How to Securely Replicate Services

MICHAEL K. REITER
AT & T Bell Laboratories
and
KENNETH P. BIRMAN
Cornell University

We present a method for constructing replicated services that retain their availability and integrity despite several servers and clients being corrupted by an intruder, in addition to others failing benignly. We also address the issue of maintaining a causal order among client requests. We illustrate a security breach resulting from an intruder's ability to effect a violation of causality in the sequence of requests processed by the service and propose an approach to counter this attack. An important and novel feature of our techniques is that the client need not be able to identify or authenticate even a single server. Instead, the client is required to possess only a single public key for the service We demonstrate the performance of our techniques with a service we have implemented using one of our protocols.

Categories and Subject Descriptors: C.2.0 [**Computer-Communication Networks**]: General—*security and protection*; C.2.4 [**Computer-Communication Networks**]: Distributed Systems; D.4.5 [**Operating Systems**]: Reliability—*fault-tolerance*; D.4.6 [**Operating Systems**]: Security and Protection—*authentication*; *cryptographic controls*; K.6.5 [**Management of Computing and Information Systems**]: Security and Protection—*authentication*

General Terms: Reliability, Security

Additional Key Words and Phrases: Causality, replication, state machines, threshold cryptography

1. INTRODUCTION

Distributed systems are often structured in terms of *clients* and *services*. A service exports a set of commands, which clients invoke by issuing *requests* to the service. After executing a command, the service may return an appropri-

This work was supported by DARPA/NASA grant NAG2-593, DARPA/ONR grant N00014-92-J-1866, grants from GTE, IBM, and Siemens, Inc., and a National Science Foundation Graduate Fellowship Any conclusions or recommendations expressed in this document are those of the authors and do not necessarily reflect the views of the National Science Foundation or the Department of Defense.

Authors' addresses: M. K. Reiter, AT & T Bell Laboratories, Room 4F-637, 101 Crawfords Corner Road, Holmdel, NJ 07733-3030; email: reiter@research.att.com; K. P Birman, Department of Computer Science, Ithaca, NY 14853; email: ken@cs.cornell.edu.

ate *response* to the client that invoked the command. In the simplest case, the service is implemented by only one *server*. If this server is not sufficiently reliable, however, then the service must be replicated.

In hostile environments, replication can introduce security risks. In particular, it is often more difficult, or at least requires more resources, to protect many distributed servers from corruption by an intruder than it is to protect only a single server [Herlihy and Tygar 1988; Lampson et al. 1992; Turn and Habibi 1986]. To compensate for this, a replicated service could be designed to remain available and correct despite several servers being corrupted by an intruder (in addition to others failing benignly). One way to do this employs the *state machine approach* [Schneider 1990] to replicating the service, in which each server individually computes each response and sends it to the client. If the client authenticates the response from each server and accepts the response sent by a majority of servers, then it obtains the correct response if a majority of servers are correct.

This approach, however, requires more from clients—in ability, computation, and storage—than not replicating the service. First, each client must always know how many servers comprise the service and must be able to authenticate each of the servers individually. This may be difficult if the set of servers can change over time or if there is no trustworthy source from which the client can obtain the identities or authentication information of the servers. Second, if there are n servers, then each client must perform $O(n)$ additional cryptographic computations to authenticate replies than if the service were not replicated. Third, each client must possess a public key for each server or a secure channel to each server, with an n-fold cost in storage. Finally, in cases in which a client must forward the (digitally signed) replies of the servers to other parties, as is the case, e.g., in many cryptographic protocols (see the "push" technique of Lampson et al. [1992]), the client must store and forward $O(n)$ replies, instead of only one as if the service were not replicated.

In this article we propose a solution to these problems using a modification of state machine replication. In its full generality, our approach can be used to implement a service with n servers so that clients accept responses computed by correct servers, and no other responses, provided that $n > t + b$ where t is the maximum number of faulty servers, and b is the maximum number of these faulty servers that behave maliciously. A novel feature of our approach is that unlike "normal" state machine replication described above, each client possesses exactly one public key for the service (as opposed to one for each server) and can treat the service as a single object for the purposes of authentication. This enhances application modularity and significantly simplifies the service interface for clients, because each client is insulated from internal security policies of the service and details of what or how many servers comprise the service. We emphasize that the client need not be able to identify or authenticate a single server to authenticate the response of the service. Moreover, the client incurs no additional cryptographic computation or storage costs than if the service were not replicated.

Even if clients accept only responses computed by correct servers, clients may still accept improper responses if an intruder has caused the correct servers to process improper requests or to process requests in an incorrect order. In this article we also discuss this issue. We focus on an attack in which an intruder effects and exploits a violation of causality in the sequence of requests processed by the service. We describe a method to avoid this attack that again requires that $n > t + b$ and that each client possess only a single public key for the service.

We have used our techniques to implement an authentication service in a security architecture for fault-tolerant systems [Reiter 1993; Reiter et al. 1992]. In our security architecture, this authentication service securely and fault tolerantly supports the distribution of cryptographic keys for secure communication in open networks. In this article we use this service to illustrate how one of our protocols can perform in practice.

The remainder of this article is structured as follows. In Section 2 we give a brief overview of state machine replication; for more detail, the reader should see Schneider [1990]. In Section 3 we enumerate our assumptions about the system. In Section 4 we present a method of implementing services that provides the availability and integrity guarantees outlined above. In Section 5, we discuss the importance of maintaining causality among client requests and a method to counter an intruder's attempts to exploit violations of causality. We conclude and discuss future and related work in Section 6.

2. STATE MACHINE REPLICATION

A *state machine* consists of a set of *state variables* and exports a set of (possibly parameterized) *commands*. The state variables encode the state of the state machine, and the commands transform that state. Each command is implemented by a deterministic program and is executed atomically (i.e., indivisibly) with respect to other commands. A *client* of the state machine invokes a command by issuing a request to the state machine. Requests should be processed by a state machine in an order that is consistent with Lamport's [1978b] causality relation. That is, requests from the same client should be processed in the order they were issued, and if one request could have caused another from a different client, then a state machine should process the former before the latter. Processing each request results in some *response* (i.e., output), which we assume is returned to the client that issued the request. Responses of a state machine are completely determined by its initial state and the sequence of requests it processes.

State machine replication is a general method of implementing a replicated service by simultaneously employing many state machine servers and coordinating client interactions with them.[1] If all servers are initialized to the same state, and if all correct servers process the same sequence of requests, then

[1]While the state machine servers must satisfy the same specification, they need not be identical replicas of one another. In fact, it may be desired that they not be identical, to avoid a (possibly deliberate) design flaw affecting all of them Employing nonidentical replicas is similar to the use of *n-version programming* as applied in Joseph [1987].

all correct servers will give the same response to any given request. By properly combining the responses of the servers to form the response of the service, where "properly" depends on the number and type of failures being considered, the service can retain its availability and integrity despite some server failures.

3. THE SYSTEM MODEL

Our system consists of a set of *processes*, n of which are servers, and the remainder of which are clients. All processes communicate exclusively by passing messages over a network. We assume nothing about this network other than what is required to implement the communication protocols on which our schemes rely; we give specifications of these protocols later in this section.

A process is *correct* in a run of the system if it always satisfies its specification. A process may *fail* in an arbitrary (Byzantine) manner, limited only by the (conjectured) properties of the cryptosystems and signature schemes we employ. In order to capture the notions of a "benign" failure versus a purposeful corruption by an intruder, we partition the faulty processes into two sets: the *honest* processes and the *corrupt* processes. Formally the only property that this partitioning must have is that any process that ever suffers "truly Byzantine" failures—i.e., failures that cannot be classified as crash, omission, or timing failures [Cristian et al. 1985]—must be classified as *corrupt*. We assume that there exist known constants t and b, $b \leq t$, such that in any system run, at most t servers fail, and of these at most b are corrupt. We assume that $n > t + b$.

Although processes fail as a single unit, it is convenient to view each process as consisting of logically separate *modules* (see Figure 1). More precisely, each server consists of a *communication module* and an *application module*. The application module of a server is simply a state machine as described in Section 2. The communication module *delivers* a request m to that state machine by executing *deliver*(m). Requests are delivered strictly sequentially, i.e., *deliver* is not executed until all previous executions of *deliver* have completed. The state machine processes requests in the order they are delivered. The communication module also implements a primitive *respond*(c, m) by which the application module can send a response m to a client c. This response m is of the form $\langle c, m', m'' \rangle$, where this response resulted from processing the request, m' and m'' are the "contents" of the response. (Alternatively, the response could include only an identifier for the request, instead of the entire request.) For simplicity, we assume that each request contains information sufficient for each server to determine (to whatever degree of certainty the application requires) the client that issued it, and that the resulting response is sent to that client.

Servers' communication modules make use of network communication to coordinate request deliveries and to communicate responses to clients. One type of network communication is of particular interest in this article: we assume that the network supports an authenticated broadcast primitive

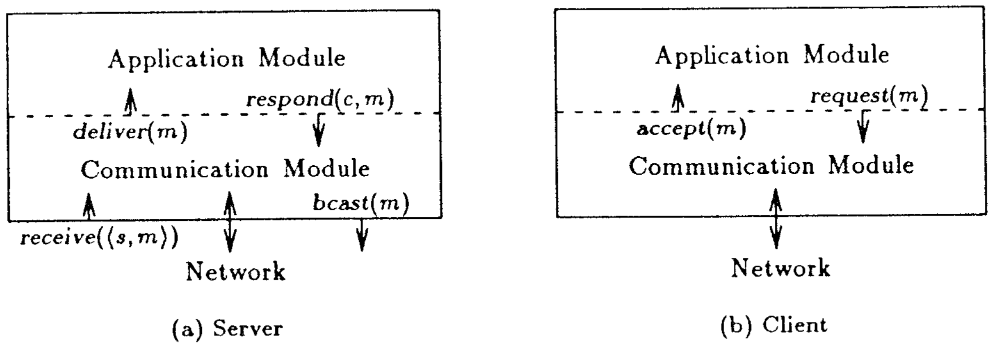

Fig. 1. Structure of processes.

bcast(m) by which the communication module of a server s can broadcast a message $\langle s, m \rangle$ to all servers. A server's communication module *receives* a message $\langle s, m \rangle$, with contents m and (apparently) from server s, by executing *receive*$(\langle s, m \rangle)$. Messages are received according to a protocol that satisfies the following specification.

—*Receipt Validity*: If a correct server s executes *bcast*(m), then *receive*$(\langle s, m \rangle)$ is executed at all correct servers.

—*Receipt Authentication*: If server s is correct or honest, then a correct or honest server executes *receive*$(\langle s, m \rangle)$ only if s previously executed *bcast*(m).

A client also consists of an application module and a communication module. The application module of a client is some client program that can issue a request m to the service by executing *request*(m). Processes' communication modules implement an *atomic broadcast* protocol by which requests are delivered to servers' application modules. This atomic broadcast protocol satisfies the following specification.

—*Delivery Validity*: If a correct client executes *request*(m), then *deliver*(m) is executed at all correct servers.

—*Delivery Order*: If *deliver*(m) is the ith execution of *deliver* at a correct or honest server, then *deliver*(m) is the ith execution of *deliver* at all correct servers. That is, all correct servers deliver the same sequence of requests, and the sequence of requests delivered by an honest server is a prefix of the sequence of requests delivered by a correct server.

Assuming that each server is initialized to the same state, Delivery Order implies that all correct and honest servers will produce the same response (or no response) to a given request. Options for implementing atomic broadcast are discussed in Section 6.1.1. Here we simply note that there already exist protocols in the literature that satisfy this specification in various models and for various definitions of *honest*. Our protocols do not rely on any bounds on message transmission times or the execution speeds of processes, and so the

Table I. Summary of Notation

Notation	Definition
system parameters	
n	total number of servers
t	maximum number of faulty servers in any system run
b	maximum number of corrupt servers in any system run (the corrupt servers are a subset of the faulty servers; so, $b \leq t$)
server routines	
deliver(m)	delivers client request m to application module
respond(c, m)	application module responds to c with m
receive$(\langle s, m\rangle)$	receives (authenticated) broadcast $\langle s, m\rangle$
bcast(m)	when executed at s, broadcasts $\langle s, m\rangle$ to servers
client routines	
accept(m)	accepts response m for application module
request(m)	issues request m to service

only such bounds required for our results, if any, are those required by the particular atomic broadcast protocol used.

A client's communication module accepts a response m for its application module by executing *accept*(m). Responses are accepted strictly sequentially; i.e., *accept* is not executed until all previous executions of *accept* have completed. The protocol by which responses are communicated to clients satisfies the following property.

—*Acceptance Validity*: If a correct server executes *respond*(c, m) and c is correct, then c executes *accept*(m).

4. PRESERVING INTEGRITY AND AVAILABILITY

Recall from Section 1 that our first goal is to ensure that each client accepts a response computed by a correct server, and no other responses, for each of its requests. We satisfy these requirements by replacing the *respond*(c, m) and *accept*(m) routines of servers and clients, respectively, with two new routines, *respond'*(c, m) and *accept'*(m), that will ensure this. Thus, the new structures of processes will be as pictured in Figure 2. While we have replaced the *respond* routine with *respond'* at the interface provided to the application module of each server, *respond* will be used in the implementation of *respond'* to send a response to a client. Similarly, *accept* will be used in the implementation of *accept'* to accept a message at a client. As desired, the *respond'* and *accept'* routines will ensure that the following hold.

—*Integrity*: A correct or honest client c executes *accept*(m) only if a correct server executes *respond'*(c, m).

—*Availability*: If a correct client c executes *request*(m), then c executes *accept*$(\langle c, m, m'\rangle)$ for some m'.

4.1 Threshold Signature Schemes

The *respond'* routine at the servers will employ a (k, n)*-threshold signature scheme*. A (k, n)-threshold signature scheme is, informally, a method of

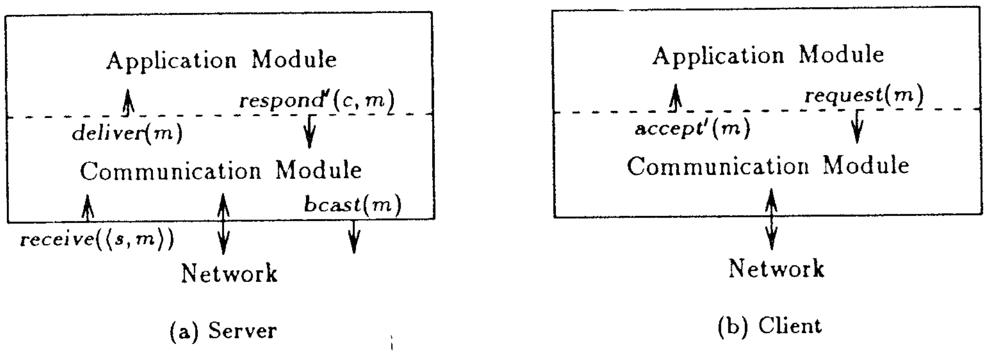

Fig. 2. New structure of processes.

generating a public key and n *shares* of the corresponding private key in such a way that for any message m, each share can be used to produce a *partial result* from m, where any k of these partial results can be combined into a signature for m that can be verified with the public key. Moreover, knowledge of k shares should be necessary to sign m, in the sense that without the private key it should be computationally infeasible to (i) create a signature for m without k partial results for m, (ii) compute a partial result for m without the corresponding share, or (iii) compute a share or the private key without k other shares.

Crytanalytic attacks on threshold signature schemes differ from those against conventional signature schemes in that the cryptanalyst may possess some number of shares and be able to acquire partial results, in addition to message/signature pairs. For our purposes, we will say, informally, that a (k,n)-threshold signature scheme is *secure* if it satisfies the following properties.

(1) Possession of only $k - 1$ or fewer shares and of partial results for various messages does not facilitate signing new messages. That is, if a possessor of such information can sign a new message, then it could also sign that message without knowledge of any shares or partial results for any messages. This property, which is formalized in Frankel and Desmedt [1992], says that the threshold signature scheme is as secure as the conventional signature scheme on which it is built.

(2) The conventional signature scheme on which the threshold signature scheme is built prevents *selective forgery* under *known-signature* attacks. That is, a cryptanalyst cannot manage to sign messages of its choice even though it can see signatures for various other messages (not of its choice).

Our *respond'* routine is not dependent on any particular implementation of a (k, n)-threshold signature scheme. For concreteness, however, we describe *respond'* in terms of an implementation proposed in Frankel and Desmedt [1992] (which is a slight variation of one proposed in Desmedt and Frankel [1992]). That implementation begins with an RSA [Rivest et al. 1978] public key (e, N) and private key d, where N is the product of two safe primes, and the Carmichael function λ is used in place of Euler's totient function ϕ to create e from d. That is, $ed \equiv 1 \bmod \lambda(N)$, where $\lambda(N)$ is the smallest positive integer such that $x^{\lambda(N)} \equiv 1 \bmod N$ for all $x \in \mathbf{Z}_N^*$. The n shares $\{K_i\}_{1 \le i \le n}$ are generated from d in such a way that for any set $T \subseteq \{1, \ldots, n\}$ of size k, $\sum_{i \in T}(K_i \cdot p_{i,T}) \equiv d - 1 \bmod \lambda(N)$, where the integers $\{p_{i,T}\}_{i \in T}$ are fixed a priori and public. (Each of the integers $p_{i,T}$ for all i and T are computed from a fixed set of n integers, each of binary length $O(\log n)$, with $O(n)$ integer multiplications and additions.) By defining the ith partial result for a message m to be $a_{m,i} \equiv m^{K_i} \bmod N$, it follows that for any $T \subseteq \{1, \ldots, n\}$ of size k, $A_{m,T} \equiv m \cdot \prod_{i \in T}(a_{m,i})^{p_{i,T}} \bmod N$ is the proper RSA signature for m, i.e., $m^d \bmod N$. Variations of this scheme have been proved to be as secure as RSA, in the sense of property 1 above [Frankel and Desmedt 1992].

For reasons of security and efficiency, it is often preferable to sign a *message digest* of a message, as opposed to the message itself [Denning 1984]. A message digest function f has the properties that the message digest $f(m)$ for any input m can be computed efficiently, but it is computationally infeasible to produce two distinct inputs m and m' such that $f(m) = f(m')$ or to produce any input m such that $f(m) = D$ for some prespecified message digest D. Several efficient implementations of message digest functions have been proposed (e.g., Rivest [1991]). We henceforth use f to denote such a function.

4.2 The Protocol

Suppose that a public key (e, N) and corresponding shares $\{K_i\}_{1 \le i \le n}$ are created as above, with the threshold parameter $k = b + 1$, and distributed so that for all i, $1 \le i \le n$, the ith server s_i is the sole possessor of K_i and any process can reliably obtain (e, N), the public key of the service. We do not discuss methods for distributing this information, although we note that all public-key systems require similar steps. The (information for computing the) integers $p_{i,T}$ for all i and T can be "hardwired" into the servers. Then, the *respond'*(c, m) and *accept'*(m) routines are implemented as follows.

Routine *respond'*(c, m) at server s_i:

(1) Execute *bcast*$(\langle f(m), a_{f(m),i}\rangle)$, where $a_{f(m),i} \equiv (f(m))^{K_i} \bmod N$ is s_i's partial result for $f(m)$.

(2) Wait until a set of messages $\{\langle s_j, \langle f(m), \alpha_j\rangle\rangle\}_{j \in T}$, $|T| = k$, has been received such that $A_{f(m),T} \equiv f(m) \cdot \prod_{j \in T}(\alpha_j)^{p_{j,T}} \bmod N$ is a valid signature for $f(m)$ (i.e., such that $(A_{f(m),T})^e \equiv f(m) \bmod N$).[2]

[2]Here we assume that messages can be received during the execution of *respond'*. This implies, for example, that locks required to receive messages must not be kept by threads waiting in step 2 of *respond'*. More generally, we assume that the replacement of *respond* and *accept* with *respond'* and *accept'* does not result in the violation of Receipt Validity or Authentication, Delivery Validity or Order, or Acceptance Validity (with *accept* replaced by *accept'*).

(3) Execute *respond*$(c, \langle m, A_{f(m),T} \rangle)$.

Routine *accept'*(m) at client c:

(1) If m is not of the form $\langle\langle c, m', m'' \rangle, S \rangle$, then return to the calling routine.
(2) If *accept*$(\langle c, m', m''' \rangle)$ for some m''' was previously executed at c, then return to the calling routine.
(3) If S is a valid signature for $f(\langle c, m', m'' \rangle)$ (i.e., if $S^e \equiv f(\langle c, m', m'' \rangle) \bmod N$), then execute *accept*$(\langle c, m', m'' \rangle)$ and return to the calling routine only after it has completed.

THEOREM 1. *If the threshold signature scheme is secure, then this protocol satisfies Integrity.*

PROOF. Suppose that the threshold signature scheme is secure. Then, because at most $b < k$ servers are corrupt and because each correct or honest server produces partial results only for message digests of responses that it computes, corrupt servers can generate signatures only for message digests of responses computed by correct servers (and possibly for other, presumably useless, message digest values). So, the corrupt servers cannot generate an improper response that a client will accept, and any response that is accepted by a correct or honest client must have been computed by a correct server. □

THEOREM 2. *This protocol satisfies Availability.*

PROOF. Because $n > t + b$, at least $b + 1 = k$ correct servers broadcast their partial results for (the message digest of) each response. Thus, by Receipt Validity, each correct server eventually receives k correct partial results for each of its responses, and so each correct server can correctly sign each of its responses. Since each request issued by a correct client c is delivered at all correct servers (by Delivery Validity), and since an execution of *respond*(c, m) at a correct server results in an execution of *accept'*(m) at c (by Acceptance Validity with *accept* replaced by *accept'*), it follows that a correct client will accept a response for each of its requests. □

4.3 Discussion

In theory, the most computationally expensive part of this protocol is step 2 of the *respond'* routine, in which the server sorts through the partial results it receives until it finds a T of size k such that $A_{f(m),T}$ is a valid signature. The server must examine at most the first $k + b = 2b + 1$ partial results received (from $k + b$ unique servers), and at most $\binom{k+b}{k}$ subsets of partial results, because in $k + b$ partial results are at least k correct ones. While this search is exponential in b in the worst case and could be costly if b is large and the actual number of corrupt servers is close to b, in most systems n (and thus b) and the actual number of corrupt servers will typically be small. (For example, our experience with the Isis system [Birman et al. 1991] suggests that fault-tolerant services are sometimes implemented by three to five servers, but rarely more.) Thus, while we are pursuing optimizations to make this search less costly as a function of b (see Section 6.1.2), we view them to be primarily of theoretical interest.

In terms of communication complexity, in a failure-free run the replacement of *respond* with *respond'* results in n executions of *bcast*, which can be executed concurrently. Therefore, the entire protocol that begins when a client issues a request and ends when it accepts a response consists of three communication "phases" that must be executed roughly sequentially: the request by the client, the dissemination of partial results (n concurrent executions of *bcast*), and the sending of the responses (n concurrent executions of *respond*).

Because a client needs to obtain only one correctly signed response for each request for Availability to be satisfied, this communication could be optimized by having only $t + 1$ servers respond to any given request. Additionally, the partial results for the response would need to be broadcast only to that set of servers. The broadcast of partial results could be optimized for the common case in most systems (i.e., when there are no corrupt processes) by employing a protocol that satisfies only Receipt Validity. However, when there are corrupt processes, this would allow these processes to disseminate incorrect partial results on behalf of correct or honest servers, further complicating each server's search for partial results that form a correct signature.

In this protocol, each server must know $O(n \log n)$ bits of public information (in order to compute the $p_{i,T}$'s), in addition to the public key for the service and its share of the private key. Each client needs to know only the public key of the service and must verify only one (correctly signed) response per request. While additional correctly signed responses may arrive at the client, they are discarded in step 2 of *accept'* without performing any cryptographic computation on them.

In comparison to "normal" state machine replication, where clients authenticate each server individually, the primary practical weakness of our approach is that by trading client burden for server burden, we limit the ability of our services to scale to very large systems with many clients. Our approach was motivated primarily by the need for a highly secure and available service —the authentication service of Reiter [1993, Ch. 3]—where alternatives for dealing with issues of scale previously existed and would often be necessary anyway.[3] In this case, our approach has yielded substantial benefits by insulating clients from the implementation details and internal security policies of the service (e.g., the value of b), and by minimizing overhead on client processors and the latencies of client protocols that use the responses of the service. Our approach should be an attractive alternative for systems with similar requirements.

[3]Having one authentication service for a very large system can be administratively impractical, and there may not be a single authority trusted by everyone to protect it [Lampson et al. 1992]. Approaches using multiple authentication services (e.g., one per administrative domain) have been employed in systems to remedy these problems (e.g., Lampson, et al. [1992] and Steiner et al. [1988]).

4.4 Performance: An Example

As an example of how the protocol of Section 4.2 might perform in practice, in this section we discuss the performance of the public-key authentication service of Reiter [1993]. This service was a primary motivation for this work and has been implemented using the protocol of Section 4.2. It has been constructed as part of a larger effort to integrate mechanisms into Isis, a system for building fault-tolerant applications [Birman et al. 1991]. The security architecture we have developed is presented in Reiter [1993] and Reiter et al. [1992].

In our security architecture, the authentication service is the trusted authority as to which public key belongs to which *principal* (e.g., computer, user). To exercise this authority, the service produces public-key *certificates* for principals. Each certificate contains a principal's identifier and the public key for that principal (among other things), bound together with the signature of the authentication service. Because requests for certificates commute, clients issue requests to the service with an unordered multicast (i.e., Delivery Order is not enforced).

The performance of our prototype service is illustrated in part (a) of Figure 3. The line indicates the mean response time in seconds as a function of n, the number of servers. For each n, we assumed that $t = b = \lfloor (n-1)/2 \rfloor$. We have illustrated the curve for up to nine servers, although we expect that few services will be implemented by more than five, as mentioned in Section 4.3. In these tests the RSA modulus N was 512 bits. These tests were executed between user processes over SunOS 4.1.1 on moderately loaded 33 MHz Sparc ELC workstations. The workstations spanned two 10 Mbit/s ethernets connected by a gateway. Each data point is the mean of 40 consecutive trials.

The cost of performing exponentiation modulo N is the direct cause of the poor response times in part (a) of Figure 3 and, in general, is the limiting factor in the performance of our authentication service. Exponentiation modulo N is required to construct a partial result from a share and a message, and k exponentiations modulo N (but with exponents much smaller than N) are used to construct a signature from k partial results. Moreover, none of these modular exponentiations lend themselves to well-known optimizations using the Chinese Remainder Theorem (see Shand and Vuillemin [1993]), since a server cannot be allowed to know the factorization of N. Part (a) of Figure 3 reflects the cost of performing these modular exponentiations in software,[4] and we expect that hardware support for modular exponentiation will generally be required at the servers to achieve satisfactory performance with the protocol of Section 4.2. Even with most presently available hardware

[4]In these tests we used the C implementation of modular exponentiation provided with the RSAREF toolkit, licensed free of charge by RSA Data Security, Inc The RSAREF toolkit was developed to support privacy-enhanced electronic mail, not interprocess communication, and faster software implementations of modular exponentiation are available from RSA Data Security, Inc. and others. However, with any software implementation of which we are aware, the cost of modular exponentiation would continue to be the primary factor limiting performance in our service.

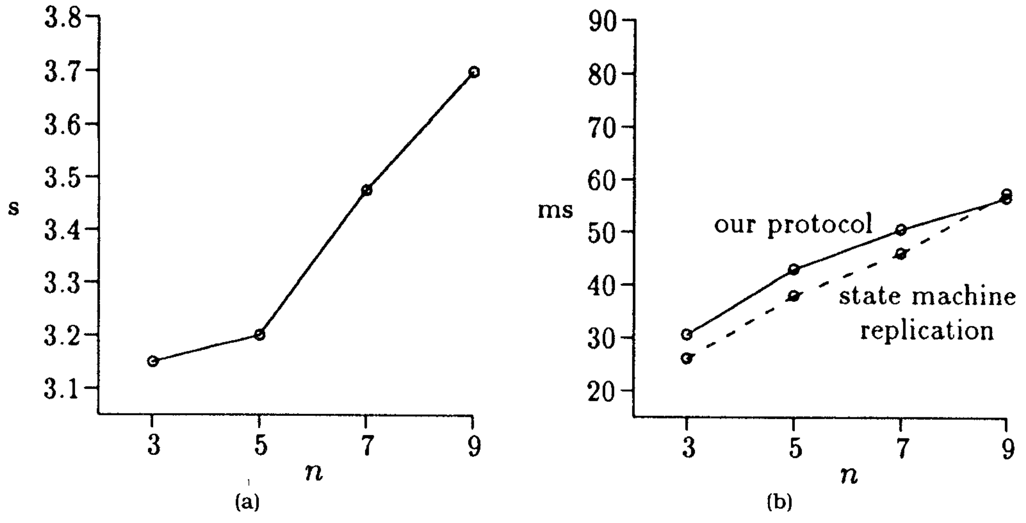

Fig. 3. Response times of an example service.

[Brickell 1990], the cost of modular exponentiation will continue to be the primary factor limiting performance in our protocol. However, advances that promise, e.g., modular exponentiation with a 512-bit exponent and modulus in a few milliseconds or less [Orup et al. 1991; Sedlack 1988; Shand and Vuillemin 1993], will make this less and less the case.

In our prototype implementation of the authentication service, we have not invested in hardware to support modular exponentiation at the servers. To compensate for its poor response time, we have designed our security architecture so that interactions with the authentication service always occur in the background, off the critical path of any other protocol or computation. Nevertheless, to gain insight into the potential performance of our service with such hardware, we have tested the performance of our prototype with the (software) modular exponentiation routines removed from the servers. This is obviously an optimistic simulation of hardware performance. However, in light of the recent advances in modular exponentiation hardware just mentioned, these tests could be indicative of the potential performance of our service.

The results of these tests are illustrated in part (b) of Figure 3. The solid line indicates the mean service response time in milliseconds as a function of n. For comparison purposes, we also modified our authentication service to simulate its performance if it used "normal" state machine replication, where each server individually computes, digitally signs, and sends its reply to the client, and the client authenticates k of these replies. (Again, all modular exponentiations were removed from the servers.) The performance of this approach is shown by the dashed line.

As illustrated in part (b) of Figure 3, state machine replication performed slightly better than our approach in most cases. This is due to the round of server communication in our approach to disseminate partial results. In the tests of Figure 3, partial results were communicated between servers using authenticated point-to-point channels; the cryptographic mechanisms used for authentication were implemented entirely in software. Given our assumption of hardware for modular exponentiation, this dissemination could possibly be optimized by exploiting this hardware. For instance, each server could be initialized with its own RSA private key and the public keys of the other servers, and partial results could be authenticated using RSA digital signatures. Moreover, this opens the possibility of using hardware multicast to disseminate partial results, since the same digitally signed partial result could be multicast to all servers at once.

While the increase in response times as a function of n in our approach stems from server communication, the increase for state machine replication results from the client having to verify multiple signatures. In these tests, all public RSA exponents were set to three (i.e., $e = 3$) for maximum efficiency in verifying signatures. With the RSA software we employed, signature verification then cost approximately 10 ms per verification. This increased cost on the client processor gives us the opportunity to make an important point: while the response time of our service is slightly worse, the client processor is free for all but the time required to make a request and to verify one signature. In settings in which client processors tend to be heavily loaded and in which equipping all clients with modular exponentiation hardware is prohibitively expensive, this could be an important feature.

It is risky to conclude too much from part (b) of Figure 3. We reiterate that the single most important factor in the performance of both our approach and normal state machine replication, namely, the speed of modular exponentiation at the servers, was removed from these tests. Moreover, there are several other factors—including the optimizations on the dissemination of partial results described above and the existence of faster software implementations of signature verification with $e = 3$ (which will cause the dashed curve to flatten somewhat)—that could impact these tests. Additionally, the service used in these tests is atypical in that it does not require that client requests be delivered by an atomic broadcast protocol. (However, since atomic broadcast is generally a requirement of both our approach and normal state machine replication, the impact on both should be the same.)

Nevertheless, part (b) of Figure 3 does indicate that with efficient hardware support for modular exponentiation at the servers, the protocol of Section 4.2 may provide acceptable response times for many services. This, combined with the additional features of our scheme (e.g., insulating clients from the service implementation and removing burden from client processors), supports the hypothesis that our approach can be a reasonable alternative to normal state machine replication in some situations.

5. PRESERVING INPUT CAUSALITY

One guarantee provided in the previous section is that any response accepted at a correct or honest client was computed by a correct server. Even the output of a correct server, though, may not reflect the way things "should be" if an intruder has caused the service to process improper requests or to process requests in an incorrect order. In general, ensuring proper responses from a correct server requires mechanisms to authenticate client requests and to enforce *access controls* on what state variables clients can write, because responses computed from state variables that can be written (directly or indirectly) by corrupt clients cannot be trusted. Approaches for authenticating client requests and enforcing access controls are well known (e.g., Lampson et al. [1992]) and will not be discussed further here.

In this section we address the issue of ensuring that requests are delivered in a correct order by correct and honest servers. Because we assume an atomic broadcast protocol to disseminate client requests, we concern ourselves only with the requirement that correct servers deliver requests in an order consistent with causality (see Section 2). A common method of attempting to preserve a causal order among client requests is for each client to refrain from communicating between the time it issues a request to the service and the time at which the request is delivered at some honest or correct server [Schneider 1990]. While this suffices to ensure that requests from the same client are delivered in the order issued, this does not suffice to ensure a causal delivery order for all requests. In particular, consider the case in which a correct client issues a request to the service, and after seeing the request, a corrupt server sends a message to a corrupt client. If the corrupt client subsequently issues a request, then there is a causal relationship between the two requests. However, it is not clear how this relationship can be detected by correct servers.

To illustrate why this may be important, we borrow an example from Reiter et al. [1992]: suppose that the service is a trading service that trades stocks, and that a client issues a request to purchase shares of stock through this service. After discovering the intended purchase, a corrupt server could collude with a corrupt client as described above to issue a request for the same stock to the service. If the correct servers deliver this request before that of the correct client, this request may adjust the apparent demand for the stock and raise the price offered to the correct client. Thus, by allowing the causally subsequent request of the corrupt client to be delivered before the request of the correct client, a type of "insider trading" may occur. Moreover, access controls alone cannot naturally avoid this problem, since the intent is that any client can request to purchase stock at any time.

In the rest of this section, we present new routines *request'*(m) and *deliver'*(m) that replace *request*(m) and *deliver*(m), respectively. Thus, if these are used with the *respond'* and *accept'* routines of Section 4.2, processes would be structured as in Figure 4. These new routines protect clients from the type of attack described above, in the sense that any request based on information obtained from a correct or honest client's request m can be delivered at correct or honest servers only after m. As before, we will use *deliver* to deliver a request in our implementation of *deliver'*, and we will use *request* to issue a request in *request'*.

In the implementation of *request'*(m), the client c encrypts m under the public key of the service before issuing the request to the service. Then, c is

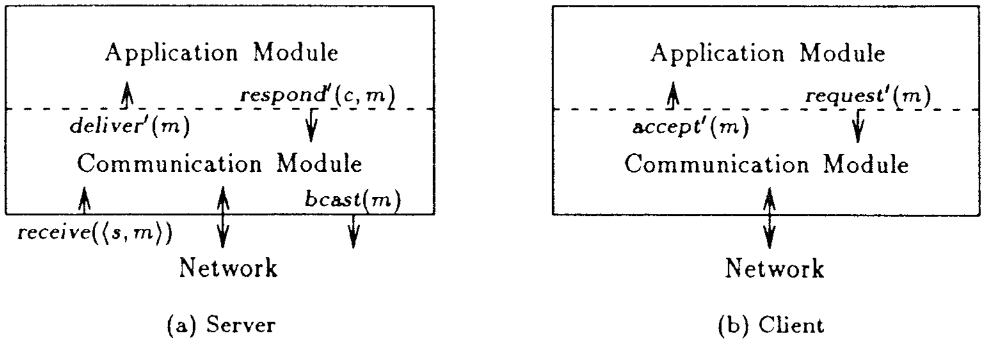

Fig. 4. New structure of processes.

provided the following guarantee if it is correct or honest. The reader should verify that this guarantee prevents the aforementioned problem.[5]

—*Causality*: If *deliver*(m') is executed at a correct or honest server s when *deliver*(m) has not yet been executed at s, then m' was issued before m was decrypted anywhere (other than c).

To guarantee that servers' states remain consistent, *request′* and *deliver′* must also ensure that client requests continue to be delivered according to the specification of atomic broadcast—i.e., that Delivery Validity (with *request* replaced by *request′*) and Delivery Order still hold.

5.1 Threshold Cryptosystems

Our *deliver′* routine at the servers will employ a (k, n)*-threshold cryptosystem*. A (k, n)-threshold cryptosystem is, informally, a method of generating a public key and n *shares* of the corresponding private key in such a way that for any message m encrypted under the public key, each share can be used to produce a *partial result* from the ciphertext of m, where any k of these partial results can be combined to decrypt m. Moreover, knowledge of k shares should be necessary to decrypt m, in the sense that without the private key it should be computationally infeasible to (i) decrypt m without k partial results for m, (ii) compute a partial result for m without the corresponding share, or (iii) compute a share or the private key without k other shares.

As with threshold signature schemes, cryptanalytic attacks against threshold cryptosystems differ from those against conventional public-key cryptosystems in that they may involve the use of partial results and some number of shares, in addition to plaintext/ciphertext pairs. For our purposes,

[5]We do not consider *traffic analysis* attacks [Voydock and Kent 1983] or attacks that exploit the *malleability* of the cryptosystem [Dolev et al. 1991].

we will say, informally, that a (k, n)-threshold cryptosystem is *secure* if it satisfies the following properties.

(1) Possession of only $k - 1$ or fewer shares and of partial results for various ciphertexts does not facilitate decrypting new ciphertexts. That is, if a possessor of such information can decrypt a new ciphertext, then it could also decrypt that ciphertext without knowledge of any shares or partial results for any ciphertexts. This property, which is formalized in Frankel and Desmedt [1992], says that the threshold cryptosystem is as secure as the conventional cryptosystem on which it is built.

(2) The conventional cryptosystem on which the threshold cryptosystem is built is resistant to *known-plaintext* attacks. That is, the cryptanalyst cannot manage to decrypt given ciphertexts even though it can see the plaintext corresponding to various other ciphertexts (but *not* corresponding to ciphertexts *of its choice*, as would be possible in a *chosen-ciphertext* attack).

As we will see in Section 5.2, we would actually prefer a threshold cryptosystem that is based on a conventional cryptosystem able to tolerate chosen ciphertext attacks. However, the above definition is in accordance with the security of all implementations of threshold cryptosystems thus far proposed, in the sense that all proposed implementations are based on conventional cryptosystems that are known to be vulnerable to chosen-ciphertext attacks.

Because the acts of signing a message and decrypting a message are operationally identical in the RSA signature scheme and cryptosystem, one implementation of a (k, n)-threshold cryptosystem can be obtained directly from the (k, n)-threshold signature scheme described in Section 4.1. Messages would be encrypted under the public key (e, N) in the usual manner, and the ith partial result for an encrypted message $m \equiv (m')^e \bmod N$ would be defined precisely as in Section 4.1—i.e., $a_{m,i} \equiv m^{K_i} \bmod N$. Then, $m' \equiv A_{m,T} \equiv m \cdot \prod_{i \in T} (a_{m,i})^{p_{i,T}} \bmod N$ for any T of size k. Other implementations of threshold cryptosystems have been proposed, based on both the RSA and ElGamal [1985] cryptosystems [Desmedt and Frankel 1990; Laih and Harn 1991].

5.2 The Protocol

Suppose that we are using the RSA threshold cryptosystem described above and that we have the initial conditions assumed in Section 4.2: the ith server s_i is secretly given sole possession of K_i; the cryptosystem threshold parameter $k = b + 1$; any process can reliably obtain the public key (e, N) of the service; and all servers know (a priori) $p_{i,T}$ for all i and T. The basic idea of our protocol is that each client encrypts each of its requests m with the public key of the service, in an attempt to force k servers to cooperate to decrypt it. Then, each correct or honest server refrains from broadcasting its partial result for the ciphertext of m until the delivery sequence through m is fixed locally. In this way, once a corrupt server collects k partial results for the ciphertext of m, the delivery sequence through m is fixed at some correct or honest server, and thus at all correct servers, and no requests can be placed before m in the delivery sequence at correct or honest servers.

This protocol preserves Causality only if each server requires k partial results for the ciphertext of m to decrypt that ciphertext. Even under the assumption that this cryptosystem is secure, however, this unfortunately is *not* the case with this or any implementation of a (k, n)-threshold cryptosystem proposed thus far. The problem is that our protocol as described above allows a corrupt server to mount chosen-ciphertext attacks, against which neither the RSA nor the ElGamal cryptosystem (nor any proposed threshold cryptosystem based on them) is resistant. In our setting, a corrupt server can see at any time how any ciphertext m of its choosing is decrypted, by simply requesting that a corrupt client issue m as an apparently legitimate request to the service. The corrupt server can then collect k partial results for m to see the plaintext to which m decrypts.

Methods of using chosen-ciphertext attacks against the RSA and ElGamal cryptosystems are well known. Here we present one method, originally due to Moore (see Denning [1984]), by which a corrupt server can decrypt the RSA ciphertext $m \equiv (m')^e \bmod N$ of a client's request m' without waiting to receive k partial results for m.

(1) The corrupt server chooses an arbitrary x and computes $y \equiv x^e \bmod N$; i.e., $x \equiv y^d \bmod N$.

(2) Via a corrupt client, the corrupt server issues the request $ym \bmod N$ to the service.

(3) The corrupt server collects k partial results for $ym \bmod N$ and forms $(ym)^d \bmod N$.

(4) The corrupt server computes $x^{-1} \equiv y^{-d} \bmod N$, and then

$$x^{-1}(ym)^d \equiv y^{-d}y^d m^d \equiv m^d \equiv (m')^{ed} \equiv m' \bmod N.$$

(NB: If x does not have an inverse mod N, then the corrupt server can factor N and break the cryptosystem, because $gcd(x, N)$ is a prime factor of N.)

Similar attacks are possible with the threshold cryptosystems described in Desmedt and Frankel [1990] and Laih and Harn [1991].

Ideally, we would like to find a threshold cryptosystem based on a conventional public-key cryptosystem that is tolerant of chosen-ciphertext attacks. To our knowledge, however, no such threshold cryptosystem has been proposed, and even conventional public-key cryptosystems that are tolerant of such attacks (e.g., Dolev et al. [1991]) are impractical. Therefore, such attacks must be *prevented.* A simple way to prevent these attacks is to authenticate client requests and have each client use a separate public encryption key for the service. This would prevent chosen-ciphertext attacks against the keys and ciphertexts of correct and honest clients, because a request from a corrupt client would be decrypted using the shares of the key for that client, and not a correct or honest one. However, this approach complicates key management, requires more storage at the servers to store a share per client, and requires that clients be authenticated.

An alternative method, which we adopt here, is to prevent chosen-ciphertext attacks using a cryptographic technique introduced in Lim and Lee [1994]. With this approach each server can determine (with a high probability), prior to creating its partial result for a request, whether the request was properly created from a plaintext. If not, then the server discards the request and does not create its partial result for the ciphertext.

In the technique of Lim and Lee [1994] as applied to RSA, the ciphertext for a message m consists of three parts. The first part m_1 is the RSA ciphertext of a secret, random seed q to a cryptographically strong pseudorandom bit generator G; i.e., $m_1 \equiv q^e \bmod N$ where e and N are the public RSA exponent and modulus, respectively. The $|m|$-bit output $G(|m|, q)$ of G with seed q, where $|m|$ is the bit length of m, is exclusive-ored with m to form the second part m_2. The third part m_3 is computed as $m_3 \equiv q^{f(m_1 \| m_2)} \bmod N$, where f is a message digest function (of a certain form), and $\|$ denotes concatenation. (See Section 4.1 for a description of message digests.) Under the assumption that inverting RSA encryption or f is not possible, the recipient of $\langle m_1, m_2, m_3 \rangle$ can verify that this ciphertext was properly constructed from a plaintext by checking that $(m_3)^e \equiv (m_1)^{f(m_1 \| m_2)} \bmod N$ [Lim and Lee 1994].

Then the *request'* and *deliver'* routines execute as follows.

Routine *request'*(m):

(1) Generate a new, random seed q, $0 < q < N$, and encrypt m as in Lim and Lee [1994], i.e.,

$$m_1 \equiv q^e \bmod N,$$
$$m_2 = G(|m|, q) \oplus m,$$
$$m_3 \equiv q^{f(m_1 \| m_2)} \bmod N,$$

where "$\oplus$" denotes bitwise exclusive-or.

(2) Execute *request*$(\langle m_1, m_2, m_3 \rangle)$.

Routine *deliver'*(m) at server s_i:

(1) If m is not of the form $\langle m_1, m_2, m_3 \rangle$, then return to the calling routine.

(2) If $(m_3)^e \not\equiv (m_1)^{f(m_1 \| m_2)} \bmod N$, then return to the calling routine (because this may be a chosen-ciphertext attack).

(3) If this is the hth execution of *deliver'*, then execute *bcast*$(\langle h, a_{m_1, i} \rangle)$, where $a_{m_1, i} \equiv (m_1)^{K_i} \bmod N$ is s_i's partial result for m_1.

(4) Wait until a set of messages $\{\langle s_j, \langle h, \alpha_j \rangle \rangle\}_{j \in T}$, $|T| = k$, has been received such that $A_{m_1, T} \equiv m_1 \cdot \prod_{j \in T} (\alpha_j)^{p_{j,T}} \bmod N$ is the correct decryption of m_1 (i.e., such that $(A_{m_1, T})^e \equiv m_1 \bmod N$).

(5) Execute *deliver*$(G(|m_2|, A_{m_1, T}) \oplus m_2)$ and return to the calling routine only after it has completed.

THEOREM 3. *This protocol satisfies Delivery Validity with request replaced by request′.*

PROOF. Suppose that *deliver′*(m) is the ith execution of *deliver′* at a correct or honest server, and suppose that each correct server completes its $(i-1)$th execution of *deliver′*. By Delivery Order with *deliver* replaced by *deliver′*, *deliver′*(m) is the ith execution of *deliver′* at all correct servers. Thus, all correct servers either return to the calling routine in step 1 or 2 of *deliver′*(m) or broadcast their partial results in step 3. Because $n > t + b$, in the latter case at least k correct partial results for m_1 are broadcast by and thus received at all correct servers, ensuring that each correct server will deliver a request and return to the calling routine. Thus, by induction it follows that each execution of *deliver′* at correct servers completes.

Delivery Validity with *deliver* replaced with *deliver′* then implies that if a correct client c executes *request′*(m), then *deliver′*$(\langle m_1, m_2, m_3\rangle)$ is executed at all correct servers, where $\langle m_1, m_2, m_3\rangle$ is as created in *request′*(m). Since any k partial results for m_1 from k correct and honest servers enables m_1 to be decrypted, m will be delivered. □

THEOREM 4. *This protocol satisfies Delivery Order.*

PROOF. By Delivery Order with *deliver* replaced by *deliver′*, if *deliver′*(m) is the ith execution of *deliver′* at a correct or honest server, then *deliver′*(m) is the ith execution of *deliver′* at all correct servers. Suppose a correct or honest server executes *deliver*(m) for some m in its ith execution of *deliver′*. Then, the ith execution of *deliver′* must be of the form *deliver′*$(\langle m_1, m_2, m_3\rangle)$. Consider this execution of *deliver′*$(\langle m_1, m_2, m_3\rangle)$ at a different correct or honest server s'. Because there is only one seed q, $0 < q < N$, such that $q^e \equiv m_1 \bmod N$, it is not possible for s' to execute *deliver*(m') for some $m' \neq m$. If s' is correct, then eventually k partial results from k correct and honest servers will be received at s', which will enable it to decrypt and deliver m. If s' is honest and does not deliver m, then *deliver′* $(\langle m_1, m_2, m_3\rangle)$ does not complete and s' never again executes *deliver′* (or *deliver*). □

THEOREM 5. *If the threshold cryptosystem is secure, then this protocol satisfies Causality.*

PROOF. Suppose that a correct or honest client c executes *request′*(m). If c makes progress for sufficiently long, then c executes *request*$(\langle m_1, m_2, m_3\rangle)$, where m_1, m_2, and m_3 are as created in *request′*(m). If the threshold cryptosystem is secure, then the earliest point at which m could be decrypted anywhere is sometime after some correct or honest server broadcasts its partial result for m_1, in *deliver′*$(\langle m_1, m_2, m_3\rangle)$. If no correct or honest server ever broadcasts its partial result for m_1, then Causality is trivially satisfied. Otherwise, let s be the first correct or honest server to broadcast its partial result for m_1.

Suppose that *deliver*(m') for some $m' \neq m$ is executed at a correct or honest server s' when *deliver*(m) has not yet been executed at s'. Then, *deliver*(m') must be executed at s' when *deliver′*$(\langle m_1, m_2, m_3\rangle)$ has not been executed at s', because otherwise *deliver*(m) would have been executed in *deliver*$'(\langle m_1, m_2, m_3\rangle)$ prior to *deliver*(m'). So, *deliver*(m') must be executed at s before *deliver*$'(\langle m_1, m_2, m_3\rangle)$. Because m_1 cannot be decrypted before s broadcasts its partial result for m_1 in *deliver*$'(\langle m_1, m_2, m_3\rangle)$, m' is delivered (and thus was issued) before m is decrypted anywhere. □

5.3 Discussion

In a failure-free run, the replacement of *deliver* with *deliver*$'$ results in an additional n executions of *bcast*, which are executed concurrently. Thus, when this protocol is used to deliver client requests, and the protocol of Section 4.2 is used to sign responses, a request results in one atomic broadcast, two sets of n concurrent executions of *bcast* to disseminate partial results, and the responses to the client. The storage requirements at the clients and servers in this protocol are the same as in the protocol of Section 4.2. The computation costs in this protocol, however, are slightly higher due to the mechanisms for preventing chosen-ciphertext attacks.

A more general discussion of the importance of detecting causal relationships in hostile environments can be found in Reiter and Gong [1993]. In the framework presented there, the attack addressed by the protocol of Section 5.2 is termed causal *denial*. That work also presents another class of attacks called causal *forgery*, although in the context of this article, preventing causal denial is sufficient to satisfy the causal requirements on the delivery of requests to the servers. While the protocol of Section 5.2 does not fully prevent denial in the sense of Reiter and Gong, we believe that the Causality guarantee should suffice for virtually all applications. The interested reader should consult Reiter and Gong [1993] for a more detailed description of these attacks and measures to prevent them in some situations.

6. CONCLUSION

We have presented a method for securely replicating services using a modification of the state machine approach. With our technique, a service can be replicated as n servers, where $n > t + b$, so that clients will accept responses computed by correct servers and will not accept other responses. We have also addressed the issue of maintaining a causal order among client requests. We illustrated a security breach resulting from an intruder's ability to violate causality in the sequence of requests processed by the service, and we presented an approach to counter this problem. An important feature of our methods is that they free the client of the responsibility of authenticating servers. This is achieved by employing two recent advances in cryptography, namely, threshold cryptosystems and threshold signature schemes. We have implemented a prototype authentication service using one of our protocols, and preliminary data indicate that our techniques can yield efficient service implementations if the servers are equipped with high-performance hardware for modular exponentiation.

6.1 Future Work

6.1.1 *Atomic Broadcast.* At the present time, one of the primary factors limiting the general usefulness of our techniques, and indeed of state machine replication, is that they rely on an atomic broadcast protocol (see Section 3). It is well known that it is impossible to find a deterministic solution to *consensus*, and thus atomic broadcast, in distributed systems that can suffer even a single crash failure [Fischer et al. 1985]. We believe that an important direction for future research is to find practical ways to circumvent this impossibility result in our system model.

One approach to circumventing this impossibility result has been to consider only *synchronous* systems, in which there are known bounds on message transmission times and execution speeds of processes. Deterministic atomic broadcast protocols that suffice for various definitions of *honest* have already been devised for synchronous systems (e.g., Cristian et al. [1985], Gopal et al. [1990], and Schneider [1990]) and have been implemented in some efforts (e.g., Shrivastava et al. [1992]). With such a protocol, only the subsystem of servers must be synchronous, because an atomic broadcast protocol for client requests can be implemented using an atomic broadcast protocol for the servers alone by having the servers atomically broadcast requests on behalf of clients. One drawback of protocols that rely on a synchronous system is that such systems can be difficult to build and maintain.

A second approach to circumventing the result of Fischer et al. [1985] is to employ randomization techniques. While there are several randomized solutions to the consensus problem in asynchronous systems (see Chor and Dwork [1989]), there has been much less research on randomized atomic broadcast protocols. In a private communication in May, 1992, T. D. Chandra described a method to transform any solution to consensus into a solution to atomic broadcast, which automatically yields randomized atomic broadcast protocols for asynchronous systems from randomized consensus protocols. Unfortunately, the protocols produced by this translation are impractical for general use.

Many real systems circumvent the impossibility of deterministic, asychronous atomic broadcast in more benign failure models by employing an appropriate *membership protocol* (e.g., Ricciardi [1992]). A membership protocol is used to remove a process from participation in the atomic broadcast protocol if it *appears* to be faulty. While this risks the exclusion of a correct process from the broadcast protocol, it eliminates the factor that makes atomic broadcast impossible in asynchronous systems, namely, that is is impossible to determine whether a process has actually failed or is only very slow. We have designed and are presently implementing a membership protocol for use in our system model that enables us to achieve a variation of atomic broadcast that is sufficient for our purposes [Reiter 1994].

Finally, we should note that for services for which requests commute, atomic broadcast is not necessary. For instance, this was the case for the authentication service discussed in Section 4.4.

6.1.2 *Verifiable Threshold Signature Schemes and Cryptosystems.* A second direction for future work is the development of *verifiable* threshold signature schemes and cryptosystems, which were initially suggested to us by Yair Frankel. In addition to the properties discussed in Sections 4.1 and 5.1, a verifiable threshold signature scheme or cryptosystem would enable the servers to detect whether partial results from other servers were created correctly (cf., verifiable secret sharing schemes such as in Feldman [1987]). To our knowledge, no implementation of such a signature scheme or cryptosystem has been proposed. However, in principle this detection capability could enable us to eliminate the (worst-case) exponential growth as a function of b of the time to search for partial results that decrypt a request in *deliver'* or sign a response in *respond'*. Thus, the development of verifiable signature schemes and cryptosystems may help to improve the efficiency of our protocols.

6.2 Related Work

This work was largely inspired by Gong [1989], which presents a replicated, shared-key authentication service. The authentication service described there allows two principals to establish a secret, shared encryption key provided that $n > t + b$. The method discussed in the present work cannot immediately be applied to construct such a service, because of the additional secrecy requirements. However, as discussed in Section 4.4, our method has been used to construct an analogous public-key authentication service. Moreover, unlike our scheme, the scheme of Gong requires each client to authenticate each server individually.

Using the state machine approach to construct services tolerant of arbitrary failures with authentication was first considered in Lamport [1978a]. Since then, other authors have focused on secure replication of *data*. Secure data replication using quorum methods was discussed in Herlihy and Tygar [1988] for the case in which both data integrity and secrecy are important. In these schemes, however, clients are again expected to be able to authenticate data repositories individually. In Rabin [1989], a space-efficient information dispersal algorithm was developed to make data highly available. The scheme decomposes a file F into pieces, each of size $|F|/l$, such that any l pieces suffice to reconstruct F. This scheme was extended in Krawczyk [1993] to defend against modification of these pieces.

ACKNOWLEDGMENTS

This work benefited from discussions with and information provided by Matt Blaze (AT & T Bell Laboratories), Tushar Chandra (Cornell University), Yair Frankel (GTE Laboratories), Li Gong (SRI International), Jack Lacy (AT & T Bell Laboratories), Tom London (AT & T Bell Laboratories), Andrew Odlyzko (AT & T Bell Laboratories), and Sam Toueg (Cornell University). We also gratefully acknowledge the suggestions of the anonymous referees.

REFERENCES

BIRMAN, K. P., SCHIPER, A., AND STEPHENSON, P. 1991. Lightweight causal and atomic group multicast. *ACM Trans. Comput. Syst. 9*, 3 (Aug.), 272–314.

BRICKELL, E. 1990. A survey of hardware implementations of RSA. In *Advances in Cryptology —CRYPTO '89 Proceedings*. Lecture Notes in Computer Science, vol. 435. Springer-Verlag, New York, 368–370.

CHOR, B., AND DWORK, C 1989. Randomization in Byzantine agreement. *Adv. Comput Res. 5*, 443–497.

CRISTIAN, F., AGHILI, H., STRONG, R., AND DOLEV, D. 1985. Atomic broadcast: From simple message diffusion to Byzantine agreement. In *Proceedings of the 15th International Symposium on Fault-Tolerant Computing*. 200–206. A revised version appears as IBM Res. Lab. Tech. Rep. RJ5244 (April 1989), IBM, Armonk, N.Y.

DENNING, D. E. 1984. Digital signatures with RSA and other public-key cryptosystems. *Commun. ACM 27*, 4 (Apr.), 388–392.

DESMEDT Y., AND FRANKEL, Y. 1992 Shared generation of authenticators and signatures. In *Advances in Cryptology—CRYPTO '91 Proceedings*. Lecture Notes in Computer Science, vol. 576 Springer-Verlag, New York, 457–469.

DESMEDT, Y., AND FRANKEL, Y. 1990. Threshold cryptosystems. In *Advances in Cryptology—CRYPTO '89 Proceedings*. Lecture Notes in Computer Science, vol. 435. Springer-Verlag, New York, 307–315.

DOLEV, D., DWORK, C., AND NAOR, M. 1991. Non-malleable cryptography. In *Proceedings of the 23rd Annual ACM Symposium on Theory of Computing*. ACM, New York, 542–552.

ELGAMAL, T. 1985. A public key cryptosystem and a signature scheme based on discrete logarithms. *IEEE Trans. Inf. Theory. IT-31*, 4 (July), 469–472.

FELDMAN, P. 1987. A practical scheme for non-interactive verifiable secret sharing. In *Proceedings of the 28th Annual Symposium on Foundations of Computer Science*. IEEE, New York, 427–437.

FISCHER, M. J., LYNCH, N. A., AND PATERSON, M. S. 1985. Impossibility of distributed consensus with one faulty process. *J. ACM 32*, 2 (Apr.), 374–382.

FRANKEL, Y., AND DESMEDT, Y. 1992. Distributed reliable threshold multisignature. Tech. Rep. TR-92-04-02, Dept. of EE & CS, Univ. of Wisconsin at Milwaukee

GONG, L. 1989. Securely replicating authentication services In *Proceedings of the 9th International Conference on Distributed Computing Systems*. 85–91.

GOPAL, A., STRONG, R., TOUEG, S., AND CRISTIAN, F. 1990. Early-delivery atomic broadcast. In *Proceedings of the 9th Annual ACM Symposium on Principles of Distributed Computing*. ACM, New York, 297–309.

HERLIHY, M. P., AND TYGAR, J. D. 1988. How to make replicated data secure. In *Advances in Cryptology—CRYPTO '87 Proceedings*. Lecture Notes in Computer Science, vol 293. Springer-Verlag, New York, 379–391.

JOSEPH, M. K. 1987. Towards the elimination of the effects of malicious logic: Fault tolerance approaches. In *Proceedings of the 10th NBS/NCSC National Computer Security Conference*. NBS/NCSC, Washington, D.C., 238–244.

KRAWCZYK, H. 1993. Distributed fingerprints and secure information dispersal. In *Proceedings of the 12th Annual ACM Symposium on Principles of Distributed Computing*. ACM, New York, 207–218.

LAIH, C. S. AND HARN, L. 1991. Generalized threshold cryptosystems In *Pre-Proceedings of ASIACRYPT '91*.

LAMPORT, L. 1978a. The implementation of reliable distributed multiprocess systems. *Comput. Netw. 2*, 95–114.

LAMPORT, L. 1978b. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM 21*, 7 (July), 558–565.

LAMPSON, B., ABADI, M., BURROWS, M., AND WOBBER, E. 1992. Authentication in distributed systems: Theory and practice. *ACM Trans. Comput. Syst. 10*, 4 (Nov.), 265–310.

LIM, C. H., AND LEE, P. J. 1994. Another method for attaining security against adaptively chosen ciphertext attacks. In *Advances in Cryptology—CRYPTO '93 Proceedings*. Lecture Notes in Computer Science, vol. 773. Springer-Verlag, New York, 420–434.

ORUP, B., SVENDSEN, E., AND ANDREASEN, E. 1991. VICTOR: An efficient RSA hardware implementation. In *Advances in Cryptology—EUROCRYPT '90 Proceedings*. Lecture Notes in Computer Science, vol. 473. Springer-Verlag, New York, 245–252.

RABIN, M. O. 1989. Efficient dispersal of information for security, load balancing, and fault tolerance. *J. ACM 36*, 2 (Apr.), 335–348.

REITER, M. K. 1994. A secure group membership protocol. In *Proceedings of the 1994 IEEE Symposium on Research in Security and Privacy*. IEEE, New York. To be published.

REITER, M. K. 1993. A security architecture for fault-tolerant systems. Ph.D. thesis, Cornell Univ., Ithaca, N.Y.

REITER, M. K., AND GONG, L. 1993. Preventing denial and forgery of causal relationships in distributed systems. In *Proceedings of the 1993 IEEE Symposium on Research in Security and Privacy*. IEEE, New York, 30–40.

REITER, M. K., BIRMAN, K. P., AND GONG, L. 1992. Integrating security in a group oriented distributed system. In *Proceedings of the 1992 IEEE Symposium on Research in Security and Privacy*. IEEE, New York, 18–32.

RICCIARDI, A. M. 1992. The group membership problem in asynchronous systems. Ph.D. thesis, Cornell Univ., Ithaca, N.Y.

RIVEST, R. L. 1991. The MD4 message digest algorithm. In *Advances in Cryptology—CRYPTO '90 Proceedings*. Lecture Notes in Computer Science, vol. 537. Springer-Verlag, New York, 303–311.

RIVEST, R. L., SHAMIR, A., AND ADLEMAN, L. 1978. A method for obtaining digital signatures and public-key cryptosystems. *Commun. ACM 21*, 2 (Feb.), 120–126.

SCHNEIDER, F. B. 1990. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Comput. Surv. 22*, 4 (Dec.), 299–319.

SEDLAK, H. 1988. The RSA cryptography processor. In *Advances in Cryptology: Proceedings of EUROCRYPT '87*. Lecture Notes in Computer Science, vol. 304. Springer-Verlag, New York, 95–105.

SHAND, M., AND VUILLEMIN, J. 1993. Fast implementations of RSA cryptography. In *Proceedings of the 11th IEEE Symposium on Computer Arithmetic*. IEEE, New York.

SHRIVASTAVA, S. K., EZHILCHELVAN, P. D., SPEIRS, N. A., TAO, S., AND TULLY, A. 1992. Principal features of the VOLTAN family of reliable node architectures for distributed systems. *IEEE Trans. Comput. 41*, 5 (May), 542–549.

STEINER, J. G., NEUMAN, C., AND SCHILLER, J. I. 1988. Kerberos: An authentication service for open network systems. In *Proceedings of the USENIX Winter Conference*. USENIX Association, 191–202.

TURN, R., AND HABIBI, J. 1986. On the interactions of security and fault-tolerance. In *Proceedings of the 9th NBS/NCSC National Computer Security Conference*. NBS/NCSC, Washington D.C., 138–142.

VOYDOCK, V. L., AND KENT, S. T. 1983. Security mechanisms in high-level network protocols. *ACM Comput. Surv. 15*, 2 (June), 135–171.

Received June 1992; revised May 1993; accepted July 1993