How to Securely Replicate Services*

Michael Reiter
Kenneth Birman

TR 92-1287
(replaces TR 92-1274)
June 1992

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

*This work was suported 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, 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.

How to Securely Replicate Services*

Michael Reiter
reiter@cs.cornell.edu

Kenneth Birman
ken@cs.cornell.edu

Department of Computer Science
Cornell University
Ithaca, New York 14853

June 15, 1992

Abstract

A method is presented 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. More precisely, a service is replicated by n servers in such a way that a correct client will accept a correct server's response if, for some prespecified parameter k, at least k servers are correct and fewer than k servers are corrupt. The issue of maintaining causality among client requests is also addressed. A security breach resulting from an intruder's ability to effect a violation of causality in the sequence of requests processed by the service is illustrated. An approach to counter this problem is proposed that requires fewer than k servers to be corrupt and that is live if at least $k+b$ servers are correct, where b is the assumed maximum total number of corrupt servers in any system run. An important and novel feature of these schemes is that *the client need not be able to identify or authenticate even a single server.* Instead, the client is required only to possess at most two public keys for the service. The practicality of these schemes is illustrated through a discussion of several issues pertinent to their implementation and use, and their intended role in a secure version of the Isis system is also described.

*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.

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 appropriate *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 immune to failure, however, then the service must be *replicated*.

In hostile environments, replication introduces other problems. For instance, it is often more difficult, or at least requires more resources, to protect many servers from corruption by an intruder than it is to protect only a single server. A replicated service should thus 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* [25] to replicating the service, so that each server individually computes the response and sends it to the client. If the client authenticates the response from each server and accepts the response, if any, sent by a majority of servers, then it obtains the correct response if a majority of servers are correct. Such schemes, however, require that the client be able to identify and authenticate the servers that comprise the service. 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 and authentication information of the servers.

In this paper we propose a combined solution to these problems using the state machine approach. The service is implemented by n servers in such a way that for some prespecified parameter k, a correct client accepts a response from the service provided that at least k servers are correct. Moreover, if fewer than k servers are corrupt, any response accepted at a correct client is guaranteed to have been computed by a correct server. An important feature of this scheme is that the client possesses exactly one public key for the service (as opposed to, e.g., 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. We emphasize that the client need not know the identity of even a single server to authenticate the response of the service.

Even in a system with fewer than k corrupt servers, at least k correct servers, and the above guarantees, correct clients may accept improper responses from the service if an intruder has caused the correct servers to process improper requests or to process requests in an incorrect order. In this paper 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. (While similar to an attack described in [22], this attack is more severe because it involves corrupt servers.) We also propose a way to avoid this attack that requires that the client possess at most one additional public key for the service and that fewer than k servers be corrupt. And, this solution is live if at least $k+b$ servers are correct, where b is the assumed maximum total number of corruptions suffered by servers in any system run.

The above discussion may be evocative of the large body of literature that provides solutions to various distributed computing problems in models where Byzantine failures can occur but authentication is possible (see, e.g., [19]). Nevertheless, our work has a somewhat different emphasis: we employ novel cryptographic techniques to achieve the aforementioned results, and one contribution of our work is the demonstration of the practical value of these techniques. Our approach thus stands in contrast to the body of literature just described, which typically assumes only a conventional digital signature scheme.

While we have not yet implemented the protocols described in this paper, we intend to use them in a secure version of the Isis distributed programming toolkit [22]. (See appendix B.) The Isis system provides process group and reliable group multicast abstractions that facilitate the construction of high-performance, fault-tolerant applications [1, 2]. Several services that will be provided in the secure version of Isis, including a secure name service and a secure public key authentication service, could be implemented using the techniques described here. And, these techniques may be a useful tool to support for application programmers. We devote one section of this paper to a discussion of several practical issues that arise in such an implementation and use of our protocols, as well as ways to optimize our methods in practice.

The remainder of this paper is structured as follows. In section 2 we give a brief overview of the state machine approach to replication; for more detail, the reader should see [25]. 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. Section 6 discusses several issues pertinent to the implementation of our protocols. We outline related work in section 7 and conclude in section 8.

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. A *client* of the state machine invokes a command by issuing a request to the state machine. Each command is implemented by a deterministic program and is executed atomically with respect to other commands. Commands should be executed by a state machine in an order that is consistent with Lamport's causality relation [18]. That is, two 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 receiving both should process the former first. Execution of 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 fault-tolerant service by simultaneously employing many state machine servers and coordinating client interactions with them. If all servers are initialized to the same state, and if all correct servers process the same sequence of requests, then all correct servers will give the same response to any given request. By properly combining the responses of the servers, where "properly" depends on the type of failures being considered, the response of the fault-tolerant service is obtained.

3 The system model

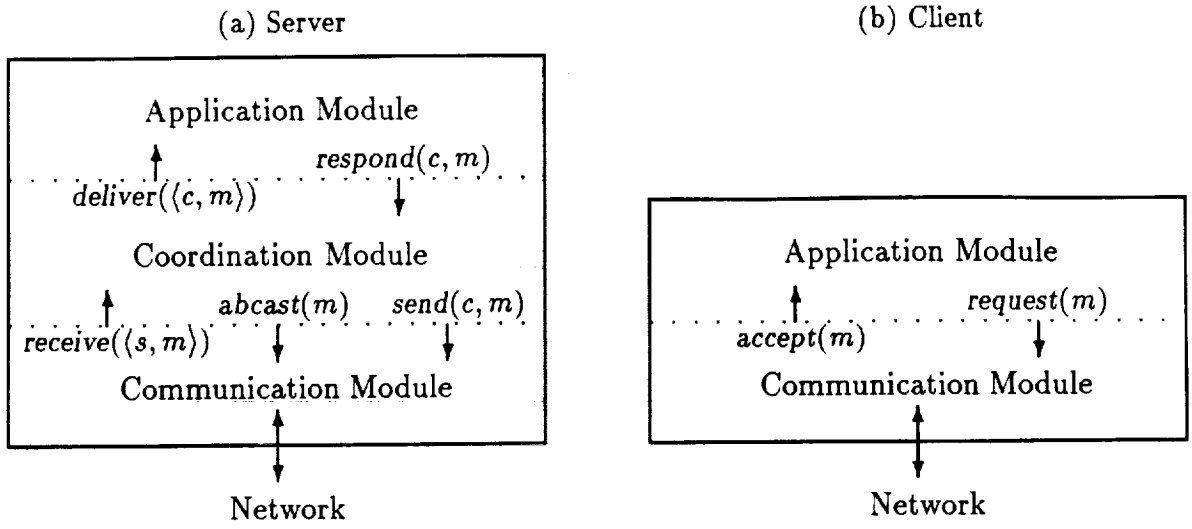
Our system consists of a set of *principals*, n of which are servers and the remainder of which are clients. All principals communicate exclusively via a network of arbitrary topology. A principal is *correct* in a run of the system if it always satisfies its specification. A principal may *fail* in an arbitrary manner, limited only by the (conjectured) properties of the cryptosystems and signature schemes we employ. Our failure model is thus most similar to "Byzantine with message authentication."

In order to capture the notions of an "accidental" failure versus a purposeful corruption by an intruder, we partition the faulty principals into two sets: the *honest* principals and the *corrupt* principals. Formally, the only property that this partitioning must have is that any principal that ever suffers "truly Byzantine" failures—i.e., failures that cannot be classified as fail-stop, crash, omission, or timing failures [4]—must be classified as *corrupt*. In a given system run, we let *corrupt* and *correct* (in *slanted* font) denote the numbers of corrupt and correct servers, respectively. We assume that there exist known constants t and b such that $b \leq t < n$ and in any system run, $corrupt \leq b$ and $correct \geq n - t$. That is, in any system run at most t servers fail, and of these at most b are "malicious."

Although principals fail as a single unit, it is convenient to view each principal as consisting of logically separate *modules*. (See figure 1.) More precisely, each server consists of a *communication module*, a *coordination module*, and an *application module*. The communication module is closest to the network, the application module is furthest, and the coordination module lies in between. The application module of a server is simply a state machine as described in section 2. The coordination module *delivers* a request $\langle c, m \rangle$, from client c and with contents m, to that state machine by calling $deliver(\langle c, m \rangle)$, which places $\langle c, m \rangle$ at the end of a list of requests to be processed that is available to be read by the state machine. We assume that calls to *deliver* are made strictly sequentially, in the sense that a call to *deliver* is not made until all previous calls to *deliver* have returned. Using the primitives supplied by the communication module, the coordination module also implements a *respond* primitive $respond(c, m)$ by which the state machine can send a response m to a client c.

The communication module implements two communication primitives for use by the coordination module. The first is a *send* primitive $send(c, m)$ by which the coordination module can send a message to a client; this is presumably used in the implementation of $respond(c, m)$ and will not

Figure 1: Structure of principals



be used further in this paper. The second is an *atomic broadcast* primitive $abcast(m)$ by which the coordination module of a server s can broadcast a message $\langle s, m\rangle$ to the set of servers. A server's communication module *receives* a message $\langle s, m\rangle$, from server s and with contents m, by calling $receive(\langle s, m\rangle)$, which places $\langle s, m\rangle$ at the end of a list of received messages that is available to be read by the coordination module. Messages are received only from servers, according to an atomic broadcast protocol $\mathcal{R}$ that satisfies the following specification.

Receipt Atomicity: A message is either received at all correct servers exactly once or is never received at any correct server.

Receipt Validity: A correct server receives a message from a correct or honest server s only if s previously broadcast that message. Moreover, if a correct server broadcasts a message, it will eventually be received at all correct servers.

Receipt Order: A correct server receives message $\langle s, m\rangle$ before another message $\langle s', m'\rangle$ iff all correct servers do. That is, all correct servers receive the same sequence of messages.

Receipt Consistency: The sequence of messages received by an honest server is a prefix of the sequence of messages received by a correct server.[1]

Options for implementing atomic broadcast are discussed in section 6; here we simply note that there already exist protocols in the literature that satisfy this specification in various models and for

[1]It is shown in [13] that even in a synchronous system, any atomic broadcast protocol that is tolerant of omission failures and that guarantees consistency with respect to faulty principals requires a majority of correct principals. Thus, if *honest* is defined to include those servers that suffer omission failures, the results of this paper require $n > 2t$.

various definitions of *honest*. Our protocols do not rely upon any bounds on message transmission times or the execution speeds of principals, and so the only such bounds required for our results, if any, are those required by the particular atomic broadcast protocol used.

A client consists of only two modules, an *application module* and a *communication module*. The application module of a client c is some client program that can issue a request $\langle c, m \rangle$ to the service by calling *request*(m). The communication module of the client implements this primitive, e.g., by signing and broadcasting $\langle c, m \rangle$ to the entire network. Assuming that each request from a correct client eventually reaches some correct server, servers can easily implement an atomic broadcast protocol for client requests by, e.g., forwarding requests to the other servers via *abcast* and determining the delivery order of requests by their order of receipt at correct servers. That is, we assume that servers also implement a protocol, called $\mathcal{D}$, that satisfies the following properties.

Delivery Atomicity: A request is either delivered at all correct servers exactly once or is never delivered at any correct server.

Delivery Validity: A correct server delivers a request from a correct or honest client c only if c previously issued that request. Moreover, if a correct client issues a request, it will eventually be delivered at all correct servers.

Delivery Order: A correct server delivers a request $\langle c, m \rangle$ before another request $\langle c', m' \rangle$ iff all correct servers do. That is, all correct servers deliver the same sequence of requests.

Delivery Consistency: 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, these properties imply that all correct and honest servers will produce the same response (or no response) to a given request. The communication module of a client accepts a response m for the application module by calling *accept*(m).

4 Preserving integrity and availability

Recall from section 1 that our first goal is a service that satisfies the following properties, for some prespecified k.

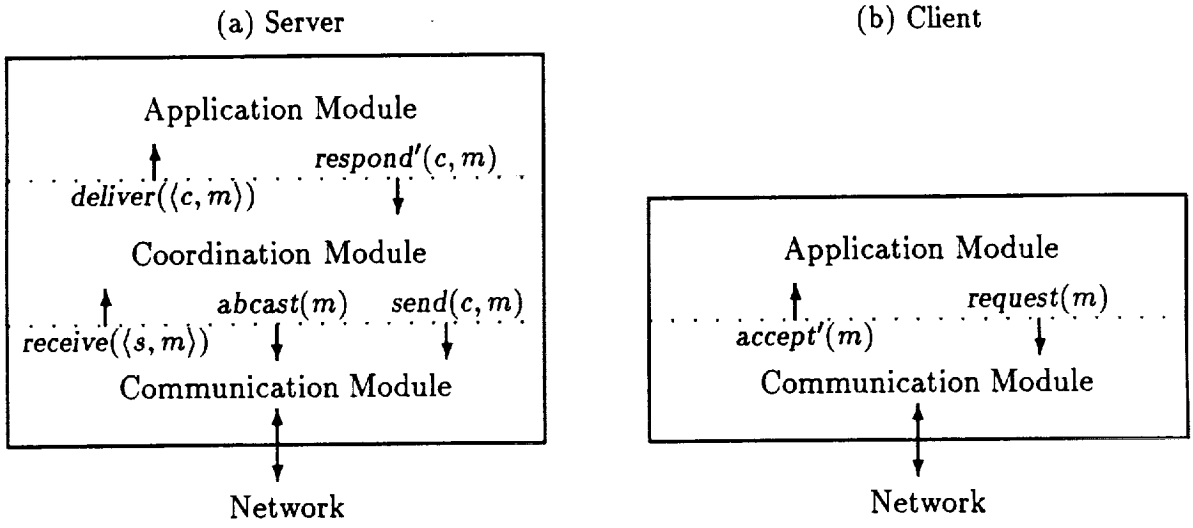
Integrity: If *corrupt* $< k$, then the response accepted at a correct client, if any, is that computed by a correct server.

Availability: If *correct* $\geq k$, then a correct client will accept a response from the service.

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 these

properties. Therefore, the new structures of principals will be as pictured in figure 2. Although we have replaced the *respond* routine with *respond'* at the interface provided to the application module of each server, we assume that *respond* is still available for execution by the coordination module. Similarly, we assume that *accept* is still available to the communication module of each client.

Figure 2: New structure of principals



Threshold signature schemes

The *respond'* routines at the different servers will employ a (k, n)-*threshold signature scheme*. A (k, n)-threshold signature scheme 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, 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.

Cryptanalytic attacks against threshold signature schemes differ from those against their conventional counterparts 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 [11], 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 corresponding to various other messages (but *not* corresponding to messages *of its choice*, as would be possible in a *chosen message* attack).

This definition is in accordance with the security of all implementations of threshold signature schemes thus far proposed, in the sense that all proposed implementations are based upon conventional signature schemes that are known to be vulnerable to chosen message attacks.

Our *respond'* routine is not dependent upon any particular implementation of a (k, n)-threshold signature scheme, although for concreteness we outline the necessary details of an implementation proposed in [7]. The scheme begins with an RSA [24] 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.[2] So, by defining the i-th 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 $m^d \bmod N$ for m.[3] Variations of this scheme have been proved to be as secure as RSA, in the sense of property 1 above [11].

The protocol

Suppose that a public key (e, N) and corresponding shares $\{K_i\}_{1 \le i \le n}$ are created as above and distributed so that for all i, $1 \le i \le n$, server s_i is the sole possessor of K_i and any principal 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 implementation of the servers. Then, the *respond'*(c, m) and *accept'*(m) routines are implemented as follows.

[2] Each of the integers $p_{i,T}$ for all i and T can be computed from a fixed set of n integers, each of binary length $O(\log n)$ (if so chosen), with $O(n)$ integer multiplications and additions.

[3] For reasons of security and efficiency, it is advisable that a *message digest* of the message be signed, as opposed to the message itself [5].

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

1. Execute $abcast(a_{m,i})$.
2. Wait to receive a set $\{a_{m,j}\}_{j \in T}$, $|T| = k$, of partial results for m such that $A_{m,T}$ is a valid signature for m.
3. Execute $respond(c, (m, A_{m,T}))$.

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

1. If m is not of the form (m', S), then return to the calling routine.
2. If S is a valid signature for m' and m' is a valid response (i.e., is of the right form), then execute $accept(m')$.[4]

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

Proof. Because each correct and honest server produces partial results only for responses that it computes, the assumptions that $corrupt < k$ and that the threshold signature scheme is secure imply that the corrupt servers can generate signatures only for responses computed by a correct server (and possibly for other, presumably useless, messages). So, the corrupt servers can neither directly generate an improper response that a client will accept nor create signatures for arbitrary messages of their choice in order to perform chosen message attacks against the signature scheme. Therefore, any response that is accepted by a correct client must have been computed by a correct server. □

Claim 2 *This protocol satisfies Availability.*

Proof. Suppose that $correct \geq k$. By Receipt Validity of $\mathcal{R}$, all correct servers eventually receive partial results from k correct servers, and so each correct server can compute a proper signature on its response. □

Discussion

In terms of communication complexity, in a failure-free run the replacement of *respond* with *respond'* results in an additional n executions of $\mathcal{R}$ (i.e., *abcast*), 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 (one execution of $\mathcal{D}$), the dissemination of partial results (n executions of $\mathcal{R}$), and the sending of the responses (n executions of *respond*). This protocol can be optimized in at least two ways, first by noticing that a client needs to receive only one correctly signed response for Availability to be satisfied. This implies that only $\min\{t + 1, n - k + 1\}$ servers need to be designated

[4] Here we have omitted consideration of several forms of attack, such as *replay* attacks [26].

to respond to any given request. In addition, the partial results for the signature of the response need to be broadcast only to that set of servers. The set of servers designated to respond to a given request can be fixed in advance or determined dynamically. A second optimization is for servers to communicate partial results by a *reliable broadcast* protocol, obtained by removing the Receipt Order requirement and replacing "sequence" and "prefix" with "set" and "subset," respectively, in the Receipt Consistency requirement of atomic broadcast. Since reliable broadcast is weaker, it can be implemented at least as efficiently. Reliable broadcast can be used because the order in which partial results are received by servers is not important in this protocol.

In theory, the most computationally expensive part of the 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 $A_{m,T}$ is a valid signature. The server must examine at most only the first $h = \min\{n, k+b\}$ partial results received (from h unique servers), and at most $\binom{h}{k}$ subsets of partial results, because in h partial results are at least k correct partial results (if *correct* $\geq k$). While this could be expensive if h is large and $k \approx h/2$, the expected search time should be small in the common case in most systems, i.e., when n and *corrupt* are small.[5] A minor optimization is to have a server always include its own index in T when computing $A_{m,T}$, and heuristics, such as using partial results from a combination of servers that previously worked, could also be used to reduce the search time in practice. Further optimizations, one of which is discussed in appendix A, are a topic of ongoing research. However, we emphasize that in most systems, the search time for a valid signature will probably be dominated by other costs and should not be a limiting factor in the performance of our protocols. Accordingly, we view further optimizations of this search to be primarily of theoretical interest.

In this scheme, each server must know $O(n \log n)$ bits of public information (in order to compute the $p_{i,T}$'s; see footnote 2), 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.

5 Preserving input causality

One guarantee provided in the previous section is that if *corrupt* $< k$, then the response accepted at a correct client will be the response 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 deliver improper requests or to deliver requests in an incorrect order. In general, ensuring proper responses from a correct server requires *access control*, because responses computed from state variables that can be written (directly or indirectly) by corrupt clients cannot be trusted. Access control is an entire research area in itself and will not be discussed further here.

[5] For example, in typical Isis applications, a service might be implemented by three to five servers, but rarely more. For such services this search would be insignificant.

In this section we address the issue of ensuring that requests are delivered in a correct order by correct servers. Because we assume an atomic broadcast protocol $\mathcal{D}$ 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 preserving causality among client requests is for each client to refrain from sending any messages 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 [25]. Consider the case, however, in which a correct client issues a request to the service, and after receiving 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 [22]: suppose that the service of interest 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, as the intent is that any client can request to purchase stock at any time.

In the remainder of this section, we present new request and delivery routines, respectively denoted *request'*(m) and *deliver'*$(\langle c, m\rangle)$, that replace *request*(m) and *deliver*$(\langle c, m\rangle)$. Therefore, if used with the *respond'* and *accept'* routines of section 4, principals would be structured as in figure 3. These new routines protect correct clients from the type of attack described above, in the sense that any request based on information obtained from a correct client's request $\langle c, m\rangle$ can be delivered at correct servers only after $\langle c, m\rangle$. As before, we will use *deliver* in our implementation of *deliver'*, and similarly for *request* and *request'*.

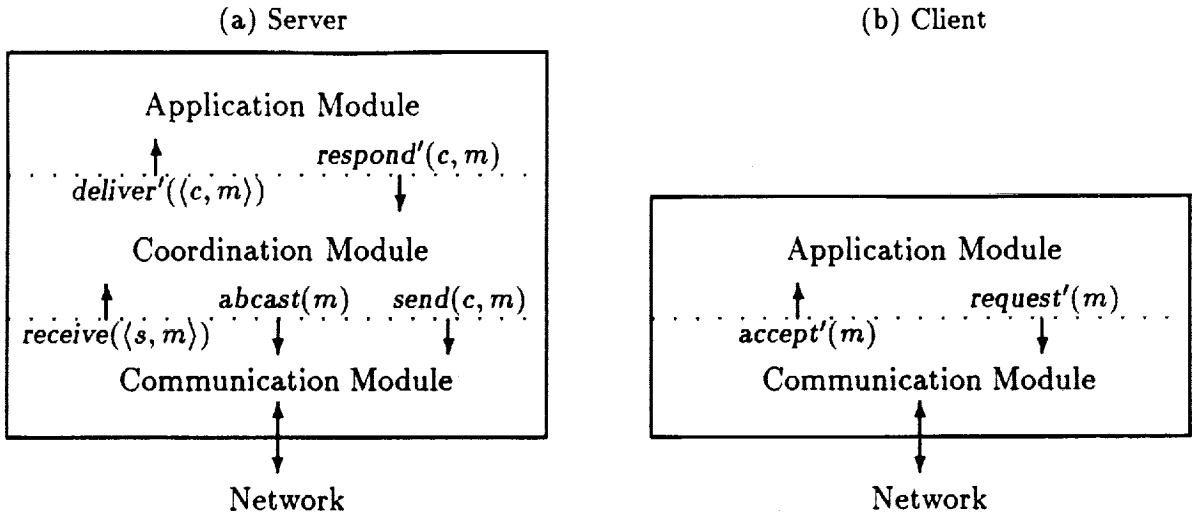
In the implementation of *request'*(m), the correct client c encrypts m under a public encryption key of the service before issuing $\langle c, m\rangle$. Then, c is provided the following guarantee. The reader should verify that this guarantee prevents the aforementioned problem, provided that *corrupt* $< k$.[6] (This k can be chosen independently of that in section 4.)

Causality: If *corrupt* $< k$, then if some request is (i) issued after m is decrypted anywhere (other than c), and (ii) delivered at a correct server, then that request is delivered at all correct servers after $\langle c, m\rangle$.

In addition to satisfying Causality, *request'* and *deliver'* must also ensure that client requests are delivered according to the specification of atomic broadcast—i.e., that Delivery Atomicity, Delivery

[6] We do not consider *traffic analysis* attacks [26] or attacks that exploit the *malleability* of the cryptosystem [8].

Figure 3: New structure of principals



Validity, Delivery Order, and Delivery Consistency still hold. The implementations of *request'* and *deliver'* that we propose satisfy all but the second half of Delivery Validity with no further assumptions; to ensure that a correct client's request will eventually be delivered at all correct servers, we require $correct \geq k + b$.

Threshold cryptosystems

Our *deliver'* routines at the different 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, 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 [11], 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).

This 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 upon 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. Messages would be encrypted under the public key (e, N) of the service in the usual manner, and the i-th partial result for an encrypted message $m \equiv (m')^e \bmod N$ would be defined precisely as in section 4—i.e., $a_{m,i} \equiv m^{K_i} \bmod N$. Then, $m' \equiv A_{m,T} \equiv m \cdot \prod_{i \in T}(a_{m,i})^{p_{i,T}}$ for any T of size k. Other implementations of threshold cryptosystems have been proposed, based upon both the RSA and ElGamal [9] cryptosystems [6, 16].

The protocol

Suppose that we are using the RSA threshold cryptosystem described above and that we have the initial conditions assumed in section 4; i.e., server s_i is secretly given sole possession of K_i, any principal 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 c encrypts the contents m of its request 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 $\langle c, m \rangle$ is fixed locally. In this way, if *corrupt* $< k$, then once a corrupt server collects k partial results for m, the sequence of requests through $\langle c, m \rangle$ has been fixed at some, and thus all, correct servers, and no requests can be placed before $\langle c, m \rangle$ in the delivery sequence at correct servers.

This protocol preserves Causality iff each server requires k partial results for m to decrypt m. Even under the assumption that this cryptosystem is secure, however, this unfortunately is *not* the case with this or any proposed implementation of a (k, n)-threshold cryptosystem. 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 threshold cryptosystem based upon 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 c issue $\langle c, m \rangle$ 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 illustrate one method, originally due to Moore (see [5]), by which a corrupt server can decrypt the RSA ciphertext $m \equiv (m')^e \bmod N$ in a correct client's request $\langle c, m \rangle$ 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., $x \equiv y^d \bmod N$.[7]
2. Via a corrupt client, the corrupt server issues a request with contents $ym \bmod N$ 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 because $gcd(x, N)$ is a prime factor of N.)

Similar attacks are possible with the threshold cryptosystems described in [6, 16].

Ideally, we would like to find a threshold cryptosystem based on a conventional public key cryptosystem that is tolerant of chosen ciphertext attacks. As already mentioned, however, no such threshold cryptosystem has been proposed, and even conventional public key cryptosystems that are tolerant of such attacks (e.g., [8]) are highly impractical. Therefore, such attacks must be *prevented.* A simple way to do this is to have each client use a separate public encryption key for the service. That is, the public key for the l-th client c_l would be a pair (e_l, N_l), and each server s_i would be given a share $K_{i,l}$ for use when cooperating to decrypt a request from client c_l. This prevents chosen ciphertext attacks against the keys and ciphertexts of correct clients, because any request received from a corrupt client will be decrypted using the shares of the key for that client, and not a correct one. The practical ramifications of this design, as well as ways to improve it in practice, are discussed in section 6.

The one remaining problem in our protocol is that corrupt servers can provide incorrect partial results that may disrupt the decryption process. Therefore, it must be possible for a server to determine when it has properly decrypted a request. To facilitate this, clients are required to send

[7] The obvious form of this attack, in which $x \equiv y \equiv 1 \bmod N$, could easily be made unproductive for the corrupt server: if each correct client signs the contents of its request *before* encrypting it and each correct server delivers a request on behalf of the client whose signature appears on the decrypted contents (and not necessarily the client that issued the request), then by issuing a request containing m, the corrupt server only expedites the delivery of the correct client's request.

with each request a *message digest* of the request. Message digests have the property that the message digest of a given message can be computed efficiently, but it is computationally infeasible to produce two messages having the same message digest or to produce any message having a prespecified target message digest. Accordingly, we henceforth assume that the included message digest uniquely identifies, but does not disclose, the contents of the encrypted message, and that the joint use of message digests and the threshold cryptosystem does not reveal information that a cryptanalyst could use to circumvent the properties of either one. The validity of this assumption obviously depends on the chosen implementation of message digests. We also note that if the message digest function is injective, a message digest does, in fact, uniquely identify a message.[8] Several efficient implementations of message digest functions have been proposed (e.g., [23]) but will not be discussed here. Let $digest(m)$ denote the message digest of m.

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

Routine $request'(m)$ at client c_l:

1. Create the RSA ciphertext $m' \equiv m^{e_l} \bmod N_l$ and message digest $D = digest(m)$ of m.
2. Execute $request(\langle m', D \rangle)$.

Routine $deliver'(\langle c_l, m \rangle)$ at server s_i:

1. If m is not of the form $\langle m', D \rangle$, then return to the calling routine.
2. Execute $abcast(a_{m',i})$, where $a_{m',i} \equiv (m')^{K_{s,i}} \bmod N_l$.
3. Wait until the first $k+b$ partial results $\{a_{m',j}\}_{j \in T}$, $|T| = k+b$, for m' have been received (from $k+b$ unique servers).
4. Search for a subset $T' \subseteq T$ of size k such that $digest(A_{m',T'}) = D$. If such a T' exists and $A_{m',T'}$ is a valid request, then execute $deliver(\langle c_l, A_{m',T'} \rangle)$.

Claim 3 *This protocol satisfies Delivery Atomicity, Delivery Order, and Delivery Consistency.*

Proof. (Delivery Atomicity) By Delivery Atomicity of $\mathcal{D}$, for each client request *deliver'* is either called exactly once at all correct servers or never called at any correct server. Clearly a request of the latter type is never delivered at any correct server, and so now consider a request $\langle c, m \rangle$, $m = \langle m', D \rangle$, of the former type. If only fewer than $k+b$ partial results for m' are received at correct servers, then the request will never be delivered at any correct server. So, suppose that $k+b$ partial results for m' are received at all correct servers. By Receipt Order of $\mathcal{R}$, all correct servers employ precisely the same set $\{a_{m',j}\}_{j \in T}$, $|T| = k+b$, of partial results when attempting to decrypt m'. Therefore, one

[8] One such implementation employs a deterministic public key cryptosystem: the digest for a message is computed by encrypting the message under an *a priori*, commonly known public key, for which the corresponding private key has been destroyed and is not known.

correct server delivers some request iff all correct servers do, and all correct servers deliver the same request, assuming that D uniquely identifies a single request.

(Delivery Order) By Delivery Order of $\mathcal{D}$, all correct servers execute the same sequence of *deliver'* calls. And, because any call to *deliver'* returns before *deliver'* is called again, it follows from the argument for Delivery Atomicity that all correct servers execute the same sequence of *deliver* calls.

(Delivery Consistency) By Delivery Consistency of $\mathcal{D}$, the sequence of *deliver'* calls at an honest server is a prefix of the sequence of *deliver'* calls at a correct server. Moreover, because the sequence of messages received at an honest server is a prefix of the sequence of messages received at a correct server (by Receipt Consistency of $\mathcal{R}$), if the honest server receives sufficiently many messages, it will use the same set $\{a_{m',j}\}_{j \in T}$, $|T| = k + b$, of partial results to decrypt each request as the correct servers do. Thus, the sequence of requests delivered at an honest server will be a prefix of the sequence of requests delivered at a correct server. □

Claim 4 *A correct server delivers a request from a correct or honest client c only if c previously issued that request. (That is, the first half of Delivery Validity holds.)*

Proof. This follows immediately from the fact that, by Delivery Validity of $\mathcal{D}$, if c is correct or honest, then *deliver'*$(\langle c, m \rangle)$ is called at a correct server only if c issued the request $\langle c, m \rangle$. □

Claim 5 *If correct $\geq k + b$, then if a correct client issues a request, it will be delivered at all correct servers. (That is, if correct $\geq k + b$, then the second half of Delivery Validity holds.)*

Proof. If *correct* $\geq k + b$, then for each call to *deliver'* at correct servers, at least $k + b$ partial results are broadcast by and, therefore, received at all correct servers. So, at a correct server, each call to *deliver'* eventually returns; i.e., the service makes progress. This, together with Delivery Validity of $\mathcal{D}$, implies that for any request $\langle c, m \rangle$, $m = \langle m', D \rangle$, issued by a correct client c, *deliver'*$(\langle c, m \rangle)$ is eventually executed at all correct servers. Since the $k + b$ partial results used to decrypt m' contain at least k partial results from correct and honest servers, m' can be decrypted and delivered. □

From the above claims, we immediately have the following.

Claim 6 *This protocol satisfies the specification of atomic broadcast (for client requests) whenever correct $\geq k + b$.*

We now prove that the above protocol satisfies Causality.

Claim 7 *If the threshold cryptosystem is secure, then this protocol satisfies Causality.*

Proof. Suppose that the threshold cryptosystem is secure and *corrupt* $< k$. Then, the earliest point at which the ciphertext m' in a request $\langle c, m \rangle$, $m = \langle m', D \rangle$, from a correct client c can be decrypted

anywhere is sometime after some correct or honest server broadcasts its partial result for m'. Let s be the first correct or honest server to broadcast its partial result for m'. By Delivery Order and Delivery Consistency of $\mathcal{D}$, all correct servers eventually execute (possibly an extension of) the same sequence of *deliver'* calls that s executes. Therefore, all correct servers will execute *deliver'*$(\langle c, m\rangle)$ before *deliver'*$(\langle \tilde{c}, \tilde{m}\rangle)$ for any $\langle \tilde{c}, \tilde{m}\rangle$ issued after m' was decrypted. If all correct servers deliver (the plaintext corresponding to) m', then Causality is satisfied. Moreover, because c is correct, the only way in which m' could not be delivered at all correct servers is if the correct servers never receive $k+b$ partial results for m'. In this case, the *deliver'*$(\langle c, m\rangle)$ call at each correct server will never return, and no more requests will be delivered at any correct server, thus trivially satisfying Causality. □

Discussion

In a failure-free run, the replacement of *deliver* with *deliver'* results in an additional n executions of $\mathcal{R}$ (i.e., *abcast*), which can be executed concurrently. Thus, when this protocol is used to disseminate client requests and the protocol of section 4 is used to sign responses, the total message complexity is $2n + 1$ broadcasts (n of which can be reliable only; see section 4) plus the number of responses, structured in four communication phases. It is worth noting that if, e.g., the same value for k is chosen for both protocols, then Availability can be guaranteed only if *correct* $\geq k + b$, and so the number of servers designated to respond to each request should be reduced to $\min\{t+1, n-(k+b)+1\}$.

As in the protocol of section 4, step 4 of *deliver'* is potentially expensive, because it may require a server to sort through $\binom{k+b}{k}$ subsets of partial results to be able to decrypt a request. In fact, a corrupt client can *force* each server to sort through $\binom{k+b}{k}$ subsets by sending a ciphertext and digest that do not "match." As before, we rely on heuristics, a small $\binom{k+b}{k}$, and a small *corrupt* in the common case to reduce the expected cost of decrypting requests from correct clients. A bad request from a corrupt client can be detected in a reasonable amount of time if $\binom{k+b}{k}$ is small, and then subsequent requests from that client can be ignored. We reiterate, however, that we do not expect the cost of decrypting requests to be a limiting factor in most systems, and so while we are pursuing additional optimizations to this process (see appendix A), we do not view this as essential.

Assuming that service responses are verified at all clients with a single public key but that each client that desires Causality has an additional public encryption key for the service, each client possesses at most two public keys for the service. Each server now possesses one key share per client desiring Causality, in addition to the key share with which it cooperates to sign responses and the $O(n \log n)$ bits of public information needed for computing the $p_{i,T}$'s. (The same $p_{i,T}$'s can be used for all instances of the signature scheme and cryptosystem.)

Finally, we note that while this protocol prevents a potentially serious attack arising from violations of causality, it does not necessarily address all such attacks. A further examination of the relationship between security and causality is a topic of ongoing research.

6 Implementation issues

As mentioned in section 1 and discussed further in appendix B, we intend to use the techniques of sections 4 and 5 in a secure version of the Isis distributed programming toolkit. In this section we comment on three practical issues that bear on the implementation and use of our protocols. We do not discuss all relevant issues; several important problems, such as the periodic changing of keys or the distribution of key shares, have purposely been omitted from discussion because they are similar to problems encountered in other cryptographic systems and can be solved using known techniques. Instead, our comments here are restricted to issues that may not be pertinent to most other cryptographic protocols.

Atomic broadcast

As previously described, our protocols assume the existence of an underlying atomic broadcast protocol, whose specification is given in section 3. It is well-known, however, that it is impossible to find a deterministic solution to consensus, and thus atomic broadcast, in an asynchronous system that can suffer even a single crash failure [10]. While at first this may appear to diminish the usefulness of our protocols, several systems (e.g., Isis [2] and Amoeba [15]) have successfully circumvented this impossibility result for all practical purposes.

In the short term, systems such as these appear to be the most promising platforms for implementing our protocols. The idea is that the communication modules of servers would be implemented using, e.g., the Isis atomic multicast primitives. Because the protocols used by Isis are tolerant of only benign failures, however, additional precautions would need to be taken to insulate the Isis protocols from an intruder. In many systems this could be achieved by running Isis as part of the operating system of each server site and then using secure boot procedures and physical security to protect the integrity of site operating systems. These measures, when coupled with the security mechanisms of [22] to defeat network intruders and to prevent unauthorized sites from participating in the protocols, should make the Isis protocols sufficiently robust for most environments. The coordination and application modules of servers could then run as application processes, so that even if some were corrupted or replaced (e.g., because their owners' passwords were guessed) the Isis protocols would remain intact.

In extremely hostile environments, however, it may be impossible to protect all server sites from corruption, and thus the above approach may not suffice. In this case, atomic broadcast protocols tolerant of Byzantine failures (with authentication) are required. Protocols that suffice for various definitions of *honest* have already been implemented (e.g., [4]) for synchronous systems, i.e., systems in which there are known bounds on message transmission times and execution speeds of (correct) processes. 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, as outlined in section 3. Thus, this approach might be feasible in a situation where the servers can communicate by a highly reliable and predictable network, but where the clients are more widely dispersed and unable to communicate with the servers by such predictable means. Moreover, if the subsystem of servers is synchronous, then our protocols can be improved: e.g., the protocol of section 5 can easily be modified so that it is live if *correct* $\geq k$. There has been little work on Byzantine atomic broadcast protocols for partially synchronous and asynchronous systems. However, in a recent private communication, T. D. Chandra described a method to transform any solution to Byzantine consensus[9] into a solution to Byzantine atomic broadcast.

Public encryption keys

Recall that the protocol of section 5 requires that each client have a different public encryption key for the service, in order to prevent a corrupt server from mounting chosen ciphertext attacks against correct clients' ciphertexts or keys. In general, having separate cryptosystem parameters for each client may require that each client be individually "registered" with the service before using it, so that the client can obtain a public key for the service and the administrator of the service can provide to the servers the shares necessary to decrypt this client's requests. While this could limit the settings in which our protocols can be used, we believe that this is not unreasonable if maintaining causality among client requests is a substantial concern.

Nevertheless, in practice it may be possible to relax the requirement that each client have its own public encryption key for the service. For instance, if a group of clients mutually trust one another to not participate in chosen ciphertext attacks against one another, then the members of that group might choose to use the same public encryption key for the service. To illustrate this, recall the trading service example of section 5. Assuming that a broker will not cooperate in chosen ciphertext attacks against other brokers in the same firm, all brokers in a single firm could use the same public encryption key for the service. This would clearly simplify key management: e.g., after a firm authorized a new broker to issue requests to the service on its behalf, each server could use its share of the private key corresponding to the public key given to that firm when cooperating to decrypt a request from that broker. In this way, the addition of a new client (i.e., broker) need not require the generation of a new public key/private key pair or the secure distribution of new private key shares to the servers. Another way in which the requirement that each client have its own public encryption key could be relaxed is described in the following subsection.

Detection of corrupt principals

One resource that we have not fully exploited in our protocols is the ability to *detect* corrupt principals based on messages received from those principals. Such techniques have been used, e.g., in

[9] Due to [10], any solution to Byzantine consensus in an asynchronous system must be randomized [3].

numerous Byzantine agreement protocols to isolate faulty principals. In practice, methods to detect and quarantine corrupt principals could be used to deter such principals from mounting attacks and to improve overall system performance.

One example of how a corrupt client can be detected was given in section 5: if a correct server executes $deliver'(\langle c, m\rangle)$, where $m = \langle m', D\rangle$, but in step 4 of $deliver'$ there is no set T' of size k such that $digest(A_{m',T'}) = D$, then c must be corrupt. In particular, the chosen ciphertext attack illustrated in section 5 would lead to the detection of the corrupt client, because that client could not compute $digest((ym)^d \bmod N)$ without knowing $(ym)^d \bmod N$. This technique, moreover, would detect any client issuing any ciphertext for which it did not know the corresponding plaintext. If this threat of detection is believed to be sufficient to deter chosen ciphertext attacks, then all clients could be allowed to use the same public encryption key for the service.

Our protocols could also benefit from techniques to detect corrupt servers. For instance, in appendix A we describe several improvements to our protocols that would result from the ability to detect when a server has provided an incorrect partial result. Other detection techniques and ways to exploit them are topics of ongoing research and will not be discussed further here.

Summary

While the requirements that an underlying atomic broadcast protocol be available and that each client use a different public encryption key may seem to reduce the practicality of our protocols, in many real systems these will not constitute substantial restrictions. We have described several approaches to implementing atomic broadcast, as well as methods for relaxing the requirement that each client have its own public encryption key for the service. We believe that, when combined with techniques to detect and isolate corrupt principals, our protocols can be applied in many types of systems to achieve efficient, fault-tolerant, and secure service implementations.

7 Related work

This work was largely inspired by [12], 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 for some prespecified value k, at least k servers are correct and fewer than k servers are corrupt. The method discussed in the present work cannot immediately be applied to construct such a service, because of the additional secrecy requirements. However, our method can be used to construct an analogous public key authentication service that has the additional advantage that, unlike the service described in [12], a client need only possess at most two keys for the service, and not one for each server.

Using the state machine approach to construct services tolerant of arbitrary failures with authentication was first considered in [17]. Since then, other authors have focused on secure replication

of *data*. Secure data replication using quorum methods is considered in [14] for the case in which both data integrity and secrecy are important. In these schemes, however, an intruder that corrupts a client may also be able to compromise the integrity and secrecy of all data. Moreover, clients are expected to be able to authenticate data repositories. In [20], a space-efficient information dispersal algorithm is developed to facilitate the provision of data integrity and availability. The scheme decomposes a file F into n pieces, each of size $|F|/l$, such that any l pieces suffice to reconstruct F.

8 Conclusion

We have presented a method for securely replicating services using the state machine approach. Using our technique, a service can be replicated as n servers in such a way that for some prespecified parameter k, a client will accept a response computed by a correct server provided that $corrupt < k$ and $correct \geq k$. We have also addressed the issue of maintaining causality among client requests. A security breach resulting from an intruder's ability to violate causality was illustrated, and a safe and live approach, which requires that $corrupt < k$ and $correct \geq k + b$, was presented to counter this problem. An important and novel feature of our methods is that they free the client of the responsibility of learning the identity and public key of each server. This is achieved by employing two recent advances in cryptography, namely threshold cryptosystems and threshold signature schemes. Moreover, we have argued for the feasibility of our techniques through a discussion of several issues pertinent to their implementation and use.

Acknowledgements

This work benefited tremendously from discussions with Yair Frankel (University of Wisconsin at Milwaukee). Tushar Chandra (Cornell University) contributed several discussions, and Sam Toueg (Cornell University) provided helpful comments and information. Presentational aspects of this paper benefited from comments by Li Gong (ORA Corporation) and Cliff Krumvieda (Cornell University).

References

[1] 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.

[2] 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.

[3] Chor, B., and Dwork, C. Randomization in Byzantine agreement. *Advances in Computer Research 5* (1989), 443–497.

[4] Cristian, F., Aghili, H., Strong, R., and Dolev, D. Atomic broadcast: From simple message diffusion to Byzantine agreement. In *Proceedings of the International Symposium on*

Fault-Tolerant Computing (June 1985), pp. 200–206. A revised version appears as IBM Research Laboratory Technical Report RJ5244 (April 1989).

[5] DENNING, D. E. Digital signatures with RSA and other public-key cryptosystems. *Communications of the ACM 27*, 4 (Apr. 1984), 388–392.

[6] DESMEDT, Y., AND FRANKEL, Y. Threshold cryptosystems. In *Advances in Cryptology—CRYPTO '89 Proceedings, Lecture Notes in Computer Science 435*, G. Brassard, Ed. Springer-Verlag, 1990, pp. 307–315.

[7] DESMEDT, Y., AND FRANKEL, Y. Shared generation of authenticators and signatures. In *Advances in Cryptology—CRYPTO '91 Proceedings, Lecture Notes in Computer Science 576*, J. Feigenbaum, Ed. Springer-Verlag, 1992, pp. 457–469.

[8] DOLEV, D., DWORK, C., AND NAOR, M. Non-malleable cryptography. In *Proceedings of the ACM Symposium on Theory of Computing* (May 1991), pp. 542–552.

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

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

[11] FRANKEL, Y., AND DESMEDT, Y. Distributed reliable threshold multisignature. Tech. Rep. TR-92-04-02, Department of EE & CS, University of Wisconsin at Milwaukee, Apr. 1992.

[12] GONG, L. Securely replicating authentication services. In *Proceedings of the IEEE International Conference on Distributed Computing Systems* (1989), pp. 85–91.

[13] GOPAL, A., AND TOUEG, S. Inconsistency and contamination. In *Proceedings of the ACM Symposium on Principles of Distributed Computing* (Aug. 1991), pp. 257–272.

[14] HERLIHY, M. P., AND TYGAR, J. D. How to make replicated data secure. In *Advances in Cryptology—CRYPTO '87 Proceedings, Lecture Notes in Computer Science 293*, C. Pomerance, Ed. Springer-Verlag, 1988, pp. 379–391.

[15] 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.

[16] LAIH, C. S., AND HARN, L. Generalized threshold cryptosystems. In *Advances in Cryptology—ASIACRYPT '91 Proceedings, Lecture Notes in Computer Science*. Springer-Verlag, 1992.

[17] LAMPORT, L. The implementation of reliable distributed multiprocess systems. *Computer Networks 2* (1978), 95–114.

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

[19] LAMPORT, L., SHOSTAK, R., AND PEASE, M. The Byzantine generals problem. *ACM Transactions on Programming Languages and Systems 4*, 3 (July 1982), 382–401.

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

[21] RABIN, T., AND BEN-OR, M. Verifiable secret sharing and multiparty protocols with honest majority. In *Proceedings of the ACM Symposium on Theory of Computing* (May 1989), pp. 73–85.

[22] REITER, M. K., BIRMAN, K. P., AND GONG, L. Integrating security in a group oriented distributed system. In *Proceedings of the IEEE Symposium on Research in Security and Privacy* (May 1992), pp. 18–32.

[23] RIVEST, R. L. The MD4 message digest algorithm. In *Advances in Cryptology—CRYPTO '90 Proceedings, Lecture Notes in Computer Science 537*, A. J. Menezes and S. A. Vanstone, Eds. Springer-Verlag, 1991, pp. 303–311.

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

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

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

A Verifiable threshold cryptography

In this appendix we identify several improvements to our protocols that would result from the discovery of *verifiable* threshold signature schemes and cryptosystems. For our purposes, a verifiable threshold signature scheme or cryptosystem is one with additional properties that would enable correct and honest servers to efficiently detect whether a server produced its partial result correctly. We discuss the impact of verifiable schemes on our protocols only in an appendix because to our knowledge there are no known, practical implementations of such schemes. However, such schemes are analogous to verifiable (conventional) threshold schemes, which have received substantial attention in the literature (see, e.g., [21]).

We abstractly capture the verification aspect of verifiable signature schemes and cryptosystems by postulating a predicate ok such that $ok(m, i, a_{m,i})$ is true iff (with a very high probability) $a_{m,i}$ is s_i's correct partial result for m. For instance, in the RSA implementation described in section 4, $ok(m, i, a_{m,i})$ would be true iff $a_{m,i} \equiv m^{K_i} \bmod N$. Similarly, when the RSA threshold cryptosystem is used to decrypt a request m from a client c_l as in section 5, $ok(m, i, a_{m,i})$ would be true iff $a_{m,i} \equiv m^{K_{i,l}} \bmod N_l$. We assume that ok can be evaluated efficiently (and in polynomial time) in our system model at all correct and honest servers. Depending on the implementation, this evaluation may require the servers to be given additional *a priori* information or to communicate additional information via existing protocol messages or additional phases of communication. Of

course, we also require that this additional information does not compromise the security of the threshold signature scheme or cryptosystem.

Given the predicate *ok*, our protocols can then be modified as described below. In the following descriptions, *rbcast* is a reliable broadcast primitive provided by servers' communication modules that replaces the stronger *abcast* primitive. (If $\mathcal{R}$ is used to implement $\mathcal{D}$ as described in section 3, however, then *abcast* must still be available to the servers.)

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

1. Execute *rbcast*$(a_{m,i})$, where $a_{m,i} \equiv m^{K_i} \bmod N$.
2. Wait to receive a set $\{a_{m,j}\}_{j \in T}$, $|T| = k$, of partial results for m such that for each $j \in T$, $ok(m, j, a_{m,j})$ is true.
3. Execute *respond*$(c, \langle m, A_{m,T} \rangle)$.

The routine *accept'*(m) remains the same.

Routine *request'*(m) at client c_l:

1. Create the RSA ciphertext $m' \equiv m^{e_l} \bmod N_l$ of m.
2. Execute *request*(m').

Routine *deliver'*$(\langle c_l, m \rangle)$ at server s_i:

1. Execute *rbcast*$(a_{m,i})$, where $a_{m,i} \equiv m^{K_{i,l}} \bmod N_l$.
2. Wait to receive a set $\{a_{m,j}\}_{j \in T}$, $|T| = k$, of partial results for m such that for each $j \in T$, $ok(m, j, a_{m,j})$ is true.
3. If $A_{m,T}$ is a valid request, then execute *deliver*$(\langle c_l, A_{m,T} \rangle)$.

It was already noted in section 4 that reliable broadcast could be used to disseminate partial results in the *respond'* routine. So, the key improvement resulting from the use of *ok* in that routine is the elimination of the (worst case) exponential search for a correct signature from step 2. Similarly, the exponential search to find a set of partial results that will properly decrypt a request has been eliminated from *deliver'*. Reliable broadcast can now be used to disseminate partial results in *deliver'* because all correct servers will use a set of k *correct* partial results to decrypt each request, and thus are no longer required to attempt decryption with the same sets of partial results. Also, clients no longer need to include message digests in their requests, again because correct servers will always use k correct partial results to decrypt each request. Finally and most importantly, Claims 5 and 6 can be strengthened respectively to:

Claim 8 *If correct* $\geq k$, *then if a correct client issues a request, it will be delivered at all correct servers. (That is, if correct* $\geq k$, *then the second half of Delivery Validity holds.)*

and

Claim 9 *This protocol satisfies the specification of atomic broadcast (for client requests) whenever correct* $\geq k$.

That is, liveness is guaranteed if *correct* $\geq k$. It is not difficult to verify that the other claims of sections 4 and 5 hold as stated.

Obviously the introduction of verifiable schemes would yield substantial benefits in our protocols. The investigation of possible verification algorithms is an important direction of ongoing research.

B The Isis security architecture

As mentioned in sections 1 and 6, the techniques developed in this paper are intended to be used in the Isis system, which is a toolkit for developing secure, highly reliable distributed applications. In this appendix, we outline the architecture of a new version of Isis that will incorporate the mechanisms of this paper, focusing on the Isis security architecture and the degree to which it supports the assumptions made in this paper.

The Isis project began in 1985 as an effort to develop a collection of tools for building highly reliable distributed software. Since its inception, the focus of the project has been on supporting loosely coupled distributed applications that must provide strong consistency guarantees despite component failures and recoveries. Put simply, Isis is oriented towards applications in which the actions taken by one system component must be coordinated with the actions taken by other system components. To this end, Isis provides to the programmer two basic building blocks: *virtually synchronous process groups* and *atomic, causally ordered group multicast.* Using these mechanisms, a number of higher-level tools are provided, including mechanisms for managing replicated data, synchronization, monitoring the status of system components, reconfiguring to recover from failures or overloads, parallel computation, primary-backup computation, and so forth. A still higher level of software provides highly reliable services, such as a fault-tolerant network file system, a network resource manager and load-balancing system, a wide-area messaging system, and a programming environment for instrumenting distributed applications and developing reliable reactive control systems. Isis has become widely popular and is currently used by hundreds of academic, commercial and military researchers and development efforts.

Initially, Isis did not address questions of distributed security. However, starting in 1991, an effort to reimplement the Isis system from the bottom up was initiated. Introduction of a comprehensive security architecture has emerged as one of the major thrusts of this activity, along with the adoption of a micro-kernel design philosophy and greater attention to performance and real time.

Until the present, the basic goal of this security work has been to guarantee the correct behavior of the Isis system abstractions in the face of various forms of malicious attack. Informally, a system

is viewed as a set of "islands" of trusted sites surrounded by potentially hostile agents to which the system provides services, but against which it must protect itself. In [22], we showed the feasibility of guaranteeing the integrity of the basic Isis services in such an environment, but under the assumption that within any process group, all member sites would be equally trusted and capable of initiating actions on behalf of the group as a whole. Thus, the presence of even a single corrupt site within a process group could lead to arbitrary security violations within that group. Despite the limitations of this model, we view the Isis security architecture as a significant advance in several respects; in particular, we believe the new Isis to be among the first systems capable of simultaneously assuring substantial degrees of both security and fault-tolerance. Implementation of the security architecture is now underway, and we expect to have detailed performance figures by early 1993.

It is our intention to implement the methods of the present paper as a security "tool" within this new version of Isis, in addition to using them to implement key Isis services. Such a tool would offer a number of practical benefits. First, it would facilitate the implementation of highly fault-tolerant services that could withstand the corruption of some server processes. Second, due to the feature that clients need not be able to identify or authenticate individual servers, service administrators could reconfigure services, possibly even changing the number and identity of servers, without affecting clients. This feature may also allow non-replicated services in existing systems to be transparently replaced with secure, replicated service implementations.

The implementation of these techniques in the Isis system does involve tradeoffs, however, because the system model supported by the Isis security architecture differs in important ways from that assumed in section 3. The primary difference was already mentioned in section 6, namely that the atomic broadcast protocol assumed in section 3 is resilient to more hostile failures than the Isis atomic broadcast protocol. Recall that we suggested one way to compensate for this difference, which involved protecting the Isis protocols within site operating systems via secure boot procedures and physical measures. This approach represents a compromise that we view as necessary for practicality (for the time being) and sufficient for most systems, albeit at a cost to the security of our solutions.

To summarize, our technique for securely replicating services has practical application in the Isis system, which is now being reimplemented to include a security architecture. This security architecture goes a long way toward supporting the assumptions enumerated in section 3, and with a few additional limitations on the failure modes that can be tolerated, will provide a viable platform for the implementation of the techniques described in this paper. We hope to report on our experience with the new Isis system, including the implementation of these techniques, within one year to eighteen months.