Integrating Security in a Group Oriented Distributed System*

Michael Reiter
Kenneth Birman
Li Gong

TR 92-1269
(replaces TR 91-1239)
February1992

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 NAG 2-593, 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.

Integrating Security in a Group Oriented Distributed System[*]

Michael Reiter
Dept. of Computer Science
Cornell University
Ithaca, NY 14853
reiter@cs.cornell.edu

Kenneth Birman
Dept. of Computer Science
Cornell University
Ithaca, NY 14853
ken@cs.cornell.edu

Li Gong
ORA Corporation
675 Massachusetts Ave.
Cambridge, MA 02139
li@cambridge.oracorp.com

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 make common group oriented abstractions robust in hostile settings, in order to facilitate the construction of high performance distributed applications that can tolerate both component failures and malicious attacks. These abstractions include process groups and causal 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 cryptosystems and key exchange protocols 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., [4, 24, 28, 17]). 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 [1]. Process groups have been incorporated into many distributed systems (e.g., [18, 5, 2, 19, 23, 14, 13]) and have been shown to simplify the implementation of fault tolerant applications [5, 2, 14]. The benefit in preserving process group abstractions in hostile environments could therefore be great. In particular, it would facilitate the construction of applications that can tolerate both component failures and malicious attacks.

To illustrate this need, consider a stock brokerage that plans to establish a trading service to trade stocks for a certain industry on the over-the-counter market (OTC). The trading service will reside in a large distributed system shared by other brokerages and traders. Because the brokerage is a *market maker* for that industry, other brokerages and traders buy and sell stocks in that industry through that brokerage, and so the service will be utilized by other trader's applications. The availability and performance of this service is crucial, and thus it must be replicated. Accordingly, the firm's programmers might like to implement this service as a fault tolerant process group using their favorite group oriented programming environment.

At the same time, however, the brokerage is concerned about the security of its service. The brokerage plans to execute this service on a potentially dynamic set of sites; e.g., the brokerage may rent additional machines during times of heavy load in order to run additional servers. And, while the brokerage is confident that it can protect these sites from corruption while servers are executing on them, it cannot trust that other sites, or the network by which the sites communicate, will behave as expected. For instance, corrupt traders may attempt to infiltrate the group, alter or forge group communication, or tamper with the group abstractions on which the consistency and correctness of the service would rely. So, if the abstractions provided by the group oriented programming environment are not robust against malicious attack, it

[*]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.

is doubtful whether it should be used at all. Moreover, the brokerage is not willing to entirely trust all of its own brokers, and thus needs some way to enforce security policies even within the set of protected sites.

Generalizing from this example, we are concerned with facilitating the construction of high performance, fault tolerant applications on a secured and potentially dynamic "island" of sites. We approach this task by making common group oriented abstractions robust on this island, despite malicious attacks against them. A second goal is to provide support for group oriented security policies within such an island. Third, since applications may need to interact with less trusted parties outside of this island, we feel it is important to provide guarantees regarding the results of such communication.

Another impetus to preserve process group abstractions in hostile environments is that the cryptographic community has identified several practical security needs in settings where groups occur naturally [7]. Proposed solutions (e.g., [10, 15, 8]), however, presuppose a group oriented infrastructure which cannot be effectively provided in a hostile environment by current systems. Secure group oriented foundations will enable applications to more easily realize the benefits of this research.

This paper presents a distributed security architecture to be integrated with group oriented systems and, in particular, with the Isis toolkit [2].[1] Isis is a toolkit for distributed programming that provides process group and reliable group multicast abstractions. With respect to Isis, the aims of this work are twofold. 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. The second is to enhance the Isis abstractions to be more suitable for use in a hostile environment. And, of course, we wish to accomplish these without unreasonably degrading the performance of the toolkit.

The goal of this paper is to describe the major features of our security architecture. More precisely, we discuss how our architecture addresses three needs, namely for group oriented authentication, secure methods for joining groups, and causal multicast protocols that are appropriate in hostile environments. We also propose a delegation and access control scheme for use in group oriented systems. We do *not* discuss key exchange protocols or specific cryptosystems in detail. We instead focus on the mechanisms we use to protect the abstractions that 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 an informal description of the abstractions provided by Isis. In section 3 we discuss the system model for which Isis was designed and then weaken this model to allow malicious behaviors. Section 4 clarifies our goals and presents the three basic aspects of our security architecture previously mentioned. Section 5 describes our group oriented delegation and access control scheme. We end with a discussion of the status of the system and future directions of research.

2 The Isis abstractions

The basic abstraction provided by Isis is the *process group*, which is simply a collection of processes with an associated *group address*. Groups may overlap and be nested arbitrarily, and processes may create and join groups at any time. And, a process may be removed from a group, either because it was perceived to fail (i.e., crash) or because it requested to leave. A group G can thus be seen as progressing through a sequence $view_0(G)$, $view_1(G)$, ... of *views* satisfying the following conditions.

- For all i, $view_i(G) \subseteq P$, where P is the set of all processes in the system.
- $view_0(G) = \emptyset$.
- For all i, $view_i(G)$ 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 is modeled as a sequence $e_p^0, e_p^1, \ldots$ of *events*, each corresponding to the execution of an indivisible action. One such event is the *delivery* of the i-th group view $view_i(G)$ of a process group G. Views are delivered to processes in sequential order, with the constraint that the i-th view of G is delivered to p only if $p \in view_i(G)$. If the i-th group view of G is delivered to p, then we say that p is *in* the i-th group view until the $(i+1)$-th view is delivered or p is removed from the group.

The primary means of communication in Isis is *group multicast*. (Point-to-point communication is treated essentially as multicast in a static group of size two, where any membership change destroys the

[1] This 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.

group.) A process in (some view of) a group can multicast to the group by specifying the group address as the destination. Isis guarantees to processes certain delivery properties regarding group multicasts. For instance, all operational destinations eventually deliver the message or, and only if the sender fails, none do. And, all destination processes are in the same group view when the message is delivered, and the set of destination processes is precisely the members of that view.

One guarantee of particular interest in this paper is that messages are delivered in an order consistent with *potential causality*. Informally, one message is said to be *causally before* another if the former was sent before and could have affected the latter [16]. Causal delivery ordering has been shown to be crucial in systems such as Isis that exploit asynchronous operations to achieve high performance [1], because when messages are asynchronously pipelined to destinations, a message could be received at the destination site before another causally before it. The danger is that, e.g., an update to a data structure could be delivered to an application before the message initializing the data structure. A delivery ordering that respects causality avoids such problems.

To define the causal delivery ordering in our setting, we introduce two more types of events that can be executed by a process p in a group G: the multicast of a message m to G, denoted $\mathbf{mcast}_p(m, G)$, and the delivery of a message m multicast to G, denoted $\mathbf{deliver}_p(m, G)$. The potential causality relation $\rightarrow$ is defined as the transitive closure of the smallest relation $\leadsto$ satisfying the following conditions.

- For all i and p, $e_p^i \leadsto e_p^{i+1}$.
- 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 primitive that provides this property is called CBCAST and is a fundamental building block on which other Isis abstractions are implemented [3].

To summarize, Isis provides process group and group multicast abstractions. Notification of group membership changes are coordinated with respect to the delivery of group multicasts. And, multicast deliveries to a process are ordered causally to allow Isis to safely exploit asynchronous communication.

3 The system model

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

The system model considered 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. The intruder can engage in any active network attack (including denial of message service) and can view all network traffic. However, we assume that the intruder is limited in these attacks by the (conjectured) properties of the cryptosystems and signatures schemes we employ. We also assume that the intruder does not engage in *traffic analysis* attacks: i.e., we do not consider such attacks here and assume that encryption is sufficient to hide the contents of a message from a network intruder. The reader should see [30] for a survey of network attacks.

We also assume that sites may be corrupted by an intruder during system execution. Once corrupted, a site may exhibit arbitrarily malicious behaviors, again limited by the cryptosystems and signature schemes we employ. For simplicity, we assume in this paper that once a site is corrupted, it remains so forever. Intuitively, a corrupt site is one on which the operating system or Isis code or data has been maliciously or accidentally altered or replaced.

We make two assumptions about the operating system running at an uncorrupt site. First, we assume that it authenticates in a secure fashion the user identifiers of local processes.[2] Second, we assume that it provides protected, private address spaces for, and private, authentic message passing between, all local system and user processes. This includes the protection

[2]This is a rather strong requirement, but the mechanisms described in this paper facilitate its implementation. For example, if smart-card technology is available, then 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 [11]. Even without such technology, the authentication mechanisms of section 4.1 provide a secure communication channel between any two sites that facilitates the authentication of user identifiers.

of virtual address spaces stored on external media.

In this paper we also assume that each uncorrupt site maintains a local clock that is synchronized with the clocks on all other uncorrupt sites. A clock synchronization algorithm is currently being implemented as part of a real-time extension of the Isis toolkit. While a detailed discussion of this algorithm is outside the scope of this paper, we note that it is based upon a *time service* which, with the authentication mechanisms described in this paper and physical safeguards to ensure its integrity, can be adapted for use in our system model.

Finally, we assume that the Isis *failure detector* and *name service* behave according to specification (i.e., are not corrupt). These are services provided by Isis and will be discussed briefly in sections 4.1 and 4.2, respectively. The security of these services is a topic of ongoing research; one possible approach is described in [20].

4 Protecting the Isis abstractions

Informally, we would like to make the Isis abstractions described in section 2 robust in the system model of section 3. In particular, we would like to ensure that all processes in a group observe correct deliveries of group views and messages. This is clearly impossible, however, if a site hosting a group member is corrupt, because that site could cause arbitrary events to be observed in any order by any process it hosts. We are thus faced with two reasonable options: we can either attempt to guarantee the Isis abstractions in all groups, but to only those processes that reside on uncorrupt sites, or we can attempt to guarantee the Isis abstractions only in groups that have no members on an uncorrupt site.

We have opted for the latter for several reasons. First, the latter approach is more consistent with our original motivation, namely to enable a programmer to construct a fault tolerant application *on trusted sites*, despite the fact that the network or other sites cannot be trusted. Second, the former could be achieved only through great cost to the performance of the system; e.g., all correct sites would be required to reach consensus on the contents of each group multicast before delivering it to the application. Such an overhead would be unacceptable to the type of applications for which Isis is intended. Third, very few of the Isis applications we have seen could tolerate the corruption of some group members, even if the Isis abstractions were provided to the correct members. And, we expect that very few developers would be willing to pay the performance penalty of making their applications tolerant of such corruptions.

A consequence of this design decision is that processes residing on untrusted sites should not be allowed to join a "trusted" group. In section 5, we present a method by which access to groups can be controlled. In those applications that require communication with processes on less trusted sites (e.g., the trading service of section 1), we suggest that such communication be performed in a point-to-point fashion between the untrusted process and a single member of the group. Then, for instance, the untrusted process could multicast to the group by sending the message to the member, which would multicast the message on behalf of the process. This approach has the disadvantage that a multicast from the untrusted site will be somewhat slower. However, it is beneficial in that it virtually eliminates any threat that the corrupt site could pose to the consistency of the group members (from the point of view of Isis); the one exception to this is discussed in section 4.3.

Continuing, then, we restrict our efforts to process groups that reside on uncorrupt sites. 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 at any time during execution, if site s is corrupt and $view_i(G)$ is the current group view, then $s \notin sites_i(G)$. Then, in this work we enhance Isis to ensure that if G is uncorrupt, then processes in $\bigcup_i view_i(G)$ observe correct sequences of events associated with group G, and "external" operations that involve G, such as group joins and communication with sites outside of G, can be performed with certain security guarantees. Henceforth, when we speak about a process group, we assume that it is uncorrupt, unless otherwise stated.

As mentioned in section 1, in this paper we focus on three issues that are fundamental to our goals. To avoid an indepth discussion of how the Isis abstractions are presented to processes, we will not describe how the mechanisms we present here are applied to implement specific abstractions (e.g., the delivery of group views). Instead, we concentrate on more basic problems that could undermine these abstractions and that must be addressed in any system offering similar functionality.

First, a necessary step is to develop a subsystem that provides efficient authentication of group multicasts. 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

With such a subsystem, a site in the group could rely upon the authenticity of messages apparently from other sites in the group. In addition, since altered messages would be detected (and ignored), attacks on the integrity of messages would become indistinguishable from lengthy message delivery times, as are denial of message service attacks already. And, since Isis is constructed for an asynchronous network, Isis would behave safely under such attacks. (Intuitively, here we are reducing these active network attacks to communication failures.) In section 4.1 we propose such an authentication subsystem.

Second, the protocol by which a process joins a group is crucial to the process group abstraction, because the joining site's perception of the group membership is directly dependent upon the security of the join protocol. In Isis, when a process requests to join a group, it specifies the group address. This address is used by the process' site to send the appropriate request to the group, and it is necessary that the process' site be able to authenticate the apparent response of the group. A related issue that 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 discuss these issues and extend our architecture to address them.

Third, in section 4.3 we discuss the task of preserving causality among multicasts in a hostile environment. We argue that simply preserving causality among messages in one or several uncorrupt groups may not suffice for groups that must communicate with less trusted principals. We then formulate the specific causal guarantees we provide, and describe protocols to implement them.

4.1 Multicast authentication

We introduce authentication mechanisms at the lowest layer of the Isis toolkit, namely the Multicast Transport Service (MUTS) [29]. 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 multicast communication to other sites. MUTS provides these abstractions while insulating the higher layers from the particular transport protocol used, which may exploit hardware multicast capability.

For our purposes, authentication must be introduced at the MUTS layer. Since MUTS is the lowest layer of the Isis toolkit, we cannot introduce authentication mechanisms closer to the network, and because Isis is designed to run on many different platforms, we cannot rely upon authentication mechanisms being available at lower layers. 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. For example, a network intruder could then undetectably tamper with the information MUTS uses to sequence messages.

Before presenting our authentication mechanisms, we briefly consider how MUTS works. Each MUTS layer maintains a current *member list* of sites for each group it (or rather, its site) is in. A copy of MUTS learns about changes to the site membership of a group from the layer above it, which communicates with other sites in the group and with the Isis *failure detector* [2, 21] to make this determination. When MUTS receives a message from a higher layer to be multicast to a group, it opens a *connection* to the members of its current member list for that group, if one does not already exist. MUTS then breaks the message into packets, and hands these packets to the transport protocol for transmission. A connection is associated with exactly one group 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 originator's member list, the old connection is disassembled and a new connection is negotiated for the new member list. 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.

Techniques for authenticating messages (or in this case, packets) have existed in the literature for many years. Traditionally, these methods have employed encryption, but methods based upon *pseudo-random functions* are also theoretically attractive. Informally, a pseudo-random function f has the property that if f is unknown, then it is computationally infeasible to produce $f(m)$ for any m with a probability of success greater than random guessing, even after having seen other $(m', f(m'))$ 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 that share a secret key K can authenticate messages between each other by appending $f_K(m)$ to each message m [22]. (In Isis, we will efficiently approximate this technique, e.g., with one-way hash functions [27].)

For MUTS we generalize these ideas to take advantage of hardware multicast capabilities that may [illegible] 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 packets sent on the connection. When a connection is created, the site initiating the connection generates a new connection key K and distributes it, in a fashion to be discussed below, to the sites on its member list. Then, the multicast "$m, f_K(m)$" of packet m on the connection can be verified at all destinations. In addition, an application can request that its message be encrypted, in which case any fragment of that message included in packet m will also be encrypted under K (e.g., using DES [6]). Provided that the connection is fresh (i.e., established with a non-replayed message), each destination can verify that m is not a replay, because it contains the sequence number for the connection. Here we do not detail the protocol by which a connection is opened, although we remark that the freshness of a connection is guaranteed by incorporating a timestamp into the connection initiation message.

Because the establishment of a connection is a somewhat frequent occurrence, it is important that a connection key be distributed efficiently. The method for distributing a connection key should, if possible, require a single multicast and a single encryption (versus one encryption for each site on the originator's member list). In Isis, we achieve this by employing a *group communication key* (or just *communication key*). A communication key is a shared key possessed by all sites in the group. The communication key for a group is created by the site hosting the first member of the group, and as other processes join, it is given to their sites as described below. The connection key for a connection is thus communicated in a single multicast, encrypted with the communication key of the group.

In an open network environment, key exchange eventually requires the intervention of some *a priori* trusted authority, which often takes the form of an *authentication service*. In Isis we employ such a service to establish secure channels by which communication keys can be distributed. We choose a public key authentication service due to the security advantages that can be achieved [9]. 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 each site, along with the site's own *site key* (a private key/public key pair), when it is booted.[3]

[3]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.

Once booted, the site obtains from the authentication service its *certificate*, which contains the identifier and public key of the site and the expiration time of the certificate, all signed by the private key of the authentication service. Each site's certificate is subsequently disseminated from the site itself, with the site contacting the authentication service for a new certificate when its certificate expires. The exact implementation of the authentication service is a topic of ongoing research; one possible approach is described in [20].

A communication key is sent to a site after obtaining the site's certificate and encrypting the communication key under the site's public key. This exchange takes place as part of the group join protocol, which will be described in the next section. We emphasize that no interaction with the authentication service is required when a communication key is distributed.

To summarize, we propose to protect group communication via a hierarchical key distribution scheme. The initial keys in the system are the site keys. A group communication key is created when the group is created and distributed using site keys when group joins take place. This communication key, in turn, is used to establish connection keys within the group. The benefits of this scheme are many. First, the use of shared keys and efficient algorithms at the level of connections should result in minimal performance cost for group multicasts, which are by far the most common operations in most Isis applications today. Second, since a different key is used per connection, the lifetime of a connection key is limited, and a different key is used for each multicasting source. This should limit an intruder's ability to cryptanalytically attack connection keys. Third, the more costly public key operations are performed less frequently, when groups are joined.

Finally, we emphasize that *sites* hold connection keys, communication keys, and private keys of sites; user processes do not. This prevents a malicious user process on an uncorrupt site from improperly distributing or using these keys. Moreover, when a process is removed from a group, the site on which it resides can destroy the communication and connection keys that it held on behalf of the process. By doing so, if the site is subsequently corrupted, then the intruder will not be able to masquerade as a group member. If the entire site crashes, we rely upon the loss of volatile storage to eliminate all keys from memory.

4.2 Joining groups

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 Isis, the protocol for a process to join a group is 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 and, once admitted, is sent a list of group members by some site in the group. Note that if the requesting site includes its certificate in the request, then the site that returns the member list can also return the group communication key encrypted under the site's public key.

In order for the join protocol to be secure, the process' site must be able to authenticate the response as being from a site in the group. And, since the process' site does not know what sites are in the group, the site keys of section 4.1 do not suffice for this. To facilitate the required authentication, we introduce a new type of key. When a group is first created, the initial site in the group creates a public key/private key pair, the private key of which is called the *group key* (or just the *private key of the group*). The group key is then communicated to group members' sites just as the communication key is, as part of the join protocol. While these operations will reduce the performance of group creates and joins, they have no effect on the performance of group communication. Like communication keys, the group key is destroyed by a site when all group members it hosts are removed from the group (or when the site itself fails). We utilize this new type of key by including the public key of the group in the group address. This enables a joining process' site to verify a site's membership in the group by challenging it to prove possession of the group key.

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. Other than by creating the group, 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 that implements a hierarchical name space, like that of a file system except with group addresses (or other information) at the leaves instead of files. A group name is a path from the root to a leaf in that hierarchical name space. A process can register the group address under some name at the name service anytime after the group is created. A group that 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. In particular, if it can be verified that the sending process is a member of either a "trustworthy" process group or a group that was *delegated* by such a group (see section 5), 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 requires that the verifying process be able to obtain authentic group addresses from the name service.

Thus, the ability to obtain authentic group addresses through the name service is fundamental to the security of group joins. The authentication mechanisms of section 4.1 can easily be adapted to allow a site to authenticate the name service, e.g., by having the name service sign group addresses with its private key and having the authentication service produce a certificate for the name service. However, as it stands there is still no reason for the process to believe an address obtained from the name service, because the name service cannot verify the integrity of the addresses it receives and stores. To compensate for this problem, we allow processes to impose access controls upon the directories of the hierarchical name space. That is, when a process creates a directory of the name space, it specifies access control policy for the directory that restricts which 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 represented and enforced. The name service will allow only an authorized process (residing on an authorized site, which can be authenticated as in section 4.1) to register a group in the directory. A process can then obtain a group address from a directory it "trusts" based upon the directory's owner and access control policy.

4.3 Causal multicast

As previously described, the CBCAST protocol implements Isis' causal delivery ordering property and is central to the Isis system. For pedagogical reasons, we begin this section with two illustrations of causal multicast ordering.

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 a group of arrows emanating from a point on a vertical line represents a CBCAST (i.e., causal multicast) to the group. An arrow ending at the vertical line below a process indicates the delivery of the multicast represented by the arrow to the process. (We have omitted drawing the delivery of the message to the sending process, which can be done immediately.) In this example, p_1 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_4 after m_1, as indicated in part (a) of figure 1. The method by which s_4 decides upon this delivery ordering depends on the particular CBCAST protocol used.

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

The CBCAST protocol is implemented above the MUTS layer. Thus, given the authentication mechanisms of section 4.1, the CBCAST protocol can rely upon the MUTS abstractions (i.e., at-most-once, sequenced multicast) in an uncorrupt group. As we will see in a moment, this enables us to use certain Isis protocols originally designed for benign environments to provide causal orderings among messages in a set of uncorrupt groups.

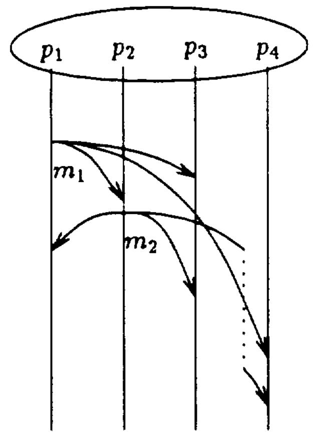
However, we argue that for applications that must interact with untrusted principals (e.g., the trading service of section 1), this guarantee may not suffice. To see why, suppose that s_3 is corrupt in part (b) of figure 1. Depending on the particular CBCAST protocol used, it may be possible for s_3, by not following the protocol, to trick s_4 into delivering m_3 to p_4 before m_1, as in part (c) of figure 1. Intuitively, s_3 might do this by omitting to include information on m_3 that conveys m_3's causal relationship with m_1. Such an attack is possible with several of the causal multicast protocols described in [3]. The danger in this is apparent in the OTC example of section 1: suppose that G_1 is the replicated trading service, m_1 contains a client's request to purchase stock, and m_2 reflects that (intended) purchase. Then, after seeing m_2, the intruder at s_3 could attempt to effect the delivery of a request m_3 for the same stock before m_1 at replica p_4 of the service, in order to raise the apparent demand for that stock and thus the price offered to the correct client. Therefore, if a causal delivery ordering between m_1 and m_3 is not enforced, a type of "insider trading" may occur.

This problem illustrates a form of attack that must be considered in any system that attempts to detect causal orderings among messages. We call this type of attack a *backdating* attack. Informally, a message m is said to be *backdated* if there exists another message m' such that m is causally after (i.e., could have been affected by) m', but at some correct site, it appears that m is not causally after m'. So, in the example above, the message m_3 is backdated, because the causality detection mechanisms at s_4 could not detect that m_3 should not be delivered until m_1 was. Naturally, there is a complementary form of attack called *postdating*, although in general it poses no threat to applications concerned with potential causality only. A more formal discussion of postdating and backdating attacks with respect to various forms of causality is outside the scope of this paper and will be presented elsewhere.

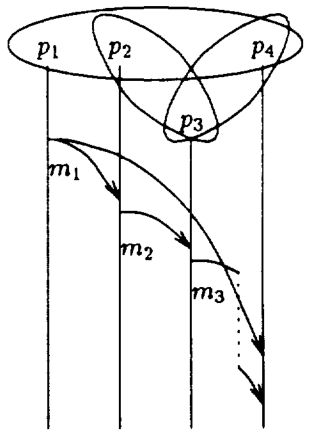
Message encryption is, in general, a prerequisite to preventing backdating attacks, because the intruder can eavesdrop on all messages and can initiate activity from a corrupt site or malicious process based upon the information so observed. To illustrate this, suppose that in the previous trading service example, the intruder observes m_1 on the network and sends m_3 based only upon information obtained from m_1 (i.e., remove m_2 from the figure). Now we have precisely the same scenario as before, in the sense that m_3 must be delivered to p_4 after m_1 in order to prevent the "insider trading" previously described. However, there is no apparent way for s_4 to detect this causal relationship. This problem can be prevented by encrypting m_1, which prevents an intruder from viewing m_1's contents. Whether encryption is justified in a specific case depends upon the particular application and message, although for simplicity we assume in the remainder of this section that all communication is encrypted as described in section 4.1.

The set of causal guarantees we provide prevents

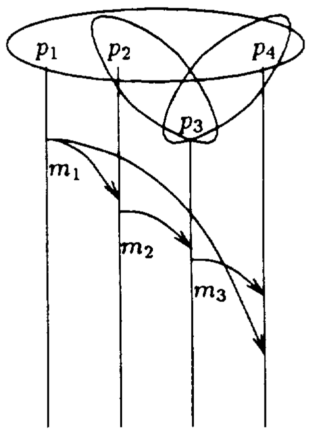
Figure 1: Causal Multicast



(a) Single group causality



(b) Multiple group causality



(c) Causality violated

backdating attacks, in addition to providing the normal causal guarantees in uncorrupt groups. To specify these guarantees, we first introduce some terminology. We say that a message is *received* at a site when MUTS at that site gives the message to the CBCAST layer, and we further stipulate that a message is *received* at the site from which it is multicast immediately after it is multicast. It is also convenient to redefine $\rightsquigarrow$ to avoid making any assumptions regarding events at corrupt sites or the semantics of multicasts in corrupt groups. Henceforth, $\rightsquigarrow$ is the smallest relation satisfying the following conditions.

- For any i and p, if e_p^i and e_p^{i+1} are executed on an uncorrupt site, then $e_p^i \rightsquigarrow e_p^{i+1}$.
- For any p, q, m and G, if G is uncorrupt and if $\mathbf{mcast}_p(m,G)$ and $\mathbf{deliver}_q(m,G)$ are executed on uncorrupt sites, then $\mathbf{mcast}_p(m,G) \rightsquigarrow \mathbf{deliver}_q(m,G)$.

As before, $\rightarrow$ is defined as the transitive closure of $\rightsquigarrow$. Suppose that $\mathbf{mcast}_p(m,G) \rightarrow \mathbf{mcast}_q(m',G')$, where G is uncorrupt. Note that by the definition of $\rightarrow$, p and q reside on uncorrupt sites (when these events are executed), although G' may be corrupt. Let r be any process to which m is delivered in G. Then, we provide the following guarantees.

- If G' is uncorrupt and m' is delivered to r in G', then m' is delivered to r after m.
- If G and G' are different groups and a message $\tilde{m}$ is (i) received at r's site after the multicast of m' from q's site but (ii) delivered to r before m, then $\tilde{m}$ was multicast before m' was multicast.

The first guarantee is simply the usual causal guarantee of section 2 applied to communication among uncorrupt groups: $\mathbf{deliver}_r(m,G) \rightarrow \mathbf{deliver}_r(m',G')$. The second guarantee is intended to prevent backdating attacks. To see this, suppose that G' is corrupt. Clearly any message $\tilde{m}$ sent on the basis of information the intruder obtains from m' can be received at r's site only after m' is multicast. And, if the attempt to backdate $\tilde{m}$ to before m were successful, then $\tilde{m}$ would be delivered before m. However, this guarantee says that if both of these conditions hold, then the intruder must have sent $\tilde{m}$ *before* seeing m'. For instance, in the OTC example of figure 1, this prevents m_3 ($= \tilde{m}$) from being delivered before m_1 ($= m$) at s_4 ($= r$'s site), because m_3 is received at s_4 after the multicast of m_2 ($= m'$).

We provide these guarantees through a combination of several protocols described in [3]. Here we describe protocols that suffice when group memberships do not change; in the remainder of this section we treat each (uncorrupt) group as simply a static set of processes. (Although we synchronize communication on view changes precisely as in [3], this introduces complexities that are best omitted for the sake of brevity.) The protocols described here also require that communication with corrupt sites be performed in a point-to-point fashion, as suggested at the beginning of section 4. We have extended these protocols to provide the above guarantees even without this requirement; one such protocol is described in appendix A.

We begin with the *vector timestamp* protocol for a single group, which provides single group causality only. More precisely, suppose we define $\rightsquigarrow_G$ for an uncorrupt group G to be the smallest relation satisfying the following conditions.

- For any $p \in G$ and any i, $e_p^i \rightsquigarrow_G e_p^{i+1}$.
- For any m and any $p, q \in G$, $\mathbf{mcast}_p(m, G) \rightsquigarrow_G \mathbf{deliver}_q(m, G)$.

As usual, let $\rightarrow_G$ denote the transitive closure of $\rightsquigarrow_G$. The vector timestamp protocol guarantees that if $\mathbf{mcast}_p(m, G) \rightarrow_G \mathbf{mcast}_q(m', G)$, then $\mathbf{deliver}_r(m, G) \rightarrow_G \mathbf{deliver}_r(m', G)$ at any common destination r.

In this protocol, a *vector timestamp* $VT_G(m)$ is appended to each multicast m in group G by the site multicasting m. Vector timestamps are assigned to messages in such a way that $VT_G(m) \prec VT_G(m')$ iff $\mathbf{mcast}_p(m, G) \rightarrow_G \mathbf{mcast}_q(m', G)$, where $\prec$ is an irreflexive partial order on the timestamps. When a message is received in a group G at a site, it is placed in the delivery sequence for a destination immediately before the earliest message m' already in that delivery sequence such that m' was received in G and $VT_G(m) \prec VT_G(m')$; if no such m' exists, then m is placed at the end of the delivery sequence. So, messages are delivered in order of receipt, except when that would result in a violation of single group causality. If G is a static group of size two created for point-to-point communication, then messages in G can be delivered in the order they are received, and vector timestamps are not used in G; i.e., a message received in G is simply placed at the end of the delivery sequence for the destination.

We extend this protocol to provide the above multiple group guarantees by using the ***conservative protocol***. In this protocol, a multicast is defined to be ***stable*** if it has been received at all of its destination sites. A group G is ***active*** for a process p if p's site does not know of the stability of a multicast to G either sent by or delivered to p. The ***conservative multicast rule*** states that a process p may multicast to group G if and only if G is the only active group for p or p has no active groups. If p attempts to multicast when this condition 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_2 simply delays sending m_2 until it knows that m_1 has been received by s_4, and then s_4 delivers m_1 before m_3, because they were received in that order.

The proof that these protocols satisfy the first guarantee is given in [3]. We now argue that these protocols provide the second guarantee. Suppose that G and G' are different groups, and that $\tilde{m}$ is received at r's site s_r after m' was multicast, but delivered to r before m. By the conservative protocol, m was stable and thus received at s_r before m' was multicast. Therefore, $\tilde{m}$ must have been received at s_r after m. By the above protocol descriptions, $\tilde{m}$ could have been delivered before m only if (i) $\tilde{m}$ were multicast in G and $VT_G(\tilde{m}) \prec VT_G(m)$, or (ii) $\tilde{m}$ were multicast in another group $\hat{G}$, and for some $\hat{m}$ previously received in $\hat{G}$ and placed before m in the delivery sequence, $VT_{\hat{G}}(\tilde{m}) \prec VT_{\hat{G}}(\hat{m})$. Now, suppose for a contradiction that $\tilde{m}$ was actually sent after (or at the same time that) m' was multicast. Possibility (i) could not happen due to the correctness of the vector timestamp protocol for group G, and similarly (ii) could not happen if $\hat{G}$ were uncorrupt. So, $\hat{G}$ must be corrupt. But, assuming that $\hat{G}$ is really a point-to-point connection, vector timestamps in $\hat{G}$ are not used, and (ii) could not happen. Thus, $\tilde{m}$ must have been sent before m'.

5 Delegation and access control

As stated before, the work of section 4 presupposes uncorrupt groups (except where otherwise stated). In light of this, it is obvious that in real systems, access to groups must be restricted. In the "secure island of sites" model discussed in section 1, access control is obviously needed to prevent a process on an untrusted site from joining a group. Access control is also needed ***within*** the secured island, because while the sites in that island are assumed to be secure, the processes they execute should not necessarily be trusted to join any group whatsoever.

In this section we propose an access control scheme based upon access control lists (ACLs) that we plan to employ in our system. We have chosen ACLs over classic capabilities to avoid several well-known shortcomings of capabilities, such as difficulties involved in revoking access rights, the inability to specify denial of access rights, and the need to confine capability propagation. And, while in some settings these problems are offset by the relative efficiency of capabilities, we have found that in majority of Isis applications, group joins are infrequent in comparison to other group operations (e.g., multicasts). Thus, we expect that the economy of capabilities would impact overall system performance only minimally. The scheme we propose here 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 (e.g., [12]).

The straightforward criteria on which to restrict access to groups are 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 that indicates the processes that would be allowed to join the group. While this may suffice for many applications, there may be some for which this is insufficient. Consider an extension of the OTC example of section 1 in which a client group authorizes the trading service to purchase certain stocks with funds in the client's bank account. Suppose that the service must then send a representative process to a group established by the client's bank to arrange the fund transfer for the stock purchase.[4] But, the bank will admit this process to the group only if the process has been duly authorized by one of the bank's clients. In this case, determining whether to admit the process based upon its owner (e.g., an individual broker) and hosting site is insufficient for two reasons: this information neither convinces the bank group that the process represents the trading service nor conveys the authorization to access the client's account.

This flavor of authorization is related to many concepts that have appeared in the literature in recent years, including ***authentication forwarding*** [26], ***cascaded authentication*** [25], and ***delegation*** [11]. Informally, each of these terms connotes the means by which one party confers access rights to another, as exemplified by the client delegating authority to the trading service in the previous example. Delegation in a group oriented system is different from that in other systems only in that groups are delegating and being delegated. In practical terms, this means that groups need to be authenticated. Fortunately, we already have the mechanisms to do this, namely group keys and the name service introduced in section 4.2.

The approach we take to delegation is best illustrated by an example. Suppose that group G_1 wishes to delegate some authority to group G_2. To do so, some member of G_1 causes the *credential*

$$G_1, T_1, G_2, \mathcal{S}_1(G_1, T_1, G_2) \qquad (1)$$

to be produced, where "T_1" is the time at which this credential expires, "$\mathcal{S}_1$" denotes the signature function of G_1 (i.e., signature with the private key of G_1), and "G_1" and "G_2" identify groups G_1 and G_2, respectively, either by address or name. Intuitively, a member of G_2 can present (1) to another party to prove that G_1 has delegated some access rights to G_2 until time T_1; in general, the access granted to members of G_2 based on (1) is left to the discretion of individual object monitors. Any party can verify (1) with the address of G_1, which contains G_1's public key (see section 4.2). A process in G_2 can delegate further to group G_3 by forming

$$G_1, T_1, G_2, T_2, G_3, \mathcal{S}_2(\mathcal{S}_1(G_1, T_1, G_2), T_2, G_3). \qquad (2)$$

(In the notation of (2), we are assuming that the signature scheme allows any party possessing the public key of G_2 to invert $\mathcal{S}_2(\mathcal{S}_1(G_1, T_1, G_2), T_2, G_3)$ to obtain $\mathcal{S}_1(G_1, T_1, G_2), T_2, G_3$.) This credential should be considered valid until time $\min\{T_1, T_2\}$. Of course, G_3 could delegate yet further in a similar fashion, and in general, credentials could become arbitrarily long. This delegation scheme is similar to that in [25], and the reader is referred there for a general discussion of its features.

The choice of whether to delegate to a group name or a group address has subtle implications. If a delegation is made to a group address (e.g., if "G_2" is a group address in (1)), then only the group that possesses the private key corresponding to the public key in that group address can utilize that delegation. However, if a delegation is made to a group name, then any group that is registered under that name before the delegation expires can possibly exploit it. We expect that both types of delegation will be useful to applications. For instance, in the previous OTC example, the client might delegate to the *name* of the trading service, if it wants the "current" trading service, whatever group that may be, to perform the transaction for it. The trading service, however, may delegate to the *address* of a satellite group that will handle the transaction with the bank. By delegating to a group name, the delegator implicitly trusts the directories that are prefixes of the name to allow only "trusted" groups to be registered under the name to which it delegated authority. Below we will describe a scheme that can be used to control access to directories.

The method by which a delegating group identifies itself is also important. This is true because the delegating group can restrict what access rights are transferred to the delegated group by identifying itself appropriately. This is known as delegating by *roles* and has been used in other delegation schemes (e.g., [11, 17]). Intuitively, associated with each role

[4] In practice, the representative would probably request to become an Isis *client* of the bank group. Isis would then invisibly create another group containing the client and the group members. The access control scheme presented in this section is particularly well-suited to restricting client access, although for simplicity we will explain it in terms of member access only.

of a group is some subset of the access rights that the group is allowed to delegate to others. When a group delegates the authority of a role, the authority transferred to the delegated group is at most that of the role, and not of the delegating group. So, in the OTC example, the client group could delegate authority to the trading service under a role that was used to establish the bank account and that would be useless, e.g., for reading the client's mail. As another example, credential (1) could represent a *referral* made by a bank group G_1 for a client G_2, which is required before the bank's loan service will negotiate with the client. In this case, G_1 would probably delegate by a role with which *no* access rights of G_1, per se, are associated, but that nevertheless indicates the needed referral. In our setting, a role corresponds to just another group name or group address. That is, a group can create roles for itself by registering other names for the group with the name service or by "duplicating" the group to obtain several group addresses (i.e., public keys).

This delegation scheme extends easily to an access control scheme for group membership as follows. Replicated at each site in a group is a set of ***delegation templates*** in addition to the set of ⟨owner, site⟩ pairs previously described. Each delegation template is a list $\mathcal{G}_1, \ldots, \mathcal{G}_n$ where each $\mathcal{G}_i$ is either a group name or address. (This does not imply that a group must know in advance the addresses or names of all groups that should appear in its delegation templates. The method of specifying these templates could employ, e.g., wild cards.) A credential

$$G_1, T_1, \ldots, G_{m-1}, T_{m-1}, G_m, S_{m-1}(\ldots) \qquad (3)$$

is said to ***match*** the delegation template $\mathcal{G}_1, \ldots, \mathcal{G}_n$ if $m \geq n$ and for all j satisfying $1 \leq j \leq n$, $G_{m-n+j} = \mathcal{G}_j$. (Here, two addresses are equal if they contain the same public key.) That is, the credential in (3) matches the template $\mathcal{G}_1, \ldots, \mathcal{G}_n$ if the credential ends with a sequence of delegations beginning with $\mathcal{G}_1$, followed by $\mathcal{G}_2$, and so on, and ending with $\mathcal{G}_n$. Intuitively, the credentials that match some delegation template of a group are those that the group accepts as legitimate patterns of delegation.

Given sets of delegation templates and ⟨owner, site⟩ pairs, access to a group is controlled as follows. If the group contact receives credential (3) embedded in a request from some process p to join the group, p is allowed to join if and only if all of the following hold.

- The authenticity of credential (3) can be verified with the appropriate group addresses (i.e., public keys).
- p's site can vouch that p is in G_m (by illustrating knowledge of the private key for G_m).
- Message (3) matches a delegation template for the group.
- None of the delegations in message (3) have expired.
- The ⟨owner, site⟩ pair of p is listed in the set of ⟨owner, site⟩ pairs for the group.

The group contact obtains any group addresses it needs to check these conditions from the name service (or from the credential itself).

We are currently considering extensions to this scheme to enable the group being joined to automatically limit the duration for which an admitted process is allowed to remain a member. These limitations could be based upon the credential used by the process to gain admittance and could vary depending on the meaning assigned to the credential by the application. For instance, depending on the meaning of (3), in the above scenario p could be allowed to remain a member (i) indefinitely, (ii) until time $\min\{T_1, \ldots, T_{m-1}\}$ is reached, (iii) until p is removed from G_m, or (iv) until either (ii) or (iii) occurs. In cases (ii), (iii), or (iv), p's site could remove p from the group when the appropriate condition occurs. Which of (i)-(iv) is chosen for a particular joining process could be determined on the basis of which delegation template and ⟨owner, site⟩ pair were matched to be admitted. Other extensions to this access control scheme are being considered but will not be discussed here.

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. A request to perform an operation on a directory would then be allowed subject to the above five conditions on the requesting process and the credential it supplies.

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, secure methods for joining groups, and protocols that provide certain causal delivery ordering guarantees. In addition, we have proposed an access control scheme based upon delegation for use in group oriented settings.

Implementation of the basic mechanisms has begun at Cornell University, in conjunction with the reimplementation of Isis. This implementation should provide valuable insight into the efficiency of our architecture and mechanisms, and it is also forcing us to consider user interface issues. In addition, we are in the process of developing the necessary information exchange protocols.

Future work on this system is heading in several directions. We are considering several different extensions to the design of the system. In particular, we are currently considering ways to employ the basic architecture described here at multiple security levels, as would be appropriate if islands of sites were secured at different levels. However, we are still unclear as to the consequences of such a design. We are also considering techniques to recover, e.g., after an intruder has infiltrated a group. Finally, we are exploring techniques for building secure, fault tolerant applications, given the secured abstractions provided by this architecture.

Acknowledgements

We are deeply appreciative of the distributed systems group at Cornell University for contributing many interesting discussions regarding this research. In particular, we thank Navin Budhiraja, Tushar Chandra, Robert Cooper, Brad Glade, Cliff Krumvieda, Keith Marzullo, Aleta Ricciardi, Patrick Stephenson, Sam Toueg, and Robbert van Renesse. Richard Platek and Raphael Yahalom also provided helpful discussions. Mark Steiglitz developed a mechanism for authenticating user identifiers under Isis; this work did not become a permanent part of the system but was a useful source of insight. Finally, we thank the anonymous referees for many useful comments.

References

[1] Birman, K. P., Cooper, R., and Gleeson, B. Design alternatives for process group membership and multicast. Tech. Rep. 91-1257, Department of Computer Science, Cornell University, Dec. 1991.

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

[3] Birman, K. P., Schiper, A., and Stephenson, P. Lightweight causal and atomic group multicast. *ACM Transactions on Computing Systems 9*, 3 (Aug. 1991), 272–314.

[4] Birrell, A. D. Secure communication using remote procedure calls. *ACM Transactions on Computing Systems 3*, 1 (Feb. 1985), 1–14.

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

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

[7] Desmedt, Y. Society and group oriented cryptography: A new concept. In *Proceedings of CRYPTO '87* (Aug. 1987), pp. 120–127. Published as *Lecture Notes in Computer Science 293*.

[8] Desmedt, Y., Frankel, Y., and Yung, M. Multi-receiver/multi-sender network security: Efficient authenticated multicast/feedback. In *Proceedings of IEEE INFOCOM* (May 1992).

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

[10] Frankel, Y. A practical protocol for large group oriented networks. In *Proceedings of EUROCRYPT '89* (Apr. 1989), pp. 56–61. Published as *Lecture Notes in Computer Science 434*.

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

[12] Gong, L. A secure identity-based capability system. In *Proceedings of the IEEE Symposium on Research in Security and Privacy* (May 1989), pp. 56–63.

[13] KAASHOEK, F. M., AND TANENBAUM, A. S. Group communication in the Amoeba distributed operating system. In *Proceedings of the IEEE International Conference on Distributed Computing Systems* (May 1991), pp. 222-230.

[14] LADIN, R., LISKOV, B., AND SHRIRA, L. Lazy replication: Exploiting the semantics of distributed services. In *Proceedings of the ACM Symposium on Principles of Distributed Computing* (Aug. 1990), pp. 43-57.

[15] LAIH, C. S., AND HARN, L. Generalized threshold cryptosystems. In *Proceedings of ASIACRYPT '91* (Nov. 1991).

[16] LAMPORT, L. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM 21*, 7 (July 1978), 558-565.

[17] LAMPSON, B., ABADI, M., BURROWS, M., AND WOBBER, E. Authentication in distributed systems: Theory and practice. In *Proceedings of the ACM Symposium on Operating Systems Principles* (Oct. 1991), pp. 165-182. Published as *ACM Operating Systems Review 25*, 5.

[18] OUSTERHOUT, J. K., SCELZA, D. A., AND SINDHU, P. S. Medusa: An experiment in distributed operating system structure. *Communications of the ACM 23*, 2 (Feb. 1980), 92-105.

[19] PETERSON, L. L., BUCHHOLZ, N. C., AND SCHLICHTING, R. D. Preserving and using context information in interprocess communication. *ACM Transactions on Computing Systems 7*, 3 (Aug. 1989), 217-246.

[20] REITER, M. K., AND BIRMAN, K. P. How to securely replicate services. Submitted for publication, Jan. 1991.

[21] RICCIARDI, A. M., AND BIRMAN, K. P. Using process groups to implement failure detection in asynchronous environments. In *Proceedings of the ACM Symposium on Principles of Distributed Computing* (Aug. 1991), pp. 341-351.

[22] RIVEST, R. L. Cryptography. In *Handbook of Theoretical Computer Science, Volume A, Algorithms and Complexity*, J. van Leeuwen, Ed. Elsevier Science Publishers B. V., 1990, ch. 13, pp. 717-755.

[23] ROZIER, M., ET AL. Overview of the Chorus distributed operating systems. Tech. Rep. CS/TR-90-25, Chorus Systèmes, Apr. 1990.

[24] SATYANARAYANAN, M. Integrating security in a large distributed system. *ACM Transactions on Computing Systems 7*, 3 (Aug. 1989), 247-280.

[25] SOLLINS, K. R. Cascaded authentication. In *Proceedings of the IEEE Symposium on Research in Security and Privacy* (Apr. 1988), pp. 156-163.

[26] STEINER, J. G., NEUMAN, C., AND SCHILLER, J. I. Kerberos: An authentication service for open network systems. In *Proceedings of the USENIX Winter Conference* (Feb. 1988), pp. 191-202.

[27] TSUDIK, G. Message authentication with one-way hash functions. In *Proceedings of IEEE INFOCOM* (May 1992).

[28] TYGAR, J. D., AND YEE, B. S. Strongbox. In *Camelot and Avalon, A Distributed Transaction Facility*, J. L. Eppinger, L. B. Mummert, and A. Z. Spector, Eds. Morgan Kaufmann, San Mateo, California, 1991, ch. 24, pp. 381-400.

[29] VAN RENESSE, R., BIRMAN, K., GLADE, B., AND STEPHENSON, P. Reliable multicast between micro-kernels. In *Proceedings of the USENIX Microkernels and Other Kernel Architectures Workshop* (Apr. 1992).

[30] VOYDOCK, V. L., AND KENT, S. T. Security mechanisms in high-level network protocols. *ACM Computing Surveys 15*, 2 (June 1983), 135-171.

A Causal multicast, continued

In this appendix, we describe a modification to the CBCAST protocol described in section 4.3 that provides the causal guarantees described there without requiring that communication with corrupt sites be only point-to-point. By removing this requirement, we provide somewhat stronger causal guarantees to, e.g., a process that finds itself in the intersection of uncorrupt and corrupt (non-point-to-point) groups. However, this protocol does not directly provide any other guarantees in a corrupt group, and so processes that require other guarantees should continue to communicate with potentially corrupt sites in a point-to-point fashion only. We assume that group memberships (of uncorrupt groups) are static, as before, and we treat a group as simply a set of processes. Again, the extensions to handle group view changes are precisely those in [3], although we omit them for the sake of brevity. The relations $\rightarrow$ and $\rightarrow_G$ are defined as in section 4.3.

The essential modification to the protocol of section 4.3 occurs in the placement of received messages in a destination's delivery sequence. When a message m is received at a site, it is simply placed at the end of the delivery sequence. Then, beginning with the earliest undelivered message in the sequence, the sequence is stepped through, and the following rule is applied in turn to each message m' (until m is reached): if m' was received in the same group G as m and $VT_G(m) \prec VT_G(m')$, then m' is moved to the back of the delivery sequence. The placement of m and this processing of the sequence are done atomically, in the sense that no new messages are delivered or added to the delivery sequence until both are complete. We claim that with no further modifications, this new protocol provides the two guarantees of section 4.3.

We begin by sketching the proof of the second guarantee. Suppose that G and G' are different groups, and that $\bar{m}$ is received at r's site s_r after m' was multicast. By the conservative protocol of [3], m was stable before m' was multicast, as was any $\hat{m}$ such that $\mathbf{mcast}_{\hat{q}}(\hat{m}, G) \rightarrow_G \mathbf{mcast}_p(m, G)$. Therefore, once m' was multicast, m was already received at s_r and could never again be moved back in the delivery sequence for r. Since $\bar{m}$ is received after m' is multicast, it follows that $\bar{m}$ must be delivered to r after m. So, the antecedent of the second guarantee cannot happen, and the condition is satisfied trivially.

We now sketch the proof for the first guarantee. If G and G' are different groups, then the second guarantee with $\bar{m} = m'$ implies the first. If G and G' are the same group and $\mathbf{mcast}_p(m, G) \rightarrow_G \mathbf{mcast}_q(m', G)$, then the first guarantee holds by the normal vector timestamp protocol and the new placement rule, because once m' is placed behind m in the delivery sequence for r, it will again be moved behind m each time m is moved to the end of the sequence. The last case is if G and G' are the same group and $\mathbf{mcast}_p(m, G) \not\rightarrow_G \mathbf{mcast}_q(m', G)$. In this case, there must be some $\hat{q}$, $\hat{m}$ and $\hat{G}$ different from G such that

$$\mathbf{mcast}_p(m, G) \rightarrow \mathbf{mcast}_{\hat{q}}(\hat{m}, \hat{G}) \rightarrow \mathbf{mcast}_q(m', G).$$

As in the argument above for the second guarantee, once $\hat{m}$ is multicast, m is received at s_r and can never again be moved back in the delivery sequence for r. And, since m' can be received at s_r only after $\hat{m}$ is multicast, m is delivered to r before m'.