Integrating Security in a Group Oriented Distributed System*

Michael Reiter
Kenneth Birman
Li Gong**

TR 91-1239
October 1991

Department of Computer Science
Cornell University
Ithaca, NY 14853-7501

* This work was supported by the Defense Advanced Research Projects Agency (DoD) under DARPA/NASA subcontract NAG2-593 administered by the NASA Ames Research Center, by grants from GTE, IBM, and Siemens, Inc., and by a National Science Foundation Graduate Fellowship. Any opinions, conclusions or recommendations expressed in this document are those of the authors and do not necessarily reflect the views, policies or decisions of the National Science Foundation or the Department of Defense.

**Visiting at Cornell University. Usual address: ORA Corporation, 675 Massachusetts Avenue, Cambridge, Massachusetts 02139.

vh

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY *(Leave blank)*

2. REPORT DATE
October 1991

3. REPORT TYPE AND DATES COVERED
Special Technical

4. TITLE AND SUBTITLE
Integrating Security in a Group Oriented Distributed System.

5. FUNDING NUMBERS
NAG 2-593

6. AUTHOR(S)
Michael Reiter, Kenneth Birman, Li Gong

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Kenneth Birman, Associate Professor
Dept. of Computer Science
Cornell University

8. PERFORMING ORGANIZATION REPORT NUMBER
91-1239

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
DARPA/ISTO

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION/AVAILABILITY STATEMENT
[illegible]D FOR PUBLIC RELEASE
[illegible]UTION UNLIMITED

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

Please see page 1 of this report.

14. SUBJECT TERMS

15. NUMBER OF PAGES
26

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT
UNCLASSIFIED

18. SECURITY CLASSIFICATION OF THIS PAGE
UNCLASSIFIED

19. SECURITY CLASSIFICATION OF ABSTRACT
UNCLASSIFIED

20. LIMITATION OF ABSTRACT
UNLIMITED

Integrating Security in a Group Oriented Distributed System[*] (Preliminary Version)

Michael Reiter
reiter@cs.cornell.edu

Kenneth Birman
ken@cs.cornell.edu

Li Gong[†]
gong@cs.cornell.edu

Department of Computer Science
Cornell University
Ithaca, New York 14853

October 8, 1991

Abstract

A distributed security architecture is proposed for incorporation into group oriented distributed systems and, in particular, into the Isis distributed programming toolkit. The primary goal of the architecture is to preserve the Isis abstractions in hostile environments. These abstractions include process groups and causal and atomic group multicast. Moreover, a delegation and access control scheme is proposed for use in group oriented systems. The focus of the paper is the security architecture; particular security protocols and cryptosystems are not emphasized.

1 Introduction

Systems that address security issues in distributed environments have traditionally been constructed upon the remote procedure call (RPC) paradigm of communication (e.g., [Bir85,SNS88,Sat89,

[*]This work was supported by the Defense Advanced Research Projects Agency (DoD) under DARPA/NASA subcontract NAG2-593 administered by the NASA Ames Research Center, by grants from GTE, IBM, and Siemens, Inc., and by a National Science Foundation Graduate Fellowship. Any opinions, conclusions or recommendations expressed in this document are those of the authors and do not necessarily reflect the views, policies or decisions of the National Science Foundation or the Department of Defense.

[†]Visiting at Cornell University. Usual address: ORA Corporation, 675 Massachusetts Avenue, Cambridge, Massachusetts 02139.

TY91]). Many systems, however, utilize more general types of communication which have not enjoyed the same amount of attention from the security community. One such alternative is *group oriented* communication, based on the *process group* abstraction [BCG91]. Process groups have been incorporated into many distributed systems [OSS80,CZ85,BJ87,PBS89,LLS90,KT91] and have been shown to facilitate the implementation of complex distributed algorithms and fault tolerant applications [CZ85,BJ87,LLS90]. The benefit in preserving the process group abstraction in hostile environments could therefore be great. In particular, it would facilitate the construction of *securely fault tolerant* applications, i.e., fault tolerant applications which remain correct even when under malicious attack.

To illustrate, consider a stock brokerage that wishes to establish an online trading service at the New York Stock Exchange (NYSE). Since the majority of the firm's trading at the NYSE will be executed through this trading service, its availability and performance are crucial, and thus it must be replicated. The firm's programmers therefore choose to implement the service as a fault tolerant process group in their favorite group oriented programming environment. While the firm can protect its own sites at the exchange from corruption, the firm's programmers cannot trust that other sites, or the network by which the sites communicate, will behave as expected. Nevertheless, interaction with other sites is necessary for efficient trading, and thus the group is forced to execute in a potentially hostile environment. In particular, intruders (e.g., corrupt, competing traders) may attempt to infiltrate the group, alter or forge group communication, or undermine the group by tampering with the group abstractions on which the consistency and correctness of the service relies. If the group oriented programming environment does not defend at least its own abstractions, then the firm's programmers are faced with an unattractive choice: either they reimplement the group oriented programming environment to include defenses against these forms of attack, or they dismiss the process group approach and resort to other, possibly less favorable ones.

This paper presents a distributed security architecture to be integrated with group oriented systems and, in particular, with the Isis toolkit [BJ87].[1] Isis is a toolkit for distributed programming which provides process group and reliable group multicast abstractions. With respect to Isis, the aims of this work are threefold. The first is to weaken the execution model assumed by Isis so that malicious behaviors are admitted, while still preserving the abstractions provided by Isis. This change will enable programmers to rely upon the Isis abstractions even in hostile environments and thus will facilitate the construction of fault tolerant services which remain correct even when under malicious attack. The second is to enhance the Isis abstractions to be more suitable for use in a

[1]More specifically, this security architecture is tailored to a reimplementation of the Isis toolkit called Horus, named after the son of Isis in Egyptian mythology. In this paper, we will use Horus terminology, which may be unfamiliar to users of earlier versions of Isis.

hostile environment. The third is to accomplish the first two without unreasonably degrading the performance of the toolkit.

The goal of this paper is to describe the major features of our security architecture. In particular, we do not discuss specific cryptosystems or security protocols in detail. We instead focus on the mechanisms we use to protect the abstractions which are fundamental in Isis and, we believe, in a group oriented setting.

The rest of this paper is structured as follows. We begin in section 2 with a more detailed but informal description of the abstractions provided by Isis. Then, in section 3 we discuss the system model for which Isis is designed and the model we consider by weakening it to allow malicious behaviors. At this point we will be in a position to clarify our goals and to enumerate what is needed to achieve them. Section 4 addresses this and proposes an architecture to achieve these requirements. Finally, section 5 describes a delegation and access control scheme for use in group oriented systems. We end with a discussion of future directions of research.

2 The Isis Abstractions

The abstractions provided by Isis can be separated into two types, namely the *process group* and *virtual synchrony* abstractions. A process group is simply a collection of processes with an associated *group address*. Usually a process group is created for cooperation in a distributed task such as replicating data, processing data in parallel, or providing fault tolerance, although Isis enforces no restrictions on the purposes for which groups are formed. Groups may overlap arbitrarily, and processes may create and join groups at any time. Moreover, a process may leave a group, either by requesting to do so or by failing (i.e., crashing). A group G can thus be seen as progressing through a sequence $view_0(G), view_1(G), \ldots$ of *views*, where

1. $view_i(G) \subseteq P$, where P is the set of all processes in the system.
2. $view_0(G) = \emptyset$, and
3. $view_i(G)$ and $view_{i+1}(G)$ differ by the addition or subtraction of exactly one process.

Members of a group learn about the membership of the group through certain events. More precisely, execution of a process $p \in P$ is modeled as a sequence $e_0^p, e_1^p, \ldots$ of *events*, each corresponding to the execution of an indivisible action. One such possible event is the delivery of the i'th group view $view_i(G)$ of a process group G, denoted by $\mathbf{view}_p(i, G)$. Views are delivered to processes in sequential order, although p observes $\mathbf{view}_p(i+1, G)$ if and only if $p \in view_i(G) \cup view_{i+1}(G)$; i.e., a process observes only those subsequences of $view_0(G), view_1(G), \ldots$ which begin when it becomes a member and end when it is removed. If $p \in view_i(G)$, then when $\mathbf{view}_p(i, G)$ is observed at p, we say that p is *in the i'th group view of G* until the event $\mathbf{view}_p(i+1, G)$.

The primary means of communication in Isis is *group multicast*. A process in a group can multicast to the group by specifying the group address as the destination.[2] The abstraction of virtual synchrony consists of certain delivery guarantees regarding group multicasts. First, all destination processes of the message are in the same group view when the message is delivered, and the set of destination processes is precisely the members of that view. Second, all operational destinations eventually deliver the message, or, and only if the sender fails, none do. Third, when multiple destinations receive the same messages, they observe consistent delivery orders, in one of the following two senses.

The first and least restrictive delivery ordering of interest is the *causal* delivery ordering, based upon the potential causality relation defined in [Lam78]. To define this ordering, we introduce two more types of events which can be executed or observed by a process. If p is in some view of G, then e_i^p might be the event which multicasts a message m to G, denoted $\mathbf{mcast}_p(m, G)$, or which delivers to p a message m multicast to G, denoted $\mathbf{deliver}_p(m, G)$. The potential causality relation "$\rightarrow$" is defined as the irreflexive, transitive closure of the smallest relation "$\leadsto$" satisfying

1. for all i and p, $e_i^p \leadsto e_{i+1}^p$, and
2. for all m, p, q, and G, $\mathbf{mcast}_p(m, G) \leadsto \mathbf{deliver}_q(m, G)$.

Isis' causal delivery ordering property guarantees that if $\mathbf{mcast}_p(m, G) \rightarrow \mathbf{mcast}_q(m', G')$, then at any common destination r, $\mathbf{deliver}_r(m, G) \rightarrow \mathbf{deliver}_r(m', G')$. In words, if the multicast of message m causally precedes the multicast of message m', then m is delivered before m' at any common destination. The multicast protocol which implements this property is called CBCAST.

This delivery ordering can be extended to a *total* ordering in the following sense, which expresses the second and more restrictive delivery ordering provided by Isis: two messages sent concurrently (i.e., that are not related causally) to the same group are delivered to all members of the group in the same order. In terms of "$\rightarrow$", this property is specified by additionally requiring that if at some

[2]Actually, nonmembers can also multicast to a group in Isis, although in this paper we do not consider this case.

p, $\mathbf{deliver}_p(m, G) \rightarrow \mathbf{deliver}_p(m', G)$, then at all other common destinations q, $\mathbf{deliver}_q(m, G) \rightarrow \mathbf{deliver}_q(m', G)$.[3] This property is implemented by the ABCAST protocol.

3 The System Model

The basic system model for which Isis is implemented is a very benign one. Informally, the system consists of a set S of sites which execute the set P of processes.[4] Sites and processes may fail, but only by crashing detectably, and if a site fails then so do the processes residing upon it. The sites, and the processes they host, communicate via an asynchronous network: no bounds on message transmission delays are assumed.

The system model we consider in this paper is obtained by weakening aspects of this model in various ways, namely by allowing the network or sites to be corrupted by an intruder. In the terminology of [VK83], a corrupt network may suffer certain passive attacks, namely the *release of message contents*, and certain active attacks, namely *message-stream modification*, *spurious association initiation*, and *denial of message service*, at the hands of an intruder. Release of message contents occurs when an intruder simply observes intelligible messages passing over the network without interfering with their flow. Message-stream modification includes transient attacks on the authenticity, integrity and ordering of messages. Spurious association initiation includes attacks in which an intruder replays a previously recorded association initiation sequence or attempts to establish associations under a false identity. Lastly, denial of message service attacks are essentially persistent message-stream modification attacks and often include discarding or delaying all messages between two communicating endpoints. Note, however, that we do not consider one other type of attack enumerated in [VK83], namely *traffic analysis*. In a traffic analysis attack an intruder gathers information about the contents of unintelligible messages from the frequency of transmissions and the lengths, sources and destinations of the messages. In this paper we make no effort to deal with traffic analysis attacks.

In addition to network corruption, there exists a set $C \subseteq S$ of corrupt sites. A corrupt site may exhibit arbitrarily malicious behaviors, limited only by the aforementioned network assumptions

[3]Several other reasonable definitions are possible for a total delivery order, although current plans for Horus include implementation of this one only. Another option which may be considered in the future is if at some p, $\mathbf{deliver}_p(m, G) \rightarrow \mathbf{deliver}_p(m', G')$, then at all other common destinations q, $\mathbf{deliver}_q(m, G) \rightarrow \mathbf{deliver}_q(m', G')$. However, we will not concern ourselves with this in this paper.

[4]Unless otherwise stated, throughout this paper the term "process" refers to an application process, and the term "site" refers to a workstation running an operating system and, once added, Isis.

and the assumption that it is computationally infeasible for an intruder to break the cryptosystems we employ. Intuitively, a corrupt site is one on which the operating system or Isis code or data has been either accidentally or maliciously altered, disrupted or replaced.

In this work we make two assumptions about the operating system running at each site, if not corrupt. First, we assume that it authenticates in a secure fashion the user identifiers of the processes it executes.[5] Second, we assume that it provides protected, private address spaces for, and private, authentic message passing between, both system and user processes local to the site. This includes the protection of virtual address spaces stored on external media.[6]

4 Protecting the Isis Abstractions

Given this statement of the system model, we are now in a position to specify our goals more precisely. Intuitively, given a process group, we would like to preserve the abstractions guaranteed to the members of the group by Isis, as described in section 2. That is, we would like to guarantee that a process in a group observes a correct sequence of events. This is clearly impossible, however, if a site hosting a group member is corrupt, because that site can cause arbitrary events to be observed in any order by any process it hosts! We thus restrict our efforts to process groups which are hosted by sites not in C. That is, let $sites_i(G)$ be the set of sites hosting the members of $view_i(G)$, and let an *uncorrupt* group G be one such that $C \cap (\bigcup_i sites_i(G)) = \emptyset$. Then, our goal is to modify Isis to guarantee that in any uncorrupt group G the Isis abstractions are observed by processes in $\bigcup_i view_i(G)$ with respect to group G. Accordingly, for the remainder of the paper, when we speak about protecting a certain abstraction with respect to a process group, we assume that the group is uncorrupt.[7] In section 5 we justify this assumption by addressing the question of how access to groups can be controlled.

We now consider what must be done to achieve this goal. Of course, in a single paper we could not hope to detail every step which is required to achieve this in a toolkit as complex as Isis. Thus, we

[5]This is a rather strong requirement, but the mechanisms described in this paper facilitate its implementation. For example, if smart-card technology is available, each user and site can be treated as an Isis group and the delegation mechanisms of section 5 can be used to authenticate the user identifiers of processes executed from remote sites, in a fashion similar to that of DSSA [GM90]. Even without such technology, the authentication mechanisms of section 4.1 provide a secure communication channel between any two sites which can be used to facilitate the authentication of user identifiers.

[6]If the operating system pages over the network, this requires the use of a pager which encrypts as it pages.

[7]We also assume that the Isis *failure detector* and *name service* are not corrupt and reside on uncorrupt sites. These services will be mentioned in sections 4.1 and 4.2, respectively.

limit our discussion to what we see as the three major obstacles, which we outline below and later discuss in detail.

First, due to the very fact that preservation of the abstractions requires communication, a necessary step is to develop a subsystem which provides message authentication. In particular, this subsystem should allow a site in a group (i.e., a site hosting a member of a group) to detect attempts by an intruder to insert, alter or replay group messages or to impersonate another site in the group. If we can achieve this, a site in the group can rely upon the legitimacy and authenticity of messages apparently from other sites in the group. In section 4.1 we propose an authentication subsystem which accomplishes these goals.

Once we have it, an authentication subsystem of this strength yields additional benefits. Since altered messages are detected (and ignored), denial of message service becomes indistinguishable from lengthy message delivery times. And, since Isis is constructed for an asynchronous environment, Isis will behave correctly under such attacks. That is, while a network intruder can prohibit the liveness of Isis and cause sites to be mistakenly deemed faulty by denying message service, these attacks will not result in the violation of any safety properties guaranteed by Isis to an uncorrupt group. Similarly, attempts by an intruder to reorder messages on the network are fruitless, as Isis assumes that the network can do this anyway.

The authentication subsystem therefore nullifies the active network attacks described in section 3, as well as any attempts to impersonate sites in an uncorrupt group. Moreover, the authentication subsystem severely limits other types of attacks an intruder can mount. For example, the majority of the process group abstraction and the first two aspects of virtual synchrony listed in section 2 are easily preserved, because their implementations require communication local to the group, which by assumption contains only uncorrupt sites.

The second obstacle, and the remaining weakness in the process group abstraction, lies in the protocol by which a process joins a group. In a request to join a group the process specifies the group address, but unless the process' site can authenticate the group specified by the group address, then the first group view the process observes, and hence all subsequent group views, may be fallacious. A related issue which must be addressed is how a process can obtain an authentic group address for a group it wishes to join. In section 4.2 we address these issues.

Third, the intruder can greatly complicate the definition and preservation of causality in our system model. In addition to Isis, numerous other systems have implemented protocols which guarantee causal delivery orderings among messages (e.g., [PBS89,LLS90]), and thus interest in preserving causality in hostile environments is not solely among Isis users. Moreover, CBCAST (i.e., causal multicast) is central to virtual synchrony in Isis also because ABCAST, the protocol which implements the total ordering property, is implemented in terms of it [BSS90]. Section 4.3 is devoted to the problems of understanding and preserving causality in our system model.

4.1 Authentication

We introduce authentication mechanisms at the lowest layer of the Isis toolkit, namely the Multicast Transport Service (MUTS) [vR91]. A copy of MUTS resides on each site, logically at the transport layer of the ISO OSI Reference Model, and provides to the layers above it at-most-once, sequenced communication to other sites. MUTS has the job of providing these abstractions while insulating the higher layers from the particular transport protocol used, which may exploit hardware multicast capability.

For our purposes, the MUTS layer is the obvious place at which to authenticate messages. Indeed, it would be fruitless to authenticate messages only at higher layers of the Isis system, as then they could not rely upon the abstractions provided by MUTS. And, other systems (e.g., DASH [AR87]) have identified additional advantages in authenticating between sites, as opposed to at higher levels.[8]

Before presenting our authentication mechanisms, we must briefly consider how MUTS works. The primary structure recognized by MUTS is the *group entity*. The group entity corresponding to the process group G is the collection of sites hosting members of the group, and accordingly it progresses through the sequence $sites_0(G), sites_1(G), \ldots$ of sets. A MUTS layer learns about changes to the membership of a group entity from the layer above it, which communicates with other sites in the group entity and with the Isis *failure detector* [BJ87,RB91] to make this determination. Each MUTS layer thus has a current *member list* of each group entity it is in.[9] When MUTS receives a message from a higher layer to be multicast to a group entity, it opens a *connection* to the members of its current member list for the group entity, if one does not already exist. A connection is associated with exactly one group entity and is simply a logical end-to-end data path from the originating site to the other sites in the originator's member list. If a site is removed from the originator's member list, it is also removed from the connection, but if a site is added to the

[8]Moreover, by our assumptions of section 3 regarding the operating system and by the fact that all Isis communication flows through MUTS, authentication between MUTS layers also prevents any impersonation whatsoever of Isis by a malicious user process residing on an uncorrupt site.

[9]Member lists should not be confused with group views delivered to the application process. The latter constitute the process group abstraction and are synchronized with communication as described in section 2. The former, however, consist of sites and are not coordinated with incoming communication or other events.

originator's member list, the old connection is disassembled and a new connection is negotiated for the new member list. To send a message on a connection, MUTS breaks the message into packets. and hands these packets to the transport protocol for transmission. Each packet carries with it the connection number and a sequence number for the connection. Connection numbers are unique system-wide, and the sequence numbers for a connection form an increasing sequence. When the sequence reaches its upper bound, the connection is disassembled and a new one is negotiated. Packet acknowledgements are managed by MUTS with a sliding-window protocol which has been adapted for use with multicast communication.

Techniques for authenticating messages (or in this case, packets) have existed in the literature for many years. Traditionally, these methods have depended upon encryption, but methods based upon *pseudo-random functions* are theoretically at least as attractive. Informally, a pseudo-random function f has the property that if f is unknown, it is computationally infeasible to produce $f(m)$ for any m with a probability of success greater than random guessing, even after having seen several other $\langle m', f(m')\rangle$ pairs. Thus, given a family of pseudo-random functions $\{f_K\}_{K\in\mathcal{K}}$, indexed by keys from some *key space* $\mathcal{K}$, two parties which share a secret key K can authenticate their messages to each other by appending $f_K(m)$ to each message m [Riv90a]. Moreover, they can be sure of the freshness of their messages if timestamps, nonce identifiers, or sequence numbers are included therein.

In Isis, we will employ an efficient approximation of a pseudo-random *hash* function; two candidates are $f_K(M) = g(K, M)$ and $f_K(M) = E_K(g(M))$, where g is a sufficiently strong one-way hash function (e.g., [Riv90b]) and E_K is an encryption function (e.g., [DES77]) with key K. Given such an approximation, authentication methods based upon pseudo-random functions are generally more efficient than those based upon encryption. However, encryption is more useful for defending against the release of message contents, and for some applications this is desirable. We will thus offer both alternatives — when sending a message, an application process can request that it be encrypted or that it be authenticated via a pseudo-random function. For the rest of this section, we discuss only the latter option; methods using encryption are similar.

For MUTS we generalize the ideas presented above to take advantage of hardware multicast capabilities that may be exploited by the transport protocol. Instead of establishing a shared key for every pair of MUTS layers, we establish a shared key per connection, called a *connection key*. The connection key is a secret held by the sites involved in the connection and is used to authenticate messages sent on the connection. When a connection is created, the site initiating the connection generates a fresh connection key K and distributes it to the sites on its member list. Then, the multicast "$P, f_K(P)$" of packet P on the connection can be verified at all destinations (and a packet encrypted under K can be deciphered only at sites involved in the connection). Moreover, provided that the connection is fresh, each destination can verify that the packet is fresh, because P contains the sequence number for the connection. Here we do not detail the protocol by which a connection is opened, although we remark that freshness of connections is guaranteed by incorporating timestamps[10] into the appropriate connection initiation messages.

In order to authenticate and distribute connection keys, we employ a *group key*. This is actually a private key/public key pair, possession of the private component of which is evidence of membership in the group entity. The group key for a group is created by the site hosting the first member of the group, and as other processes join, the group key is given to their sites. Connection keys for a group are thus communicated in the obvious way, encrypted with the public key of the group and signed with the private key of the group.

It must be emphasized at this point that *sites* hold connection keys and private keys of groups: user processes do not. Thus, when a process leaves a group voluntarily, the site on which it resides can destroy the group and connection keys which it held on behalf of the process. By doing so, if the site is subsequently corrupted, the intruder will not be able to masquerade as a group member.[11] Similarly, if a member process crashes, again the site will destroy the keys it held on the process' behalf. If the entire site crashes, we rely upon the loss of volatile storage to eliminate all keys from memory.

Of course, we must discuss how group keys are distributed. Like all other key distribution schemes, in order to distribute a group key we require some form of an *authentication service*, i.e., an *a priori* trusted authority. We choose to employ a public key authentication service due to the security advantages which can be achieved [Dif88]. Associated with the authentication service is a private key (known only to the service) and a corresponding public key. The public key is given to the MUTS layer on each site, along with the site's own *site key* (a private key/public key pair), when the site is booted.[12] Once the site is booted, it requests from the authentication service its *certificate*, which contains the identifier and public key of the site and the expiration

[10]See appendix A.

[11]Formally our assumptions exclude the subsequent corruption of the site of a former group member, although in practice this erasure of keys is prudent. Our assumptions also omit the case in which a site, due to corruption, does not destroy the group or connection keys when its process leaves the group.

[12]The boot procedure appropriate for each site in a particular setting is dependent on many factors, such as the physical security of the site, whether the site is diskless, and the role of the site in the system. Thus, a complete discussion of this issue is outside the scope of this paper. However, the boot procedure used at each site should prevent an intruder from booting the site with false operating system or Isis code or with a false authentication service public key.

time of the public key, all signed by the private key of the authentication service. Each site's certificate is subsequently stored at, and disseminated from, the site itself. This method of storing certificates has the benefit of eliminating the need for a public key or certificate repository as is used in many security architectures (e.g., Strongbox [TY91]), and although we will not describe our key distribution protocols here, it also does not in general increase the message complexity of our protocols due to the particular patterns of communication seen in Isis. We emphasize that no interaction with the authentication service is required to distribute the private key of a group. Ideally the authentication service would interact with a site only when it needs to give the site a fresh certificate, and indeed the authentication service could be taken offline until such a need arises.

We can generalize this scheme by allowing multiple authentication services, as originally proposed in [BLNS86]. Intuitively, each authentication service would be responsible for generating certificates for some subset of the sites, and each site would be given, at boot time, the public key of the authority it should trust. This generality has consequences, however, in the sense that authentication among sites which trust different authentication services becomes complex. One solution is to allow several authentication services to generate certificates for the same site, and another is to employ "higher authorities," much like the *cross-certifying authorities* of SPX [TA91], to vouch for the public keys of other authentication services. The details of this have not yet been sufficiently investigated, however, and so we will not discuss them further here. In the first implementation of this system, we intend to use a single authentication service, and this generalization is regarded as an enhancement for later development.

4.2 Joining Groups

As described earlier, the protocol by which a process joins a group is crucial to the process group abstraction, because if this is not secure, an intruder may cause the process to observe fallacious group views and thus to act incorrectly. In the current plans for Isis, the protocol for a process to join a group runs as follows. First, the requesting process specifies the group address of the group it wishes to join. This address contains the address of a *group contact*, which is a distinguished site in the group. The process' site sends the join request to the group contact, which then formally admits the process to the group and gives the process' site its first group entity member list.[13]

In order for the join protocol to be secure, the process' site must be able to authenticate the response

[13]Moreover, if the requesting site included its certificate in the request, then the group contact could also return the private key of the group encrypted under the site's public key.

from the group contact. It would appear that this is done easily via the methods of section 4.1: the group contact signs its response with its site key and appends its certificate, thus allowing the receiving site to verify it. A difficulty arises, however, if the group address is outdated in the sense that the group contact contained therein has since left the group. Even though in theory our model prohibits the former group contact from being corrupted, in practice it would be prudent for the requesting site to verify that the supposed group contact is genuinely in the group before accepting any group information from it. To facilitate this, we include the public key of the group in the group address, and thus when the requesting process specifies the group address, its site can verify that the supposed group contact is actually in the group.

Of course, the success of this scheme hinges on the ability of a process to obtain the authentic address of a group it wishes to join; for the remainder of this section we address this issue. In Isis, a process can obtain a group address in either of two ways: it can simply receive it from another application process, or if the group is registered at the Isis *name service*, then the process can request the group address from the name service by specifying the *group name*. The name service is a fault tolerant Isis service which implements a hierarchical name space, like that of a file system except with groups at the leaves instead of files. A group name is a path from the root to a leaf in that hierarchical name space. A group member can register the group address under some name at the name service anytime after the group is created. A group which has not been registered with the name service is an *anonymous group*, and the address of an anonymous group can be obtained only from another application process (or by creating the group).

If a process receives an address from another application process, it can trust that address only as much as it trusts the other process. This is not to say that this scheme is worthless for secure operations. On the contrary, if the sending process is a member of either a "trustworthy" process group or a group which was *delegated* by such a group (see section 5), and proves it by having its site exhibit knowledge of the private key of the appropriate group and by including the appropriate credentials, then the address may be perfectly acceptable. But, to verify the claims of the sending process, the receiving process must obtain the group addresses (i.e., the public keys) of the delegating groups and the group of which the sending process claims to be a member. So, in many cases verification of group addresses received from other processes eventually requires that the verifying process be able to obtain legitimate group addresses from the name service.

Accordingly, we now consider how authentic group addresses can be obtained through the name service. The authentication mechanisms of section 4.1 can easily be adapted to allow a site to authenticate the name service, say by having the name service sign group addresses with its private key and having the authentication service produce a certificate for the name service. So, authenticating information *received from* the name service is not a problem.

The major impediment to the success of this scheme is the inability of the name service to authenticate information *sent to* it by a process attempting to register a group. Intuitively, this is because the name service has no reason to believe one process over another regarding the correct address of a group, as it keeps no record of group membership. We solve this problem by allowing processes to impose access controls upon the directories of the hierarchical name space, thus providing the name service some means by which to discriminate between valid and invalid information. When a process creates a directory of the name space, it specifies access control policy for the directory that restricts which processes, and in particular the sites from which these processes, can register a group or create a directory in that directory; in section 5, we describe a method by which this access control policy can be specified and enforced. The name service will then allow only an authorized process (residing on an authorized site) to register a group in the directory. Provided that a directory allows registrations only from nonmalicious processes on uncorrupt sites, the name service can verify the authenticity of a group address being registered in that directory simply by authenticating the registering process' site via the methods of section 4.1.

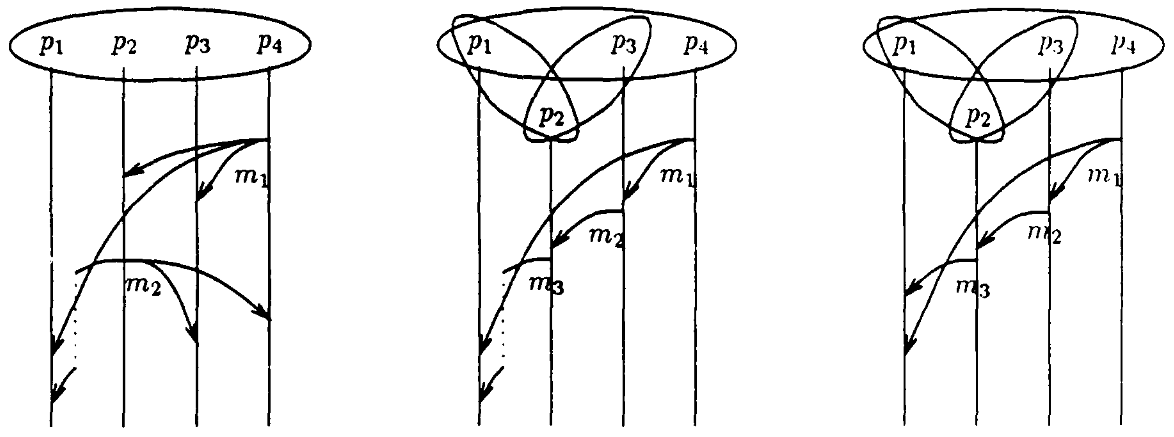
4.3 Causal Multicast

As previously described, the CBCAST protocol implements the causal delivery ordering property of virtual synchrony. It is implemented above the MUTS layer described in section 4.1, and thus its messages can be authenticated by the methods described there. Even having limited the intruder's ability to alter and forge messages, though, there are still difficulties in determining what causality means, and precisely what abstraction we should try to protect, in our system model. In this section we address these issues, but first we illustrate the role of CBCAST in the basic Isis model.

Consider first an instance of *single group causality*, illustrated in part (a) of figure 1. This shows a single process group with four members p_1, p_2, p_3 and p_4, residing respectively on sites s_1, s_2, s_3 and s_4. Time increases down the vertical lines, and an arrow ending at the vertical line below a process indicates the *delivery* of the multicast represented by the arrow to the process.[14] In this scenario, p_4 multicasts m_1 to the group, and after s_2 delivers it to p_2, p_2 multicasts m_2 to the group. Causality requires that m_2 be delivered to p_1 after m_1, as indicated by the delay of message m_2 until after m_1 in part (a) of figure 1.

[14]In Isis, we distinguish between the *receipt* of a message at a site and the *delivery* of a message to the application process.

Figure 1: Causal Multicast



(a) Single group causality (b) Multiple group causality (c) Causality violated

The more complex flavor of causality is called ***multiple group causality***, illustrated in part (b) of figure 1. In this situation the four processes are organized into three overlapping process groups $G_1 = \{p_1, p_3, p_4\}$, $G_2 = \{p_2, p_3\}$ and $G_3 = \{p_1, p_2\}$. Here p_4 multicasts message m_1 to group G_1. After m_1 is delivered to p_3, p_3 multicasts m_2 to G_2, and upon delivery of m_2 to p_2, p_2 multicasts m_3 to G_3. Multiple group causality requires that m_3 be delivered to p_1 after m_1, as indicated in the figure.

In our system model, the definition and preservation of causality is more complex. First, complications arise from the fact that (processes on) corrupt sites can exhibit arbitrary communication behavior, and not simply the group multicasts by which causality is defined in Isis. In fact, this holds for any site in a corrupt group, because the intruder can forge group communication for those sites. Second, even with a reasonable definition of causality that incorporates the behavior of corrupt sites, it is not clear how we could (or if we should) respect causal obligations originating in corrupt groups, again because communication in corrupt groups, and thus the perceived order in which events occurred in a corrupt group, may be fallacious. Third, while communication still occurs only through message passing, the intruder is an observer of all messages and can respond from a corrupt site based upon the information in them. For example, in part (b) of figure 1, the intruder could observe m_1 on the network and, if $s_2 \in C$, could immediately send m_3, based upon information in m_1 and without waiting for m_2; now is there a causal relationship between m_1 and m_3? Of course, we can avoid this issue by encrypting all messages, but this is a costly step to take before even defining the notion of causality in our system model.

For the sake of brevity, we address these issues elsewhere and in this paper provide the following guarantee: if $p_1, \ldots, p_n$ are (not necessarily distinct) processes residing on uncorrupt sites and there exists a *causal chain*

$$e^{p_1}_{i_1} \leadsto e^{p_2}_{i_2} \leadsto \ldots \leadsto e^{p_n}_{i_n} \tag{1}$$

such that

- $e^{p_1}_{i_1} = \mathbf{mcast}_{p_1}(m, G)$,
- $e^{p_n}_{i_n} = \mathbf{mcast}_{p_n}(m', G')$,
- G and G' are uncorrupt, and
- if $e^{p_j}_{i_j} = \mathbf{mcast}_{p_j}(\hat{m}, \hat{G})$ and $\hat{G}$ is corrupt, then $p_j = p_{j+1}$ (i.e., the causal chain is well-defined).

then $\mathbf{deliver}_q(m, G) \rightarrow \mathbf{deliver}_q(m', G')$ at all destinations q of m and m'. We argue that this is a reasonable guarantee to provide for several reasons. First, single group causality, which is necessary to provide the ABCAST property to a group, is the special case of this guarantee in which $p_1, \ldots, p_n$ are members of the same group $G = G'$ and if $p_j \neq p_{j+1}$, then $e^{p_j}_{i_j} = \mathbf{mcast}_{p_j}(\tilde{m}, G)$ and $e^{p_{j+1}}_{i_{j+1}} = \mathbf{deliver}_{p_{j+1}}(\tilde{m}, G)$ for some message $\tilde{m}$; i.e., single group causality is the case in which the causal chain never leaves the group G. Second, if a group must rely upon multiple group causality, this guarantee allows the group to protect itself by judiciously choosing the groups with which it shares members, because uncorrupt sites will observe incorrect orderings between multicasts only if the causal chain which links them traverses a corrupt group.[15] Third, a stronger guarantee would be useless to many applications, because an uncorrupt group may be contaminated at the application level by messages with a corrupt group in their causal history, regardless of what sort of causal guarantees are made regarding those messages.

To provide this guarantee, we first choose a secure protocol which provides single group causality in an uncorrupt group. By the work of section 4.1, any protocol that maintains all causality information for group communication local to the group will suffice: one such protocol is the *vector timestamp* protocol for a single group described in [BSS90], which we do not discuss here. Second,

[15]Of course, here we are using the term "causal chain" informally, because one which traverses a corrupt group may not be well-defined.

we extend this protocol to account for causal chains which exit and reenter the group. As in the single group case, any protocol which maintains the relevant causality information only in the groups encountered on the chain is sufficient, because we are concerned with the case in which the chain traverses only uncorrupt groups. However, for reasons which will be outlined below, we adopt a more cautious strategy.

Intuitively, we enforce multiple group causality between m and m' by ensuring before the causal chain (1) even leaves the group G that any causal obligations resulting from the multicast of m will be satisfied. One protocol which does this is the *conservative protocol* of [BSS90], described as follows. A multicast is ***stable*** if it has been received at all of its destination sites, and a group $\hat{G}$ is ***active*** for a process p if p's site does not know of the stability of a multicast to $\hat{G}$ either sent by or delivered to p. The ***conservative multicast rule*** states that a process p may multicast to group $\hat{G}$ if and only if $\hat{G}$ is the only active group for p or p has no active groups. If p attempts to multicast when this rule is not satisfied, the multicast is delayed, and during this delay no multicasts are delivered to p. So, in part (b) of figure 1, s_3 simply delays sending m_2 until it knows that m_1 has been received by s_1. Then, the delivery algorithm at s_1, which specifies that two multicasts in different groups are delivered in order of receipt, enforces the causal delivery property.

Because we are attempting to provide causal orderings defined by causal chains through only uncorrupt groups, more efficient protocols than the conservative one could be used. We have chosen the conservative protocol, though, due to its behavior in the face of corruption. Informally, even if a causal chain beginning in an uncorrupt group passes through a corrupt group, the corrupt group cannot generate a message based upon the incoming information and effect its delivery to a member of the first group prior to the delivery of the initial multicast of the chain. Returning to part (b) of figure 1, what we mean is that even if $s_2 \in \mathcal{C}$ it could not manage to get m_3 delivered to p_1 before m_1 as in part (c) of the figure.

Thus, while in this paper we have not formally defined the notion of a causal chain which passes through a corrupt group, in an informal sense the conservative protocol provides even a stronger causality guarantee than we had promised. This type of guarantee may be important to applications in which the *timing* of the release of information is important. For instance, if G_1 represents the trading service of section 1 and m_1 contains instructions to buy a large quantity of a certain stock for a client, then after discovering the intended purchase via m_2, the intruder may wish to deliver a purchase order m_3 for that same stock to p_1, before m_1 is delivered to p_1. The conservative protocol prevents this sort of attack.

We conclude this section with mention of an additional method by which groups can protect the causal chains on which they rely. In Isis, a *causality domain* [BCG91] is a set of groups among which causality is preserved. So far we have assumed that all groups are contained in the same causality domain, but in reality Isis supports many such domains. We are currently considering ways to protect access to causality domains as an additional fire wall against malicious intrusion. However, this approach has not yet been sufficiently investigated, and we will not discuss it further here.

5 Delegation and Access Control

The guarantees provided to a process group in section 4 are contingent upon the group being uncorrupt. In light of this, it is obvious that in real systems, access to groups must be restricted. This is also required if a programmer wishes to build a "trusted" group: a group obviously cannot be trusted if *any* process may join it simply by so requesting!

As in any situation requiring access control, we have available to us two basic approaches, namely access control lists (ACLs) and capabilities. The advantages generally cited for ACLs include that they more naturally solve traceability, confinement and revocation problems. Capabilities, on the other hand, better support the principle of least privilege, allow for more efficient, decentralized transfer of access rights, and in general can be verified more efficiently than ACLs can be checked. Several authors have argued for hybrid schemes which exploit the advantages of both approaches (e.g., [KH84,Gon89]).

While the advantages traditionally cited for each approach also apply in our setting, we argue that the advantages of capabilities are less applicable to our needs. First, in the majority of Isis applications, the membership of a typical process group is relatively static. That is, group joins are infrequent in comparison to other group operations, such as multicasts. Thus, although in general capabilities can be verified more efficiently than ACLs can be checked, we expect that in many cases this would have no significant effect on overall system performance. Second, a major pitfall of classic capability systems, namely that the ability to access an object implies the ability to grant access to the object, seems particularly hazardous in our system model. By passing capabilities to sites outside the group, the group places trust in those external sites to not propagate the capabilities in unintended ways. This may at first appear to be a moot point, because by granting a capability to a corrupt site, the group has also entrusted a corrupt site to enter a process in the group. However, if the site is subsequently suspected of being corrupt and is placed on an exception list to prevent any process on that site from joining the group, there is still no way to prevent the corrupt site from passing the capability to others. Indeed, once the capability has been passed to a corrupt site, the ability to control access to the group based upon the capability is nullified, because the capability may propagate unchecked.

For these reasons, classic capabilities are not the access control mechanism of choice in our system, and instead we view ACLs, or possibly a hybrid scheme, as being more suitable. In the remainder of this section, we describe an access control scheme based only upon ACLs which we plan to employ in our system. This scheme is sufficiently powerful to be used as the sole means to control access to groups, although it could also easily be adapted for use in a hybrid scheme such as that in [Gon89].

The straightforward criteria on which to restrict access to groups is the owner and site of the process requesting access. That is, when a group is created, the creating process would specify a set (i.e., ACL) of ⟨owner, site⟩ pairs which indicates the processes which would be allowed to join the group. The problem with this approach is that it is not sufficiently expressive. Consider an extension of the NYSE example of section 1 in which a client process authorizes the brokerage service to purchase stocks with funds in the client's account at XYZ bank. After locating the stock, the brokerage service must send a representative to a group established by XYZ bank to arrange the fund transfer for the stock purchase. However, XYZ bank will admit the representative to this group only if it has been legitimately authorized by a client of the XYZ bank. Thus, the simple scheme of admitting the representative based upon its owner (say, an individual stock broker) and hosting site is insufficient here for two reasons: this information neither convinces the bank group that the process represents the brokerage nor conveys the authorization granted by the client.

This flavor of authorization is closely related to many concepts which have appeared in the literature in recent years, including *authentication forwarding* [SNS88], *cascaded authentication* [Sol88], and *delegation* [GM90]. Informally, each of these terms denotes the means by which one party authorizes another to act on its behalf, as exemplified by the client delegating authority to the stock brokerage in the previous example. The *delegation problem* in this example is how the brokerage representative can convince the bank group to admit it, given that a client has legitimately delegated authority to the brokerage service.

The delegation problem in a group oriented system is different from that in other systems only in the sense that groups, instead of processes, are delegating and being delegated. In practical terms, this means that groups need to be authenticated, instead of processes or sites. Fortunately, we already have in place the mechanisms to do this, namely group keys and the name service introduced in sections 4.1 and 4.2.

The approach we take to delegation is best illustrated by an example. Suppose that group G_1

wishes to delegate authority to group G_2. To do so it sends to G_2 the message

$$G_1, T_1, G_2, S_1(T_1, G_2), \qquad (2)$$

where "T_1" is the time at which this delegation expires, "S_1" denotes the signature function of G_1 (i.e., signature with the private key of G_1), and "G_1" and "G_2" are the *names* of G_1 and G_2, respectively.[16] Intuitively, a member of G_2 can present (2) to another party to prove that any member of G_2 can speak on behalf of G_1 until time T_1. The other party verifies this claim with the address of G_1. (Recall that the address for a group now contains the group's public key.) Moreover, a process in G_2 can delegate further to group G_3 by sending it

$$G_1, T_1, G_2, T_2, G_3, S_2(S_1(T_1, G_2), T_2, G_3). \qquad (3)$$

A recipient of this message should believe that a member of G_3 has authority until time $\min\{T_1, T_2\}$ to act on behalf of G_1,[17] provided that the message can be verified by the appropriate public keys. Of course, G_3 could delegate yet further in a similar fashion, and in general, such *delegation chains* could become arbitrarily long. This scheme has many of the same features as that in [Sol88], and the reader is referred there for further discussion.

A problem with this delegation scheme is that the delegating group has no means by which to restrict the authority it grants to the group it delegates. For instance, after delegation (2), object monitors may allow members of G_2 to access any resource that G_1 could, including those that G_1 did not intend. This is a problem common to many delegation schemes, and it can be dealt with in several ways. In [Sol88], with each delegation is included a set of *constraints* which may explicitly specify the subset of resources normally accessible to the delegating party to which the delegated party should be granted access. It appears more difficult to develop a general access control mechanism using constraints, though, because the form of the constraints may be too application specific. Another approach, which is taken in [GM90] and which we adopt here, is to limit the access rights of the delegated party through the use of *roles*. Associated with each role of a group is some subset of the access rights of the group. When a group delegates the authority of a role, the authority transferred to the delegated group is only that of the role, and not of the delegating group. So, in the NYSE example, the client group could delegate authority to the

[16]This form of delegation conceivably could admit the use of group addresses instead of names, although for the purposes of access control we allow delegation by name only.

[17]Of course, an application may interpret (2) and (3) differently than as we have stated. For example, based upon (3) an object monitor may grant a member of G_3 the access rights of G_1, G_2, or some combination thereof.

brokerage service under a role which was used to establish the bank account and which would be useless, say, for reading the client's mail. Although delegation via roles is less flexible and requires more forethought than the use of constraints, in our case the use of roles is particularly attractive, because a role corresponds to just another group name. That is, a group can create roles for itself by registering other names for the group with the name service.

We can now extend this delegation mechanism into an access control scheme as follows. A process specifies access control policy for a group by providing a set of *delegation templates* when it creates the group, in addition to the ⟨owner, site⟩ pairs previously described. (Subsequently, a member of the group can change the access control policy for the group by adding or removing delegation templates or ⟨owner, site⟩ pairs, although doing so does not remove any members from the group.) A delegation template is a list $\mathcal{G}_1, \ldots, \mathcal{G}_n$ where each $\mathcal{G}_i$ is a set of group names. A delegation template specifies a set of delegation chains which are acceptable credentials for a process to join the group. A delegation chain

$$G_1, T_1, \ldots, G_{m-1}, T_{m-1}, G_m, \mathcal{S}_{m-1}(\ldots). \qquad (4)$$

is said to *match* the delegation template $\mathcal{G}_1, \ldots, \mathcal{G}_n$ if $m \geq n$ and for all j satisfying $1 \leq j \leq n$, $G_j \in \mathcal{G}_{m-n+j}$. That is, the chain in (4) matches the template $\mathcal{G}_1, \ldots, \mathcal{G}_n$ if the chain ends with a sequence of delegations beginning with an element of $\mathcal{G}_1$, followed by an element of $\mathcal{G}_2$, and so on, and ending with an element of $\mathcal{G}_n$.

Given a set of delegation templates, access to a group is controlled as follows. Suppose the group contact receives message (4) embedded in a request from some process p to join the group. Then, p is allowed to join if and only if

1. the authenticity of message (4) can be verified with the appropriate public keys,
2. p's site can vouch that p is in G_m (by illustrating knowledge of the private key for G_m),
3. message (4) matches a delegation template for the group,
4. none of the delegations in message (4) have expired, and
5. the ⟨owner, site⟩ pair of p is listed in the set of ⟨owner, site⟩ pairs for the group.

In this way, the sets of delegation templates and ⟨owner, site⟩ pairs together constitute an ACL for the group.

Finally, we note that this access control mechanism can be extended to objects other than groups. We have already seen one need for this, namely the directories of the hierarchical name space implemented by the name service described in section 4.2. As in many file systems, a name service directory has three natural types of access to it, namely *search*, *read* and *write*. So, as when creating a group, a process can specify when creating a directory in the name service a set of delegation templates and a set of ⟨owner, site⟩ pairs for each of these types of access.

6 Conclusion and Future Work

In this paper we have described a security architecture for use in the Isis toolkit, but structured in such a way that most mechanisms should also be useful in other group oriented settings. The major features of the security architecture include a group oriented authentication subsystem, a secure method for joining groups, and protocols which protect certain causal delivery ordering guarantees. In addition, we have proposed an access control scheme based upon delegation for use in group oriented settings.

Future work on this system is heading in several directions. First, the system is currently being implemented at Cornell University. This implementation should provide valuable insight into the efficiency of our architecture and mechanisms. It is also forcing us to consider user interface issues — while such constructs as delegation templates will certainly force the Isis interface to change, we would like to ensure that currently existing Isis applications can benefit from the new mechanisms with minimal changes. Second, in addition to the extensions proposed in the previous sections, we are pursuing other improvements to the basic architecture. For example, we are currently investigating ways to incorporate information flow controls into Isis and to relate this work to the TCSEC taxonomy [DoD83], which is a set of criteria, covering issues such as access control, information flow, and covert channels, for classifying systems according to the levels of security assurance they provide. Third, we are also considering methods of exploiting the Isis abstractions, once secured, to enhance the overall security of applications.

7 Acknowledgements

We are deeply appreciative of the Isis group at Cornell University for contributing many interesting discussions regarding this research. In particular, we thank Brad Glade, Robbert van Renesse, Aleta Ricciardi, and Patrick Stephenson. Richard Platek, Sam Toueg and Raphael Yahalom also provided

many helpful discussions. Mark Steiglitz developed a mechanism for authenticating user identifiers under Isis [Ste91]; this work did not become a permanent part of the system but was a useful source of insight into the problem of authenticating UNIX user identifiers within the Isis environment.

References

[AR87] D. P. Anderson and P. V. Rangan. A basis for secure communication in large distributed systems. In *Proceedings of the 1987 IEEE Symposium on Security and Privacy*. April 1987.

[BCG91] K. P. Birman, R. Cooper, and B. Gleeson. Programming with process groups: Group and multicast semantics. Technical Report TR91-1185. Department of Computer Science, Cornell University, January 1991.

[Bir85] A. D. Birrell. Secure communication using remote procedure calls. *ACM Transactions on Computer Systems*, 3(1):1–14, February 1985.

[BJ87] K. P. Birman and T. A. Joseph. Reliable communication in the presence of failures. *ACM Transactions on Computer Systems*, 5(1):47–76, February 1987.

[BLNS86] A. D. Birrell, B. W. Lampson, R. M. Needham, and M. D. Schroeder. A global authentication service without global trust. In *Proceedings of the 1986 IEEE Symposium on Security and Privacy*, pages 223–230, April 1986.

[BSS90] K. P. Birman, A. Schiper, and P. Stephenson. Fast causal multicast. Technical Report TR90-1105, Department of Computer Science, Cornell University. April 1990. To appear in *ACM Transactions on Computer Systems*.

[Cri89] F. Cristian. Probabilistic clock synchronization. *Distributed Computing*, 3(3):146–158, 1989.

[CZ85] D. R. Cheriton and W. Zwaenepoel. Distributed process groups in the V kernel. *ACM Transactions on Computer Systems*, 3(2):77–107, May 1985.

[DES77] Data encryption standard. National Bureau of Standards. Federal Information Processing Standards Publication 46, Government Printing Office. Washington. D. C., 1977.

[Dif88] W. Diffie. The first ten years of public-key cryptography. *Proceedings of the IEEE*, 76(5):560–577, May 1988.

[DoD83] Department of Defense trusted computer system evaluation criteria. CSC-STD-011-83. Department of Defense Computer Security Center, Fort Meade, Maryland, August 1983.

[DS81] D. E. Denning and G. M. Sacco. Timestamps in key distribution protocols. *Communications of the ACM*, 24(8):533–536, August 1981.

[GGKL89] M. Gasser, A. Goldstein, C. Kaufman, and B. Lampson. The Digital distributed system security architecture. In *Proceedings of the 12th National Computer Security Conference, NIST/NCSC*, pages 305–319, October 1989.

[GM90] M. Gasser and E. McDermott. An architecture for practical delegation in a distributed system. In *Proceedings of the 1990 IEEE Symposium on Security and Privacy*, pages 20–30, May 1990.

[Gon89] L. Gong. A secure identity-based capability system. In *Proceedings of the 1989 IEEE Symposium on Security and Privacy*, pages 56–63, 1989.

[Gon91] L. Gong. A security risk of depending on synchronized clocks. Submitted for publication, September 1991.

[KH84] P. A. Karger and A. J. Herbert. An augmented capability architecture to support lattice security and traceability of access. In *Proceedings of the 1984 IEEE Symposium on Security and Privacy*, April 1984.

[KT91] F. M. Kaashoek and A. S. Tanenbaum. Group communication in the Amoeba distributed operating system. In *Proceedings of the 11th International Conference on Distributed Computing Systems*, pages 222–230, May 1991.

[Lam78] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, July 1978.

[LLS90] R. Ladin, B. Liskov, and L. Shrira. Lazy replication: Exploiting the semantics of distributed services. In *Proceedings of the Ninth Annual ACM Symposium on Principles of Distributed Computing*, August 1990.

[OSS80] J. K. Ousterhout, D. A. Scelza, and P. S. Sindhu. Medusa: An experiment in distributed operating system structure. *Communications of the ACM*, 23(2):92–105, February 1980.

[PBS89] L. L. Peterson, N. C. Buchholz, and R. D. Schlichting. Preserving and using context information in interprocess communication. *ACM Transactions on Computer Systems*, 7(3):217–246, August 1989.

[RB91] A. M. Ricciardi and K. P. Birman. Using process groups to implement failure detection in asynchronous environments. In *Proceedings of the Tenth Annual ACM Symposium on Principles of Distributed Computing*, pages 341–351, August 1991.

[Riv90a] R. L. Rivest. Cryptography. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science, Volume A, Algorithms and Complexity*, chapter 13, pages 717–755. Elsevier Science Publishers B. V., 1990.

[Riv90b] R. L. Rivest. The MD4 message digest algorithm. Internet RFC 1186, October 1990.

[Sat89] M. Satyanarayanan. Integrating security in a large distributed system. *ACM Transactions on Computer Systems*, 7(3):247–280, August 1989.

[SNS88] J. G. Steiner, C. Neuman, and J. I. Schiller. Kerberos: An authentication service for open network systems. In *Proceedings of the USENIX Winter Conference*, pages 191–202, February 1988.

[Sol88] K. R. Sollins. Cascaded authentication. In *Proceedings of the 1988 IEEE Symposium on Security and Privacy*, pages 156–163, April 1988.

[Ste91] M. Steiglitz, May 1991. Personal communication.

[TA91] J. J. Tardo and K. Alagappan. SPX: Global authentication using public key certificates. In *Proceedings of the 1991 IEEE Symposium on Security and Privacy*, pages 232–244, May 1991.

[TY91] J. D. Tygar and B. S. Yee. Strongbox. In J. L. Eppinger, L. B. Mummert, and A. Z. Spector, editors, *Camelot and Avalon, A Distributed Transaction Facility*, chapter 24, pages 381–400. Morgan Kaufmann, San Mateo, California, 1991.

[VK83] V. L. Voydock and S. T. Kent. Security mechanisms in high-level network protocols. *Computing Surveys*, 15(2):135–171, June 1983.

[vR91] R. van Renesse. MUTS: Design and implementation. Unpublished manuscript, September 1991.

A Synchronized Clocks

In section 4.1, we described a site's certificate as containing "the expiration time of the public key" of the site. Presumably, a recipient of a certificate for some site should be able to determine from the expiration time whether the certificate is fresh. And, this concept of time has other applications in our system; e.g., in section 4.1 we also use timestamps to ensure the freshness of connections, and in section 5, timestamps are used to expire delegations.

In this paper, the source of time available to each uncorrupt site is assumed to be a clock which is synchronized with the clocks on all other uncorrupt sites to within ϵ time units, where for our purposes ϵ is quite large relative to that required in most other applications using synchronized clocks; e.g., an ϵ of a few seconds may suffice. While synchronized clocks are not necessary to determine the freshness of communication, it was recognized early in the literature that synchronized clocks can reduce communication in authentication protocols [DS81]. Accordingly, many security architectures, such as Kerberos [SNS88] and DSSA [GGKL89], have employed synchronized clocks for precisely this purpose.

However, it is important that clocks be synchronized *securely* if they are to be used to protect against attacks in a secure system. More precisely, the clock synchronization algorithms must be tolerant of attacks similar to those which the system must tolerate, because if the clocks can be successfully altered, then, e.g., uncorrupt sites can become vulnerable to classic replay [DS81] and suppress-replay attacks [Gon91]. Since most operating systems do not provide such a secure source of time, we are forced to implement our own synchronized clocks or to find a suitable alternative.

Fortunately, a clock synchronization algorithm based upon the work in [Cri89] is currently being implemented for use in a real-time extension of the Isis toolkit. The algorithm employs a fault tolerant *master clock*, around which a set of *slave clocks* (i.e., sites) synchronize periodically by requesting the master's clock value, measuring the round-trip response time from the master, and approximating the master's value based upon the response time and the value in the message. While detailed discussion of the clock synchronization algorithm is outside the scope of this document, we note only that the master clock will employ MUTS for communication and thus will be amenable to authentication via the mechanisms described in section 4.1. Therefore, we expect that securely synchronized clocks which satisfy our relatively modest requirements can be achieved easily, provided that the master clock itself is not corrupt.