How to Securely Replicate Services*
(Preliminary Version)

Michael Reiter
Kenneth Birman

TR 92-1274
March 1992

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

*This work was supported by the Defense Advanced Research Projects Agency (DoD) under DARPA/NASA subcontract NAG 2-593 administered by the NASA Ames Research Center, by grants from GTE, IBM and Siemens, Inc., and by a National Science Foundation Graduate Fellowship. Any opinions, conclusions or recommendations expressed in this document are those of the authors and do not necessarily reflect the views, policies or decisions of the National Science Foundation or the Department of Defense.

How to Securely Replicate Services*
(Preliminary Version)

Michael Reiter
reiter@cs.cornell.edu

Kenneth Birman
ken@cs.cornell.edu

Department of Computer Science
Cornell University
Ithaca, New York 14853

March 24, 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 that fewer than k servers are corrupt and, to ensure liveness, that $k \leq n - 2t$, where t is the assumed maximum total number of both corruptions and benign failures suffered by 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.

*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* [23] to replicating the service, so that each server individually computes the result 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. In our method, 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 [20], 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, that fewer than k servers are corrupt, and, to ensure liveness, that $k \leq n - 2t$, where t is the assumed maximum total number of both corruptions and benign failures suffered by servers in any system run.

The above discussion may be evocative of the large body of literature providing solutions to various distributed computing problems in models where Byzantine failures can occur but authentication is possible (see [17, 24]). Nevertheless, our work has a somewhat different emphasis: we employ specific cryptographic techniques to achieve the aforementioned results, and in fact a significant 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.

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 [23]. 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. In section 6 we outline related work, and we conclude in section 7.

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 [16]. 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."

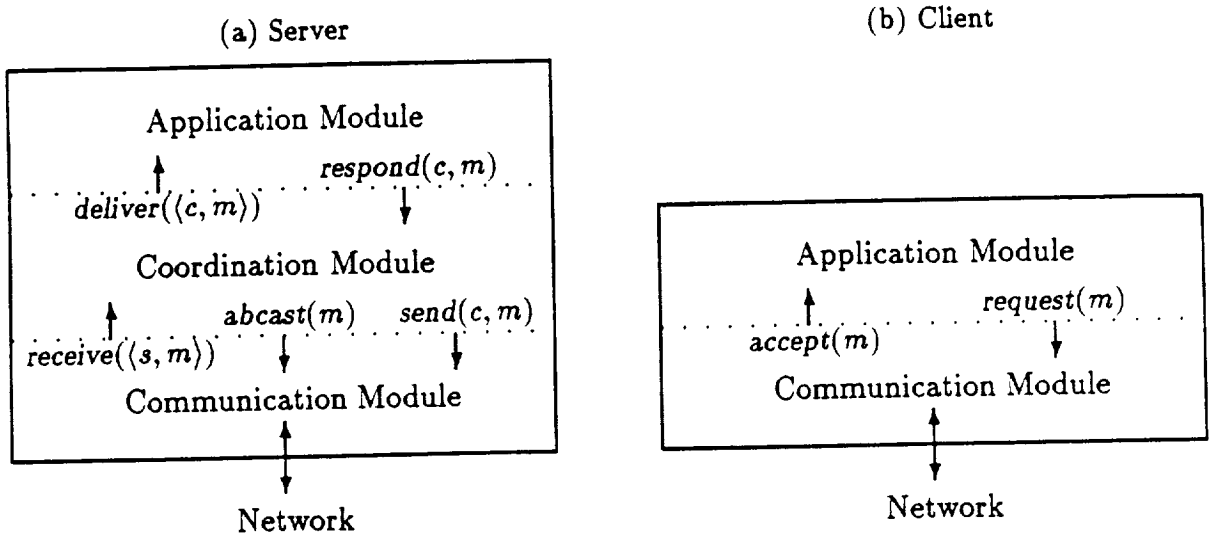
In order to capture the notions of an "accidental" failure versus a purposeful corruption by an intruder, we partition the failed principals into two sets: the *honest* principals and the *corrupt* principals. Formally, the only property that this partitioning must have is that any principal that ever suffers "truly Byzantine" failures—i.e., failures that cannot be classified as fail-stop, crash, omission, or timing failures—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.

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 $\langle c, m \rangle$ on 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 be used further in this paper. The second is an *atomic broadcast* primitive *abcast*(m) by which a server s can broadcast a message $\langle s, m \rangle$ to the other servers. Servers' coordination modules communicate exclusively through the use of this broadcast primitive. 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 is tolerant of $t < n$ server failures; henceforth we assume that a total of at most t servers fail in any system run. The protocol $\mathcal{R}$ 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.

Figure 1: Structure of principals



Receipt Validity: A correct server receives a message from a correct server iff the latter previously broadcast that message.

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]

There already exist protocols in the literature that satisfy this specification in various models and for various definitions of *honest*. For instance, if the honest principals are defined to include only those principals suffering crash failures, the system is synchronous, and the network is sufficiently connected, then the protocol described in [4] for Byzantine failures with authentication satisfies this specification for any choice of $t < n$. Moreover, Chandra [2] has developed randomized[2] protocols satisfying the above specification for the same definition of *honest* in an asynchronous system, by combining randomized solutions to consensus [3] and deterministic solutions to reliable broadcast [1]. The required relationship between n and t for these protocols depends on the underlying consensus and reliable broadcast protocols used. Our protocols do not rely upon any bounds on message

[1] It is shown in [12] 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 $n > 2t$.

[2] It is well-known that there is no deterministic solution to consensus, and thus atomic broadcast, in an asynchronous system that can suffer even a single crash failure [10].

transmission times, 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. By assuming that this request eventually reaches some correct server, and by having that server's communication module forward this request by executing *abcast*$(\langle c, m \rangle)$, the above specification of atomic broadcast can be made to hold for client requests. That is, we assume that servers also implement a protocol, called $\mathcal{D}$, satisfying 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 client iff the latter previously issued that request.

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.

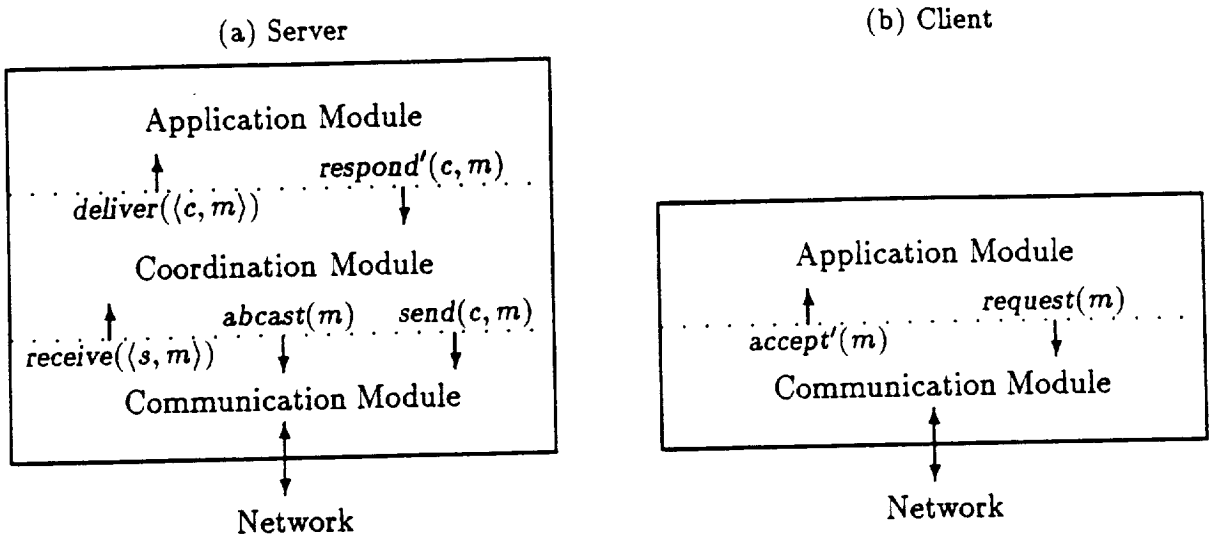
Integrity: If *corrupt* $< k$, then the response accepted at a correct client, if any, is that computed by a correct server.

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



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 is necessary to sign m, in the sense that without the private key it is 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 that a (k, n)-threshold signature scheme is *secure* if, informally, there is no feasible algorithm that, given some numbers of these items of information, can perform any of tasks (i)–(iii) above for some new message m. Note that to be secure, a signature scheme need *not* be able to tolerate attacks in which a cryptanalyst can see the partial results or the signature for any message *of its choice*, as would be possible in a *chosen message attack*. Such attacks can easily be prevented [5].

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]; a detailed understanding of this scheme is not essential. The scheme begins with an RSA [22] 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. 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 a proper signature for m.[3]

For our routines we assume that (the coordination module of) each server s_i is secretly given sole possession of K_i and any principal can reliably obtain the public key (e, N) of the service. We do not discuss how this distribution of private shares or public keys is accomplished, although we note that *all* public key systems require similar steps. The integers $p_{i,T}$ for all i and T can be "hardwired" into the implementation of the servers. The *respond'*(c, m) and *accept'*(m) routines are implemented as follows.

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

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

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

1. If m is not of the form $\langle m', S \rangle$, then return to the calling routine.
2. If S is a valid signature for m', then execute *accept*(m').[4]

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

Proof. If the signature scheme is secure and *corrupt* $< k$, then the corrupt servers cannot generate k partial results from which to sign a message. Thus, the only message that could be properly signed is that computed by a correct server. □

Claim 2 *This protocol satisfies Availability.*

Proof. Suppose that *correct* $\ge k$. 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. □

[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].

[4] Here we do not consider attacks on the *freshness* of responses [25].

In terms of communication complexity, in a failure-free run the replacement of *respond* with *respond'* results in an additional n executions of $\mathcal{R}$, 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 $t + 1$ servers need to be designated 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 appropriately weakening the Receipt Consistency requirement of atomic broadcast. Since reliable broadcast is weaker, it possibly can be implemented more efficiently. Reliable broadcast can be used because the order in which partial results are received by servers is not important in this protocol.

Potentially the most computationally expensive part of the algorithm 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 $l = \min\{n, k+t\}$ partial results received (from l unique servers), and at most $\binom{l}{k}$ subsets of partial results, because in l partial results are at least k correct partial results (if *correct* $\geq k$). While this could be expensive if l is large and $k \approx l/2$, the expected search time for a valid signature should be small in the common case in most systems, i.e., when n and *corrupt* are small. One optimization is to have a server always include its own index in T (i.e., always include its own partial result in the signature computation). Also, certain heuristics, such as using partial results from a combination of servers that previously worked, can be used to further reduce the expected search time. Additional optimizations are a topic of ongoing research.

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.

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 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 [23]. 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 see why this may be important, 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. It is worth noting that 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'*.

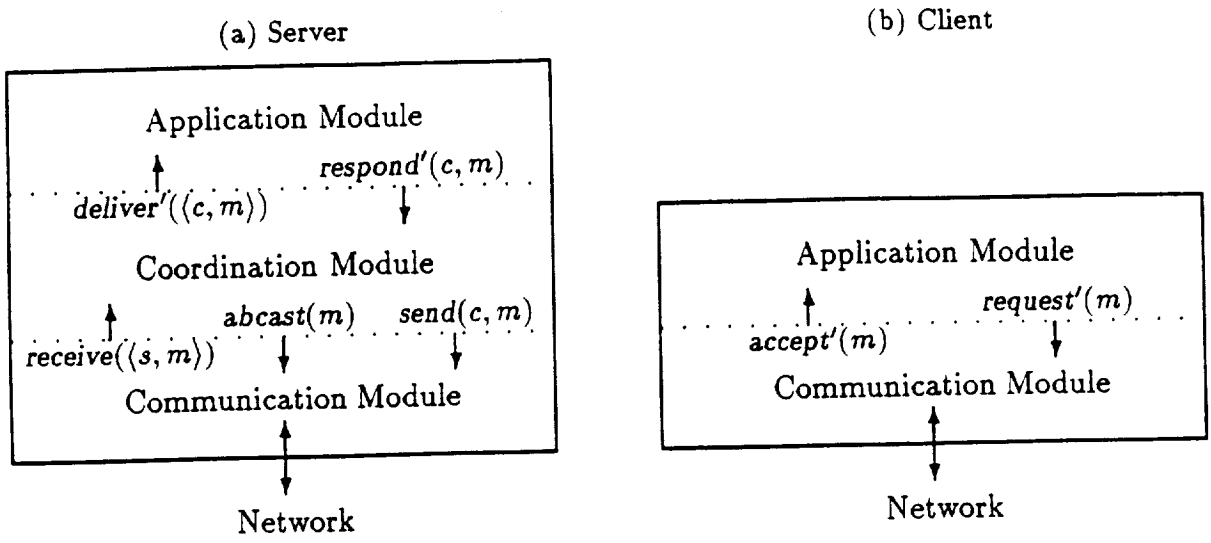
In the implementation of *request'*(m), the correct client c encrypts m under a public encryption key of the service before issuing $\langle c, m\rangle$. Then, c is provided the following guarantee. The reader should verify that this guarantee prevents the aforementioned problem, provided that *corrupt* $< k$.[5] (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 the sending client), 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 Validity, Delivery Order, and Delivery Consistency still hold. The implementations of *request'* and

[5] Here we do not consider *traffic analysis* attacks [25] or attacks that exploit the *malleability* of the cryptosystem [8].

Figure 3: New structure of principals



deliver' that we propose satisfy all but the "if" direction 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 $k \leq n - 2t$, and thus $n > 2t$.

Our *deliver'* routine employs 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 is necessary to decrypt m, in the sense that without the private key it is 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 may involve the use of partial results and some number of shares, in addition to plaintext/ciphertext pairs. For our purposes, we will say that a (k, n)-threshold cryptosystem is *secure* if, informally, there is no feasible algorithm that, given some numbers of these items of information, can perform any of tasks (i)–(iii) above for some new ciphertext m. Again we point out that to be secure, a threshold cryptosystem need *not* be able to tolerate attacks in which a cryptanalyst can see the partial results or the plaintext for any ciphertext of its choice, as would be possible in a *chosen ciphertext attack.* This is in accordance with the security of all implementations of threshold cryptosystems thus far proposed: all proposed implementations are known to be vulnerable to chosen ciphertext attacks, because the conventional cryptosystems on which they are built are vulnerable to such 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, 14].

Suppose that we are using the RSA threshold cryptosystem described above and that we have the initial conditions assumed in the previous section; 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 algorithm 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* $< k$, 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 algorithm 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., $x \equiv y^d \bmod N$.[6]

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

[6]The obvious simplified form of this attack, in which x and y are chosen so that $x \equiv y \equiv 1 \bmod N$, 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 would only expedite the delivery of the correct client's request.

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, 14].

Therefore, until a practical threshold cryptosystem is designed that can tolerate chosen ciphertext attacks, such attacks must be prevented. A simple way to do this is to have a separate public key for each client. That is, the public key for client c_j would be a pair (e_j, N_j), and each server s_i would be given a share $K_{i,j}$ for use when cooperating to decrypt a request from client c_j. 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. In practice, having separate cryptosystem parameters for each client may require that each client be individually "registered" with the service outside of the system before employing the service. While this could limit the settings in which our protocols can be used, we believe that this is not an unreasonable restriction if maintaining causality among client requests is of such importance.

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 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 the 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.[7] Several efficient implementations of message digest functions have been proposed (e.g., [18, 21]) but will not be discussed here. Let $digest(m)$ denote the message digest of m.

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

[7] 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.

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

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

Routine *deliver'*$(\langle c_j, m \rangle)$ 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 $a_{m',i} \equiv (m')^{K_{i,j}} \bmod N_j$.
3. Wait until the first $k+t$ partial results $\{a_{m',i'}\}_{i' \in T}$, $|T| = k+t$, for m' have been received (from $k+t$ 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*$(\langle c_j, A_{m',T'} \rangle)$.

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 $k+t$ partial results for m' are received at correct servers, then the request will never be delivered at any correct server. So, suppose that $k+t$ partial results for m' are received at all correct servers. Because partial results are broadcast atomically, all correct servers employ precisely the same set $\{a_{m',i'}\}_{i' \in T}$, $|T| = k+t$, of partial results when attempting to decrypt m'. Therefore, one 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 servers execute the same sequence of *deliver'* calls. And, because any call to *deliver'* returns before *deliver'* is called again, it follows from Delivery Atomicity that all 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 $\{a_{m',i'}\}_{i' \in T}$, $|T| = k+t$, 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 requests delivered at a correct server. □

Claim 4 *If a request is delivered from a correct client at a correct server, then it was issued by that client. (That is, the "only if" direction of Delivery Validity holds.)*

Proof. By Delivery Validity of $\mathcal{D}$, if *deliver'*$(\langle c, m\rangle)$ is called at a correct server and c is correct, then c must have issued this request. Therefore, a request from a correct client can be delivered at a correct server only if the client issued that request. □

Claim 5 *If $k \leq n - 2t$, then if a correct client issues a request, it will be delivered at all correct servers. (That is, if $k \leq n - 2t$, then the "if" direction of Delivery Validity holds.)*

Proof. By Delivery Validity of $\mathcal{D}$, *deliver'* is called exactly once for each request $\langle c, m\rangle$, $m = \langle m', D\rangle$, issued by a correct client c. If $k \leq n - 2t$, then $k + t \leq n - t \leq$ *correct*, and so at least $k + t$ partial results for m' are broadcast and, therefore, received at all correct servers. Since the set of $k + t$ partial results used at each correct server contains at least k partial results from k correct servers, m' can be decrypted and delivered. □

From the above claims, we immediately have the following.

Claim 6 *If $k \leq n-2t$, this protocol satisfies the specification of atomic broadcast (for client requests).*

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 $k+t$ 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. □

In a failure-free run, the replacement of *deliver* with *deliver'* results in an additional n executions of $\mathcal{R}$, 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) and $t + 1$ responses, structured in four communication phases. And, a client may need two public keys for the service, depending on the particular cryptosystem and signature scheme used.

As in the protocol of section 4, step 4 of *deliver'* is potentially expensive, because it may require a server to sort through $\binom{k+t}{k}$ subsets of partial results to be able to decrypt a request. In fact, a

corrupt client can *force* each server to sort through $\binom{k+t}{k}$ subsets by sending a ciphertext and digest that do not "match." As before, we rely on heuristics, a small $\binom{k+t}{k}$, and a small *corrupt* in the common case to reduce the expected cost of decrypting requests from correct or honest clients. A bad request from a corrupt client can be detected in a reasonable amount of time if $\binom{k+t}{k}$ is small, and then subsequent requests from that client can be ignored.

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 Related Work

This work was largely inspired by [11], 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 and has the additional advantage that, unlike the service described in [11], 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 [15]. Since then, other authors have concentrated upon secure replication of data. Secure data replication using quorum methods is considered in [13] for the case in which both data integrity and secrecy are important. In these schemes, however, an intruder that successfully 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 [19], 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.

7 Conclusion and Future Work

We have presented a method for securely replicating services using the state machine approach. Using our protocols, 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 at least k servers are correct and fewer than k servers are corrupt. We have also addressed the issue of ensuring causality among client requests. A security breach resulting from an intruder's ability to violate causality was illustrated, and a safe and live approach was presented to counter this problem,

provided that $corrupt < k \leq n - 2t$. 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.

In addition to those topics of ongoing research mentioned in the previous sections, another direction of research is ways to employ the techniques described here in a hierarchical fashion to enhance the security of applications. As a simple example of this, one could conceivably employ a different replicated service to produce each *partial result* for a message, and then these partial results could be combined to either sign or decrypt this message appropriately. However, the consequences and benefits of such designs have not yet been fully investigated and will be discussed further elsewhere. Another topic that has not been sufficiently studied is how the *detection* of corrupt clients and servers can be achieved and exploited to optimize our protocols.

Acknowledgements

This work benefited tremendously from many discussions with Yair Frankel (University of Wisconsin at Milwaukee). Tushar Chandra (Cornell University) also contributed several interesting discussions, and Sam Toueg (Cornell University) provided helpful comments and information. Cliff Krumvieda (Cornell University) commented on an early draft of this paper.

References

[1] Bracha, G., and Toueg, S. Asynchronous consensus and broadcast protocols. *Journal of the ACM 32*, 4 (Oct. 1985), 824–840.

[2] Chandra, T. D., Feb. 1992. Private communication.

[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 *Proceedings of CRYPTO '89* (Aug. 1989), pp. 307–315. Published as *Lecture Notes in Computer Science 435*.

[7] Desmedt, Y., and Frankel, Y. Shared generation of authenticators and signatures. In *Proceedings of CRYPTO '91* (1991).

[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 concensus with one faulty process. *Journal of the ACM 32*, 2 (Apr. 1985), 374–382.

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

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

[13] HERLIHY, M. P., AND TYGAR, J. D. How to make replicated data secure. In *Proceedings of CRYPTO '87* (Aug. 1987), pp. 379–391. Published as *Lecture Notes in Computer Science 293*.

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

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

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

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

[18] MERKLE, R. C. A fast software one-way hash function. *Journal of Cryptology 3*, 1 (1990), 43–58.

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

[20] 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).

[21] RIVEST, R. L. The MD4 message digest algorithm. Internet RFC 1186, Oct. 1990.

[22] 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.

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

[24] SRIKANTH, T. K., AND TOUEG, S. Simulating authenticated broadcasts to derive simple fault-tolerant algorithms. *Distributed Computing 2* (1987), 80–94.

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