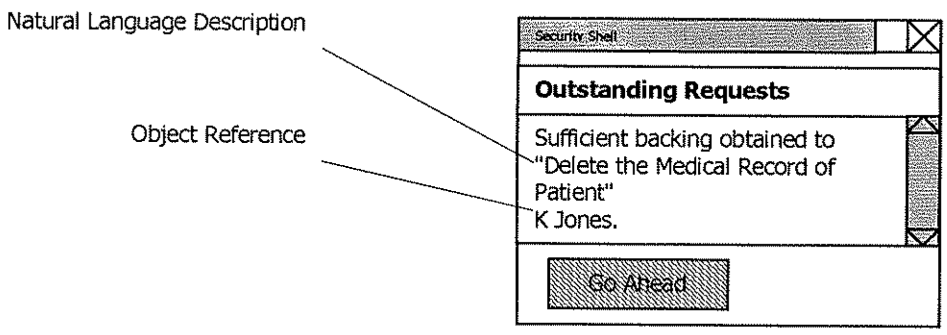
In this section each of these requirements will be addressed and exactly how each can be satisfied by an implementation will be demonstrated. In fact this section starts with BR6 and shows how a request for backing could be initiated and how a user might be prompted to give consent.

9.3.1 Interacting with Potential Backers

In Chapter 3 (Section 3.3.2) the idea of the security shell was introduced as it was used in the PerDiS system. To recap, it is a means of bridging the gap between the user and the access control layer. This is needed when the application has no notion of access control built into it. In the PerDiS system this provides a mechanism for users to assign people to roles and modify access control lists.

This idea relieves the application program from the responsibility of providing interaction between the user and the security mechanisms at the Shared Object level. Additionally it allows existing non-secure applications to be converted for use in a secure setting with a minimum of inconvenience. Of course new applications might well work best if access control is presented to the user in a manner consistent to whatever abstraction the application presents. On the whole though, access control can be hidden from the user, at least until the user attempts to exceed his rights. However some access control concepts such as delegation and backing do require direct user actions and the security shell can facilitate this.

A security shell allows the user to both make a request for backing and to give consent to (or reject) incoming requests. Upon attempting an action that involves a method invocation the security shell asks for confirmation for the request to be sent out to the appropriate group of potential backers. The security shell of a potential backer alerts the user when the request is received. If the user chooses to consent, the security shell replies with a signed certificate. The exact construction of backing certificates is discussed in the following section. Once the security shell of the original request has received a sufficient quantity of positive responses, i.e. certificates, the method invocation can be attempted. This might result in the security shell highlighting the request or alerting the requestor in some other way. See Figure 9.3.



Outstanding Requests

Sufficient backing obtained to "Delete the Medical Record of Patient"
K Jones.

Figure 9.3 How the Security Shell may look when being used by the requestor to display the progress of outstanding requests. The 'Go Ahead' button is shown shaded because insufficient backing has been currently obtained.

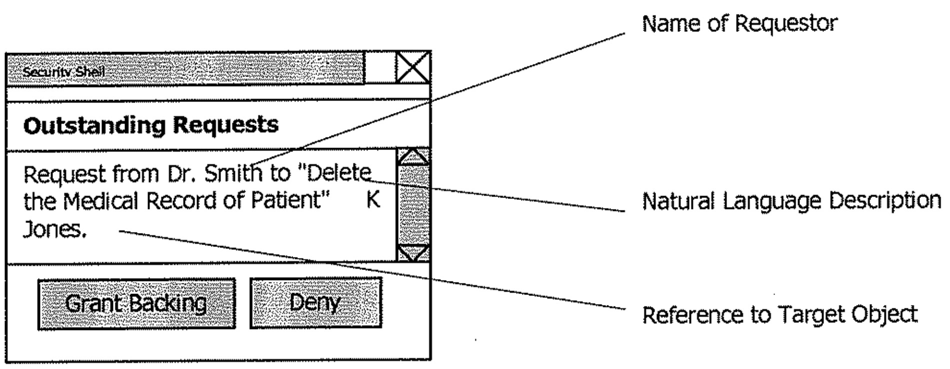

Security Shell
Outstanding Requests
Request from Dr. Smith to "Delete the Medical Record of Patient" K Jones.
Grant Backing
Deny
Name of Requestor
Natural Language Description
Reference to Target Object

Figure 9.4 How the Security Shell may look when being used by a potential backer to display requests for backing. The natural language description of the backing being sought makes the nature of the backing obvious to the backer.

The security shell must also display the natural language description of the backing request to the potential backer, this fulfils requirement BR6 and this is shown in Figures 9.3 and 9.4. Recall from the previous section that this description enables the backer to properly understand what consent is being sought, because the method and object names are not at an appropriate level for the user's understanding. If the method call that corresponds to the request takes any arguments, then these are integrated into the natural language request as described in Appendix C and Section 9.2.3.

9.3.2 Certificates of Backing

Chapter 8 and BR1 have already shown that one principal's backing for another must ultimately be represented within a digitally signed certificate of consent. However the details of what exactly is contained within the backing certificate have not yet been discussed. The contents are a direct consequence of the final semantics of backing that were resolved in Section 8.3.3 and resulted in requirements BR3, 4 and 5.

The scope of the certificate must of course be limited to one principal, but also limited to one object and one method on that object. This can be achieved simply by naming the principal, object and method uniquely inside the certificate[2].

The value of the arguments for the method call must also be included in the certificate. Backers obviously don't give their consent to invoke the method without any set of arguments in mind. As was made clear in Section 8.2.3, the backer when presented with a request from another to back is given the values of any arguments that the requestor intends to use. The backer then includes these in the signed body of the certificate and the Guard can subsequently verify that the method is actually invoked with these values.

Finally, satisfying backing requirement BR2 demands that an expiry time be added into the certificate, in fact this is a standard requirement for most signed certificates. Hence the structure of the certificate is as follows:

- The object reference.
- The method signature.
- The values of any arguments
- The unique backing request number.
- The natural language statement of what the backing is for.
- The expiry date/time after which the backing statement can no longer be used.
- A signature signed with the private key of the backer.

The unique backing request number is used to ensure that BR5 is satisfied. Recall from Section 8.3.3 that a Guard must recognise any attempt to use backing certificates that relate to the same request to invoke a method more than once, i.e. only the first invocation should succeed.

The backing request identifier uniquely identifies a particular request. It is generated by the access control system and is included in a principal's request for backing. Any backers then incorporate the identifier within the certificate and it is

[2] Unique shared object naming was discussed in Chapter 7 (Section 7.3) and uniquely naming the method in question is a matter of stating the method signature as it appears in the ACL. Naming principals has also been discussed previously in this chapter and Chapter 7.

left to the Guard to recognise any that have been used before. Making the identifier an increasing sequence number (for each principal) relieves the Guard of having to store every previously used identifier.

The natural language description is also incorporated into the certificate. This demonstrates to anyone who validates the certificate what the action the backer thought he was backing.

9.3.3 Collection of Backing

The backing certificates obviously come from many places and hence must be collected together into one place so that the requestor of backing can collect them and present them to a Guard. As already stated, the process of requesting and collecting backing may not be a swift one. It is not desirable that policies that require the backing of others to be procured are limited to highly interactive applications. It is quite conceivable for example that the final collection of the required amount of backing might take over a day. Hence it is not possible to rely upon synchronous communication between requestor and backer for the transfer of a certificate. An asynchronous solution is needed, which will allow the parties involved to go on and off-line during the procedure.

The problem is solved with the introduction of shared objects called 'Outstanding Backing Objects' (OBOs). When a backer decides to go ahead and attempt to collect backing, a new OBO is created. This would be done by the security shell if one is being used, or by the application if that has the functionality. The OBO contains details of the request and will act as a repository for any certificates that are generated. Potential backers can then pick up the details of the request from the OBO and install a certificate by invoking its methods.

Ideally when sufficient backing is collected, the initiator should cancel the corresponding OBO. In a secure system any apportion of responsibility should be done with care, however it is not essential to the secure running of the system that the outstanding request is cancelled since it will naturally time-out anyway, as will the Guard's store of already used backing request identifiers.

Note that the OBO can be a Secured Shared Object as described in Chapter 7 and hence can in reality be replicated amongst the current participants in a group activity. In this case new participants receive details of outstanding requests for backing when they begin to participate in the activity. This is facilitated through the state transfer mechanism in the underlying secure group communication system. In this way the functionality of OBOs can also include auditing, since during their existence they contain the details of every principal that gave backing to another. There is no reason why the version of the OBO on the distributor cannot write these details to a file before it ceases to exist. If it is necessary to prove in the future that backing was attained then retention of the certificates beyond the successful invocation is essential. Auditing is one way in which this could be achieved.

The mechanism through which potential backers can find out about all existing and relevant OBOs and hence obtain references to them is intimately linked with the mechanism for making access control less complex and scalable. This is the topic of the following section.

9.4 Tasks for the Management of Scalability and Complexity

This section is included in order to relieve any concerns that the access control system might be overly complex or that it does not scale well. The concept of tasks is introduced as originally proposed by Coulouris and Dollimore [CDKR97] [CDR98a] [CDR98b] [CD94b], although the concept has been discussed elsewhere [SAN96] [TS94]. Tasks can be used to tackle the following problems with access control for groupware:

- **Complexity**: It may be laborious to specify the rights of a large numbers of principals to access a large number of objects. Hence the task concept allows generalisations to be made about objects and principals' rights.
- **Scalability**: Assigning objects and rights to particular tasks means that their scope is limited, no matter what the proliferation of tasks.

9.4.1 The Original Task Framework

In any system that requires access control there are by its nature a number of principals that are potentially involved. This number could be large, hence making generalisations about the principals' roles is an obvious way of simplifying the construction of ACLs. This idea is part of Lampson's access control model for distributed systems. In an object-oriented groupware system there are not only potentially large numbers of principals but also large numbers of objects that need

ACLs. Generalising about the objects too can ease the complexity of rights specification.

The task framework combines the two notions of generalisation. A generalised task will have a set of objects and a set of roles that are common to all specific instances of the task. Take for example a GP's task of administering to a patient. In general each patient will have a set of objects comprising his or her medical record (such as notes, prescription and referral details etc.). Every patient has these objects and the rights to access them (when expressed using generic roles) are the same for each instance of a task. Expressing them once in a generic **Security Template** is more practical than having to do it for every patient. An extremely simple (and far from ideal -as we will go on to explain) example might be as follows:

Task: Administering to Patient

Roles \ Objects	Notes	Referrals	Prescriptions
Patient's GP	read/write	read/write	read/write
Nurse	read	read	read
Receptionist	-	read	read

Figure 9.5 The table shows the rights of roles within the task "Administering to Patient" to access three types of objects used by all instances of the task.

When a task comes into existence, the details that need to be available to the participants such as the ACLs and the roles which take the form of signed certificates are copied from the security template into a Shared Object called the **Task Object**. This is a Shared Object just like those that form part of the application. Changes to the ACLs and the actual principals entitled to take on the roles will be potentially available to all. The task object therefore can then satisfy queries regarding the current membership of a role. Obviously this is important for the implementation of backing.

9.4.2 Tasks and Implementing State-Dependent Access Control and Backing

The original benefits of reduced complexity and confinement of scope for scalability are of course retained when the framework is extended to facilitate state-dependent access control and backing policies. Additionally, the framework can help with concerns about naming of objects and roles and finding out about the state of outstanding requests for backing.

The access control concepts introduced in this thesis require that ACLs be able to refer to specific objects and roles. Since every object and role forms part of some task, they can therefore be referred to through their task (for example, the patient notes in the specific task instance of administering to a patient X). Additionally, it is possible for the tasks themselves to be named by universal resource locators (URLs) as are other Internet abstractions.

Outstanding Backing Objects have already been introduced. However an explanation of how potential backers can discover that a new request has been issued was deferred. The Task Object provides a solution. As has already been explained the Task Object is a repository for other information that is specific to a task such as ACLs and role membership. Additionally therefore placing references to OBOs in the task object provides a way for principals to discover the existence of a new request. When a principal decides to proceed and attempt to collect backing for some action, the OBO is created and a method called on the Task Object which updates it with a reference. In this way requests for backing are confined to a task. Again note that Task Objects and Outstanding Backing can be Secure Shared Objects and hence can be replicated at the machines of any participant in a task. In reality therefore, the installation of a reference to an OBO amounts to a single multicast to all current participants in the task.

9.5 Evaluation of Access Control for Groupware

The last two chapters have introduced just two techniques that enable more accurate enforcement of security policies. State-dependent access control and the concept of backing are techniques that are particularly applicable to secure groupware applications because they allow more dynamic security policies to be enforced. This work is also covered in a technical report [RD97c] and has been presented at two conferences [RD97d] [ROW98].

The examples given in the chapter together with those in the appendix show that state-dependence and the technique of backing are both prevalent in real security policies and this on its own is justification for this work. However it is not claimed here that these two techniques alone will enable the enforcement of all policies, indeed even before completion of this research other techniques such as veto for example have been observed in the literature from which the presented policies were drawn. Nevertheless, any techniques that allow a closer representation of real-life policies can only be a good thing.

Evaluating this work properly would require an implementation to be used over a period of time by multiple parties collaborating using groupware. This is obviously beyond the means of this project. Evaluation therefore is forced to be somewhat speculative and consist mainly of the justification and motivation for the work that has already been covered in the early sections of Chapter 8.

One very compelling justification of the backing work comes from the field of law. Very often it is a legal requirement for one person to procure the consent of others, such examples are particularly numerous in the fields of medicine and company law. Both of these areas have been drawn upon in the policy examples contained within Appendix B.

Take as one example, the law surrounding decision-making in companies run as a partnership. Decisions must be taken collectively and there are legal guidelines for the level of consent amongst the partners that must be achieved before certain types of resolutions. Ordinary resolutions for example require that there is a majority in favour of the decision: effectively therefore the chairman is required to obtain the backing of 50 per cent of the partners. Other types of resolutions require different levels of backing: special and extraordinary resolutions require a 75 per cent majority and elective resolutions require unanimity.

If group applications are to facilitate the negotiation and decision making that forms part of running a company then the digital representation of backing would be important. Conventional message authentication techniques could, in retrospect establish who said what in a distributed meeting for example, but in order to say for certain if participants backed another in a specific action then the tighter proof provided by the system outlined in this thesis would be far more convincing. Simple messages relating to a group discussion might not be specific about the action being backed, or they may not contain a time at which the backing expired for example. This alone justifies the implementation of backing.

Actually backing can have justification beyond the realm of access control. In many instances it would be useful for a principal to collect backing for an action even though laws and regulations do not force them to do so. Being able to prove in retrospect that a good level of consent existed at the time of an action could prove useful in a legal sense. Medical examples highlight this: for example a doctor making a potentially controversial decision might collect backing from other colleagues around the country in order to subsequently add weight to his actions.

10

Conclusions

10.1 Summary

This thesis has presented ideas that are aimed at allowing computer aided group work to be used for secure applications. Often security sensitive tasks are left without the benefit of computer support because of a lack of appropriate security mechanisms. The work has been divided into two main areas corresponding to different levels of abstraction between the application and the network. These were Secure Group Communication and Access Control. At both levels security has been introduced with interactive groupworking in mind and at both levels security is integrated appropriately with the abstraction being presented.

- **Secure Group Communication for Groupware**

  One of the most important attributes of any groupware system whether being used for secure applications or not is the performance that it can deliver: a sufficiently fast system can greatly enhance the productivity of distributed group tasks, but a slow system will only hinder its effectiveness. In order to meet this performance demand, a distributed groupware system can replicate data so that it is located wherever it is needed. However, replicating and maintaining data is complex and so often the application programmer will turn to systems software that can hide the replication behind a convenient abstraction. Secure group communication software provides a process group abstraction and is intended for replicating secure data.

  This thesis has introduced a secure group communication system that is innovative because it makes appropriate assumptions about the trust afforded to group members. Similar existing systems that have all been developed for fault-tolerance do not make trust assumptions that are appropriate to groupware. The system presented here strikes an appropriate balance between trust and performance, which makes it uniquely suited to groupware. Timed performance tests and a logical proof of authentication supported the system's suitability.

- **Access Control for Groupware**

  Many groupware application programmers use a shared object abstraction for hiding both replication and communication. This can conveniently be built on top of a group communication system. This thesis has presented a secure variant of this abstraction that uses in its implementation the secure group communication system for groupware. The system presents to the programmer the illusion of shared objects. Any instance of an application can map these objects into its local address space and can invoke the objects' methods as if they were entirely local. Any changes to an object's state as a result are transparently visible to all. Security is consistent with the level of abstraction that the programmer sees, because the ability to invoke shared objects' methods can be controlled.

  The access control enforced by the Secure Shared Objects system follows the conventional access control model of Lampson et al [LABW92]. However this was not derived with groupware in mind. Computerised groupware introduces new ways of interacting and negotiating that mimic real-life interaction. In secure group activities, some of the things being negotiated are rights. Hence rights in secure groupware applications are far more dynamic than in conventional applications. This causes problems with specification because conventional access control lists are relatively static: changes to rights generally require some intervention.

  This thesis has introduced a step towards making rights specification and enforcement more dynamic and Lampson's access control model is extended accordingly. Access control lists are now able to contain something that more closely represents security policy rather than merely a list of rights.

  First the notion of allowing rights to depend upon the state of the system has been investigated. Rights often change in parallel with the state of the system. Many security policies taken from secure but not necessarily computerised activities have been listed to demonstrate this observation. Secondly the concept of backing was introduced. Splitting responsibilities between two or more people is a very common way of reducing the possibility that a corrupt individual acting alone can do harm. A method of allowing the concepts of backing and state-dependence to be expressed in an access control list were described together with implementation details that would allow such policies to be enforced. The concept of backing was further supported by the observation that obtaining support and consent for actions is often a legal requirement.

10.2 Future work

The work surrounding group communication for groupware is relatively complete, hence most of the future work surrounds access control. However the implementation of the group communication system was only experimental and hence there is room for this to be developed into a completely useable system. Section 5.3 did explain how dividing the system into layers allowed the system to be flexible and lightweight. The programmer is given the choice of selecting only the layers that provide the essential semantics for the application and thus can maximise performance. This is the same idea as that used by other group communication systems, however in those cases layering for choice allowed the programmer to select the minimum necessary communication semantics. Extending this idea to the security layers was sensible. Given that the particular model of trust chosen was not the only option and that in other trust situations a different choice might be more applicable, then the design and implementation of alternative layers for the system is appropriate.

One topic that was not explored to its end in this thesis was the possibility of allowing methods that actually update state to be included in access control lists. It was implicit that only state inquiries could be included in rights evaluation. This was chiefly because none of the security policies studied as part of this work suggested that it was necessary to include update methods. However the idea opens up interesting new possibilities for access control. It means that an attempt to invoke a method (both successful and unsuccessful) could result in a state change in the system. This would be useful for security auditing for example, since a method that records the attempt could be automatically invoked.

The TAOS operating system that was reviewed in Chapter 3 treated rights as programming level types and these could be sent as parameters to and returned as a result of a procedure call. Combining this notion with the concept of backing allows easy implementation of elections. For example, a method that returns a right which allows a principal to take on the role of chairman could be protected with a backing policy that demanded that the principal invoking it got the consent of the majority of the board.

The secure group communication system tackles security for groupware at the communication level and the access control concepts presented here offer security for groupware at the level of shared objects. However security has not been tackled at the level of the application. Obviously every application is different, however the basic concepts of principals, role membership and backing etc are all apparent at the application level. However the concept of protecting the methods of programming level objects is not in the user's view of the system, but it is at this level that security policy must be specified by the construction of ACLs. This gap in semantics was highlighted in the work presented here on backing by the requirement for natural language statements to enable the user to have a good understanding of what it means to back a request to invoke a method upon an object.

The problem is therefore that security policy exists at a higher level than that at which it must be specified. Applications may offer ad hoc solutions to this problem, but in general shared objects may be shared by many and different applications. Offering a general solution for bridging this gap therefore is a topic for further research.

One possibility is the use of policy specifiers. These would most likely be natural language descriptions of common security policy forms which could be selected and combined with descriptions of the operations (method calls) in order to form descriptions of the policy. These could then be automatically translated into access control lists for the methods, possibly taking the form of the language for ACLs presented in Appendix C.

Appendix A

Proof of Authentication

This appendix uses the technique of Burrows, Abadi and Needham to prove that the authentication goals of the group join protocol are met. For a full description of the technique the reader is referred to the BAN paper [BAN90]. The authentication goals were originally stated in Section 4.2.2 as:

S3. A prospective member can authenticate the group that it wishes to join.

S4. A group can authenticate a prospective member and refuse admission.

S3 amounts to the new member believing that the message granting membership (JOIN 2a) came from the trusted Distributor and no one else. This is achieved when the new member believes that it shares a secret key with the Distributor.

S4 is achieved when the Distributor believes that it is really communicating with the new member and not some process masquerading as that principal. This is achieved when D believes that the new member believes that it is sharing a secret key with D.

The first part of this proof shows that when a certificate for a principal X is observed by some principal Y that is signed by some party CA that Y trusts to certify public keys, then the statement inside the certificate is believed. Hence when the public key certificate is observed, the key contained within it is believed to belong to the stated source.

Initially, Y knows the public key of the authority CA, i.e. Y believes PuCA, also Y believes that CA is an authority as regards to X's public key, in other words Y believes that CA controls PuX. Finally CA itself must believe the real public key of X, i.e. CA believes PuX.

Initial beliefs:

Y believes	**CA believes**
PuCA	PuX
CA controls(PuX)	

Y sees the certificate and the goal is to prove that Y believes the public key of X. The certificate is signed with the public key of the trusted certification authority CA. The certificate is constructed as follows:

[PuX, Expiration] PrCA

Idealised: fresh (PuX), { PuX } PrCA

Applying the BAN Message Meaning rule for public/private keys:

since Y believes PuCA and Y sees { PuX } PrCA

then Y believes CA said (PuX)

Applying the BAN Nonce Verification rule:

since Y believes fresh (PuX) and Y believes (CA said (PuX))

then Y believes (CA believes (PuX))

Applying the BAN Jurisdiction rule:

since Y believes (CA controls (PuX)) and Y believes (CA believes (PuX))

then **Y believes PuX**

Hence in the remainder of the proof, if some principal sees a public key certificate then the key contained within is believed. In the following descriptions, D represents the Distributor and A the prospective member. Therefore:

since D sees A's certificate, from the proof above:

then **D believes PuA**

since A sees D's certificate, from the proof above:

then **A believes PuD**

The initial beliefs of the Distributor (D) and the new member (A) are therefore as follows:

A believes	**D believes**
D controls $A \leftrightarrow D$	
PuCA	PuCA
CA controls (PuD)	CA controls (PuA)
PuD	PuA

The join request JOIN 1 plays no part towards the beliefs of either party and so is not included in the proof. In response to the join request D replies with JOIN 2a:

D → A: [{ JOIN2a, Nonce, SN, M'ship, State, A ↔ D, GEK } PuA] PrD

Idealised: fresh (A ↔ D), { A ↔ D } PrD

Applying the Message Meaning rule for public/private keys:

since A believes PuD and A sees { A ↔ D } PrD

then A believes D said (A ↔ D)

Applying the Nonce Verification rule:

since A believes fresh (A ↔ D) and A believes (D said (A ↔ D))

then A believes (D believes (A ↔ D))

Applying the Jurisdiction rule:

since A believes (D controls (A ↔ D)) and A believes (D believes (A ↔ D))

then **A believes (A ↔ D)**

Hence the first authentication goal is achieved. A's response is the JOIN 3 message:

A → D: [{ JOIN3, A, A ↔ D } PuD] PrA

Idealised: fresh (A ↔ D), { A ↔ D } PrA

Applying the Message Meaning rule for public/private keys:

since D believes PuA and D sees { A ↔ D } PrA

then D believes A said (A ↔ D)

Applying the nonce verification rule:

since D believes fresh (A $\leftrightarrow$ D) and D believes (A said (A $\leftrightarrow$ D))

then **D believes (A believes (A $\leftrightarrow$ D))**

Appendix B

Dynamic Security Policies

This appendix lists some examples of dynamic security policies with references, some of which are referred to in the main chapters of this thesis. The list of potential policies to support state-dependent access control and the concept of backing is vast, however this section concentrates on interesting medical and financial examples. The one thing that all these examples have in common is that they cannot be conveniently or accurately expressed and enforced using conventional access control techniques (if at all).

Security Policies that are State Dependent

P1. [AND96] "[In access control lists] *groups may be used instead of names"* ... *"Some extra restrictions may be needed in defining groups; for example, the group might be any clinical staff on duty in the same ward as the patient."*

P2. [AND96] *"No-one shall have the ability to delete clinical information until the appropriate time period has expired."*

P3. [EDW96] An example policy for controlling when a principal is interrupted: *"Don't let people bother me when I'm working on my thesis"*

P4. [HHLS94] Regarding legal arrangements between partners running a company: *"The agreement may state that a particular partner (with limited experience) only has authority to make contracts within certain limits."*

P5. [HHLS94] Regarding the right of shareholders in a company to call an Extraordinary General Meeting: *"Members holding at least 10* *per cent of the company's shares have the right to requisition a meeting"*

P6. [KA81] *"Tellers can't cash personal cheques"*

P7. [KA81] *"Bank tellers should not be able to process transactions that involve themselves."*

P8. [TIN90] *"A nurse under the supervision of a physician may access the patient's medical record for medical treatments and other medical care activities. He or she can only access the portion of the medical record and data which is relevant to the nurse's duty and functions."*

Security Policies that require Principals to Obtain Prior Backing

P9. [DRA96] *"A clinician must obtain the backing of a majority of a patient's carers in order to prescribe new treatment."*

P10. [GS86] Policy for installing new entries in a group calendar: *"Participants vote on the alternatives" ... "If one of the proposals is confirmed, it is permanently installed in the calendar."*

P11. [HHLS94] In a company that is a partnership: *"Unless the agreement provides to the contrary, all partnership decisions will be made on the basis of a simple majority."*

P12. [HHLS94] Also in a company that is a partnership: *"decisions on changing the nature of the business or on introduction of a new partner require unanimity."*

P13. [HHLS94] In a private company with shareholders: *"A special resolution requires a 75 per cent majority [of shareholders] for it to be passed.", "An ordinary resolution is passed with a simple majority", "An elective resolution is only carried if it receives unanimous consent."*

P14. [KA81] *"A teller must seek permission to adjust a special purpose bank account for reconciling differences between the actual amount of money taken and the recorded amount (should any discrepancy occur)."*

P15. [KA81] *"The person opening mail and making records of received cheques must get his records checked before the data becomes part of the system."*

P16. [MON] *"Certain treatments can only be given with the consent of the patient and a second opinion."*

P17. [TIN90] *"A consulting physician or psychiatrist can read a patient's record by obtaining the patient's permission."* Note that in the context of this paper (under Connecticut law) a patient does not himself have the right to read their own medical record and as a consequence this policy could not be enforced through delegation.

P18. [MM94] From the Abortion Act 1967 (UK Law) *"A person shall not be guilty of an offence under the law of abortion when termination is performed by a registered medical practitioner and two registered medical practitioners have formed the opinion in good faith that the continuance of the pregnancy would involve risk, greater than if the pregnancy was terminated ... subject to the pregnancy not exceeding its 24th week."*

Appendix C

Formal Definition of Dynamic ACL Syntax

The following description uses BNF to more formally describe the syntax of the Access Control Lists initially presented in Chapter 9. The language could be used for example to construct ACLs such as the following:

```
Patient X Medical Record:

 void addNotes(String s):          Doctor;

 String getPrescriptionHistory():   Doctor, Nurse;

 void deleteNotes(int index)

 "delete the medical record of patient $index":

                            Doctor and exactly(1, Doctor),

 void closeRecord():        Doctor and (Today.year()-this.creationYear() > 80);

Design D:

 void moveWall(int distance):       Manager, Engineer and (distance <= 5);
```

The BNF notation below uses the symbol ' | ' to represent alternatives and the curly braces ' { ' and ' } ' to denote zero or more repetitions of the constructs within. Other symbols such as semicolons, commas and curved braces are part of the language.

For brevity identifiers are not defined but can be assumed to be the same as identifiers found in imperative programming languages, i.e. strings of characters without spaces. Also the constructs that must be taken from the programming language that defines the objects being protected (in this case Java) are not defined. The reader is referred to a description of the Java language for these details [SUN96].

ACL	→	Object-Identifier : Method-List
Method-List	→	{ Java-Method-Signature : Entry-List ; } |
		{ Java-Method-Signature
		NatLang-Description: Entry-List ; }
Entry-List	→	Entry | Entry, Entry-List
Entry	→	Element | (Entry) | Entry **and** Entry
NatLang-Description	→	" String " |
		" { String $Java-Parameter-Identifier } "
Element	→	Principal-Identifier | Role-Identifier |
		Boolean-Expression
Boolean-Expression	→	Expression Boolean-Operator Expression
Expression	→	Number | Object-Method-Identifier |
		Parameter-Identifier |
		Expression Operator Expression
Object-Method-Identifier	→	Java-Method-Identifier
		Object-Identifier . Java-Method-Identifier
Principal-Identifier	→	Identifier
Role-Identifier	→	Identifier
Object-Identifier	→	Identifier
Parameter-Identifier	→	Identifier

References

[ACDK97] Anker T, Chockler GV, Dolev D and Keidar I "The Caelum Toolkit for CSCW: The Sky is the Limit" Institute of Computer Science, The Hebrew University of Jerusalem, http://www.cs.huji.ac.il/, 1997.

[AK94] Achmatowicz R and Kindberg T "Object Groups for Groupware Applications: Application Requirements and Design Issues" Proceedings of the European Research Seminar on Advances in Distributed Systems, pp. 269-274, 1994.

[AND96] Anderson RJ "Security in Clinical Information Systems" Computer Laboratory, University of Cambridge, January 1996.

[BAN90] Burrows M, Abadi M and Needham R "A Logic of Authentication" ACM Transactions on Computer Systems, vol. 8, pp. 18 - 36, February 1990.

[BIR93] Birman KP "The Process Group Approach to Reliable Distributed Computing" Communications of the ACM, vol. 36, no. 12, pp. 36 - 53, December 1993.

[BJ87] Birman KP and Joseph TA "Reliable Communication in the Presence of Failures" ACM Transactions on Computer Systems, vol. 5, no. 1, pp. 47 - 76, 1987.

[BSS91] Birman KP, Schiper A and Stephenson P "Lightweight Causal and Atomic Multicast" ACM Transactions on Computer Systems, vol. 9, no. 3, pp. 272 - 314, August 1991.

[BTK90] Bal HE, Tanenbaum AS and Kaashoek MF "Experience with Distributed Programming in Orca" Proceedings of the International Conference on Computer Languages, IEEE, pp. 79 - 89, 1990.

[CD94a] Coulouris G and Dollimore J "Requirements for Security in Cooperative Work: Two case studies" TR 671, Department of Computer Science, Queen Mary and Westfield College, University of London, May 1994.

[CD94b] Coulouris G and Dollimore J "Protection of Shared Objects for Cooperative Work" TR 703, Department of Computer Science, Queen Mary and Westfield College, August 1994.

[CDKR97] Coulouris G, Dollimore J, Kindberg T and Roberts M "A Security Architecture for PerDiS in Java" Position Paper for the Workshop on Persistence and Distribution in Java, Lisbon, Portugal, October 1997.

[CDR98a] Coulouris G, Dollimore J and Roberts M "Secure Communication in Non-Uniform Trust Environments" Proceedings of ECOOP Workshop on Distributed Object Security, Brussels, July 1998.

[CDR98b] Coulouris G, Dollimore J and Roberts M "Role and Task-based Access Control in the PerDiS Groupware Platform" To be presented at Third ACM Workshop on Role-Based Access Control, George Mason University, VA, October 1998.

[COO96] Cooper DA "SCOM: A Security Layer for Horus" TR96-1589, Department of Computer Science, Cornell University, Ithaca, New York, June 1996

[CW87] Clark DD and Wilson DR "A Comparison of Commercial and Military Computer Security Policies" Proceedings IEEE Symposium on Security and Privacy, pp. 184 - 194, Oakland Canada, IEEE Computer Society Press, 1987.

[DABW95] van Doorn L, Abadi M, Burrows M and Wobber E "Secure Network Objects" TR 385, Digital Systems Research Center, Palo Alto, California, USA, 1995.

[DM96] Dolev D and Malkhi D "The Transis Approach to Highly Available Cluster Communication" Communications of the ACM, vol. 39, no. 4, pp. 64 - 70, April 1996.

[DRA96] Draper R "Electronic Patient Records: Usability vs. Security, with Special Reference to Mental Health Records" Proceedings of Personal Information Security, Engineering and Ethics, Cambridge, June 1996.

[EDW96] Edwards WK "Policies and Roles in Collaborative Applications" Proceedings of Computer Supported Cooperative Work, ACM, Cambridge MA, USA, pp. 11 - 20, 1996.

[EGR91] Ellis CA, Gibbs SJ and Rein GL "Groupware: Some Issues and Experiences" Communications of the ACM, vol. 34, no. 1, pp. 39 - 58, January 1991.

[GM89] Gray I and Manson S "The Audit Process" Van Nostrand Reinhold (International) London, ISBN 0-278-00044-4, 1989.

[GON89] Gong L "Securely Replicating Authentication Services" Proceedings of the IEEE International Conference on Distributed Computing Systems, pp. 85 - 91, 1989.

[GON96] Gong L "Enclaves: Enabling Secure Collaboration over the Internet" Proceedings of the Sixth USENIX Unix and Network Security Symposium, San Jose, July 1996.

[GR96] Greenberg S and Roseman M "Groupware Toolkits for Synchronous Work" Computer-Supported Cooperative Work, Trends in Software Series, Edited by Beaudouin-Lafon M, John Wiley & Sons Ltd, 1996.

[GS86] Greif I and Sarin S "Data Sharing in Groupwork" Proceedings of the first Conference on Computer Supported Cooperative Work (Austin, Texas), ACM New York, pp. 175 - 183, December 1986.

[HAY96] Hayton R "An Open Architecture for Secure Interworking Services" PhD Thesis, Fitzwilliam College, University of Cambridge, March 1996.

[HHLS94] Harvey A, Harvey A, Longshaw A and Sewell T "Business Law and Practice" Jordan Publishing Limited, Bristol, ISBN 0-85308-248-0, 1994.

[HIL97] Hilchenbach B "Observations on the Real-Life Implementation of Role-Based Access Control" Proceedings of the Twentieth National Information Systems Security Conference, Baltimore, Maryland, October 1997.

[HL97] Hagimont D and Louvegnies D "Javanaise: Distributed Shared Objects for Internet Cooperative Applications" Technical Report, INRIA Rhone-Alpes, 1997.

[HT95] Holbein R and Teufel S "A Context Authentication Service for Role Based Access Control in Distributed Systems - CARDS" J. H. P. Eloff, S.H. von Solms (eds.), Information Security - the Next Decade IFIP/SEC '95, Chapman & Hall, London, 1995.

[HUT] Hutchison A "Security in Group Applications: Lotus Notes as Case Study" Department of Computer Science, University of Zurich, Winterhurerstrasse 190, CH-8057, Zurich, Switzerland.

[INT95] Internet Engineering Task Force "Secure Socket Layer, Version 3.0" 1995.

[ISO88] ISO/CCITT "The Directory - Authentication Framework X.509" ISO 9594-8, Organisation for International Standardization, 1988.

[KA81] Kusner K and Anterpol J "Modern Banking Checklists: with Commentary" 3rd edition, Warren, Gorham and Lamont, 1981.

[KIN96] Kindberg T "Sharing Objects over the Internet: the Mushroom Approach" IEEE Global Internet 96, London, November 1996.

[KNT91] Kohl J, Neuman B and Ts'o T "The Evolution of the Kerberos Authentication Service" Massachusetts Institute of Technology, 1991.

[KT91] Kaashoek F and Tanenbaum A "Group Communication in the Amoeba Distributed Operating System" Proceedings of 11th International Conference on Distributed Computing Systems, pp. 222 - 230, 1991.

[LABW92] Lampson B, Abadi M, Burrows M and Wobber E "Authentication in Distributed Systems: Theory and Practice" ACM Transactions on Computer Systems, vol. 10, pp. 265 - 310, November 1992.

[LAM74] Lampson B "Protection" ACM Operating Systems Review 8, vol. 1, pp. 18 - 24, January 1974.

[LCJS87] Liskov B, Curtis D, Johnson P and Scheifler R "Implementation of Argus" Proceedings of the Eleventh ACM Symposium on Operating Systems Principals, Austin, ACM Press, pp. 111 - 122, 1987.

[LG95] Lewis M and Grimshaw A "The Core Legion Model" Department of Computer Science, University of Virginia, August 1995.

[LLSG92] Ladin R, Liskov B, Shrira L and Ghemawat S "Providing Availability using Lazy Replication" ACM Transactions on Computer Systems, vol. 10, no. 4, pp. 360 - 391, 1992.

[LOT] Lotus Notes Corporation "Lotus Notes: An Overview" On-Line Documentation, http://www.lotus.com/, Lotus.

[LSP82] Lamport L, Shostak R and Pease M "The Byzantine Generals Problem" ACM Transactions on Programming Languages and Systems, vol. 4, no. 3, pp. 382 - 401, July 1982.

[LYRFET98] Lampson B, Ylonen T, Rivest R, Frantz W, Ellison C, Thomas B "Simple Public Key Certificate" Internet Draft, Internet Engineering Task Force, http://www.ietf.org/ids.by.wg/spki.html, March 1998.

[MAF94] Maffeis S "A Flexible System Design to Support Object-Groups and Object Oriented Distributed Programming" Department of Computer Science, University of Zurich, Switzerland, 1994.

[MAR84] Marzullo K "Maintaining the Time in a Distributed System" Technical Report OSD-T8401, Xerox Corporation, 1984.

[MEA] Meadows C "The Need for a Failure Model for Security" Code 5543, Center for High Assurance Computer Systems, Naval Research Laboratory, Washington, DC 20375.

[MHP98] McDaniel P, Honeyman P and Prkash A "Lightweight Secure Group Communication" Technical Report 98-2, Center for Information Technology Integration, University of Michigan, Ann Arbor, April 1998.

[MM94] Mason, McCall-Smith "Law and Medical Ethics" Fourth Edition, Butterworths, London, Dublin, Edinburgh, 1994

[MS91] Moffett J and Sloman M "Content-Dependent Access Control" ACM SIGOPS Operating Systems Review, vol. 25, no. 2, pp. 63 – 70, April 1991.

[NKCM90] Neuwirth C, Kaufer D, Chandhok R and Morris J "Issues in the Design of Computer Support for Co-Authoring and Commentating" Proceedings of the ACM Conference on Computer Supported Cooperative Work, Los Angeles CA, pp. 183 - 195, 1990.

[NM91] Nardi B and Miller J "Twinkling Lights and Nested Loops: Distributed Problem Solving and Spreadsheet Development"

Computer Supported Cooperative Work and Groupware, Edited by Saul Greenburg, Academic Press London, pp. 29 - 52, ISBN 0-12-299220-2, 1991.

[NS78] Needham R and Schroeder M "Using Encryption for Authentication in Large Networks of Computers" Communications of the ACM, vol. 21, no. 12, December 1978.

[POS81a] Postel J "Internet Protocol" Technical Report RFC 791, FTP from Internet Network Information Center, nic.ddn.mil /usr/pub/RFC, 1981.

[POS81b] Postel J "Transmission Control Protocol" Technical Report RFC 793, FTP from Internet Network Information Center, nic.ddn.mil /usr/pub/RFC, 1981.

[PSWL95] Parrington GD, Shrivastava SK, Wheater SM and Little MC, "The Design and Implementation of Arjuna" USENIX Computing Systems Journal, vol. 8, no. 3, 1995.

[RB92a] Reiter MK and Birman KP "How to Securely Replicate Services" Department of Computer Science, Cornell University, Ithaca, New York, June 1992.

[RB92b] Rodden T and Blair S "Distributed Systems Support for Computer Supported Cooperative Work" Computer Communications, vol. 15, no. 8, pp. 527 - 538, October 1992.

[RBFHK95] van Renesse R, Birman KP, Friedman R, Hayden M and Karr DA "A Framework for Protocol Composition in Horus" Proceedings of the Fourth ACM Symposium on Principles of Distributed Computing, Ottawa, Ontario, ACM SIGOPS - SIGACT, pp. 80 - 90, August 1995.

[RBG92] Reiter MK, Birman KP and Gong L "Integrating Security in a Group Oriented Distributed System" TR92-1269, Department of Computer Science, Cornell University, Ithaca, New York, February 1992.

[RD95] Rowley AJ and Dollimore J "Replicated Secure Shared Objects for Groupware Applications" TR 716, Department of Computer Science, Queen Mary and Westfield College, University of London, March 1995.

[RD97a] Rowley AJ and Dollimore, J "An Implementation of a Secure Group Communication for Groupware Applications" TR 732, Department of Computer Science, Queen Mary & Westfield College, University of London, January 1997.

[RD97b] Rowley AJ and Dollimore J "Secure Group Communication for Groupware Applications" Proceedings of the 2nd European Research Seminar on Advances in Distributed Systems, Zinal, Switzerland, March 1997.

[RD97c] Rowley AJ and Dollimore J "A Framework for More Effectively Specifying Security Policy for Groupware Applications" TR 743, Department of Computer Science, Queen Mary & Westfield College, University of London, October 1997.

[RD97d] Rowley AJ and Dollimore J "A Framework for More Effectively Specifying Security Policy for Groupware Applications" MedNet97 Abstract Book, School of Cognitive and Computing Sciences, University of Sussex, November 1997.

[REI93] Reiter MK "A Security Architecture for Fault-Tolerant Systems" PhD Thesis, TR93-1367, Department of Computer Science, Cornell University, Ithaca, New York, August 1993.

[REI94a] Reiter MK "A Secure Group Membership Protocol" AT&T Bell Laboratories, Holmdel, New Jersey, 1994.

[REI94b] Reiter MK "Secure Agreement Protocols: Reliable and Atomic Group Multicast in Rampart" AT&T Bell Laboratories, Holmdel, New Jersey, 1994.

[REI94c] Reiter MK "The Rampart Toolkit for Building High-Integrity Services" Theory and Practice in Distributed Systems, International Workshop, Dagstuhl Castle, Germany, Springer-Verlag, September 1994.

[REI96] Reiter MK "Distributing Trust with Rampart" Communications of the ACM, vol. 39, No. 4, April 1996.

[RHB94] van Renesse R, Hickey TM and Birman KP "Design and Performance of Horus: A Lightweight Group Communications System" Department of Computer Science, Cornell University, 1994.

[RG97] Roseman M. and Greenberg S "Building Groupware with GroupKit" Tcl/Tk Tools, Edited by Harrison M, O'Reilly Press, p535 - 564, 1997.

[ROW98] Rowley AJ "Dynamic Access Control for Shared Objects in Groupware Applications" Proceedings of ECOOP Workshop on Distributed Object Security, Brussels, Belgium, INRIA, July 1998.

[RSA78] Rivest RL, Shamir A and 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.

[RSC92] Richardson J, Schwarz P and Cabrera L-F "CACL: Efficient Fine-Grained Protection for Objects" ACM OOPSLA, pp. 263 - 275, 1992.

[SAN96] Sandhu R "Access Control: The Neglected Frontier" Information Security and Privacy, Lecture Notes in Computer Science, no. 1172, pp. 219 - 227, Springer Verlag, 1996.

[SCH90] Schneider FB "Implementing Fault-Tolerant Services Using the State-Machine: A Tutorial" ACM Computing Surveys, vol. 22, no. 4, pp. 300 - 319, 1990.

[SD92] Shen H and Dewan P "Access Control for Collaborative Environments" ACM Proceedings CSCW 92, pp. 51-58, 1992.

[SDP91] Shrivastava SK, Dixon GN and Parrington GD "An Overview of the Arjuna Distributed Programming System" IEEE Software, January 1991.

[SKR97] Shapiro, Kloosterman, Riccardi "PerDiS - a Persistent Distributed Store for Cooperative Applications" Proceedings of the 3rd Cabernet Plenary Workshop, IRISA Rennes, France, April 1997.

[SNS88] Steiner J, Neuman C and Schiller J "Kerberos: An Authentication Service for Open Network Systems" Proceedings of Usenix Winter Conference, Berkeley, 1988.

[SUN90] Sun Microsystems Inc. "Network Programming" Sun Microsystems, Mountain View, CA, March 1990.

[SUN96] Sun Microsystems "JDK 1.1 Documentation" Sun Microsystems, URL: http://www.javasoft.com/products/jdk/1.1/docs/

[TFB91] Tatar D, Foster G and Bobrow D "Design for Conversation: Lessons from Cognoter" Computer Supported Cooperative Work and Groupware, Edited by Saul Greenburg, Academic Press London, pp. 55 - 79, ISBN 0-12-299220-2, 1991.

[TH86] Turn R and Habibi J "Interactions of Security and Fault-Tolerance" Proceedings of the ninth NBS/NCSC National Computer Security Conference, pp. 138 - 142, September 1986.

[TIN90] Ting TC "Application Information Security Semantics" Database Security III, Status and Prospects, Elsevier Science Publishers BV, North Holland, IFIP, 1990.

[TS94] Thomas R and Sandhu R "Conceptual Foundations for a Model of Task-Based Authorizations" IEEE Computer Security Foundations Workshop 7, pp. 66 - 79, Franconia, June 1994.

[VDN91] Valacich J, Dennis A and Nunamaker J "Electronic Meeting Support: the GroupSystems Concept" Computer-Supported Cooperative Work and Groupware, Academic Press London, pp. 133 - 154, ISBN 0-12-299220-2, 1991.

[WABL94] Wobber E, Abadi M, Burrows M, and Lampson B "Authentication in the TAOS Operating System" ACM Transactions on Computer Systems, vol. 12, no. 1 February 1994.