Project IST-1999-11583

**Malicious- and Accidental-Fault Tolerance
for Internet Applications**

Full design of Dependable Third Party Services

Christian Cachin (editor)

IBM Research, Zurich Research Laboratory

MAFTIA deliverable D5

Public document

28 February 2002

Research Report RZ 3394, IBM Research, Zurich Research Laboratory

Editor

Christian Cachin

Contributors

Christian Cachin
Klaus Kursawe
Jonathan A. Poritz

Address of all authors:

IBM Research
Zurich Research Laboratory
Säumerstr. 4
CH-8803 Rüschlikon
SWITZERLAND
`http://www.zurich.ibm.com/`

Contents

Abstract

This document describes the designs of a generic distributed certification authority and of a trusted party for optimistic fair exchange that are based on fault-tolerant middleware for service replication. It also discusses other uses of the replication middleware for implementing trusted services. It may serve as a blueprint for building generic trusted third-party services that use the state-machine replication approach.

1 Introduction

Distributed systems running in error-prone and adversarial environments must rely on trusted components, such as secure directories, name and authorization services, or certification authorities. Building centralized trusted services has turned out to be a valuable design principle for computer security because the trust in them can be leveraged to many, diverse applications that all benefit from centralized management. But centralization introduces a single point of failure. Even worse, it is also difficult to protect any single system against the sort of attacks found on the Internet today.

The fault tolerance of centralized components can be enhanced by distributing them among a set of servers and by using replication algorithms for masking faulty servers. Thus, no single server has to be trusted completely and the system derives its integrity from a majority of correct servers. Our approach builds on fault-tolerant protocols for dependable secure service replication. It uses active replication [20], which works for any service that is implemented by a deterministic state machine.

This work gives the detailed specification of a *distributed certification authority* based on the state-machine replication method and on a threshold signature scheme (Section 3), and the specification of a *distributed trusted party for optimistic fair exchange* (Section 4).

It also reviews other applications to trusted services that MAFTIA's distributed state-machine replication protocols have already found (Section 5).

Together with the specifications of further trusted services found in the previous MAFTIA deliverable on the subject [5], they form the basis for realizing generic trusted services with the state-machine replication method.

Detailed descriptions of the relevant parts of the MAFTIA middleware for asynchronous group communication are specified in D26 [5] and in D24 [22]. Because the updated protocol and API specifications as used here are not available as a MAFTIA deliverable at the time of this writing, we refer to the technical report [8] that documents the parts to be used in the context of secure service replication.

2 System Model

We briefly recall the system model for our distributed trusted service from [5].

Our approach is based on protocols for secure state-machine replication and coordination among a group of servers connected by a wide-area network, such as the Internet. These protocols are described in [5, 22].

The trusted service is implemented by a static group of n *servers*, of which up to t may fail. They are connected by reliable asynchronous point-to-point links and have no access to a common clock. Faulty servers can fail in arbitrary, malicious ways and are called *corrupted*, the remaining ones are called *honest*. Our replication protocols work under the assumption that $n > 3t$, which is optimal for an asynchronous network with malicious faults.

The servers are connected only by asynchronous point-to-point communication links and do not have access to synchronized clocks. Thus, our approach automatically tolerates timing failures as well as all attacks that exploit timing. The group model is static, which means that failed servers must be recovered by mechanisms that are currently outside of the architecture.

We need a trusted dealer to generate the secret keys for a particular configuration of the group. The dealer is needed only once, when the system is initialized. The keys must be distributed to all servers in a trusted way. Our reason for introducing a trusted dealer is that no efficient protocols for generating all necessary cryptographic keys in a distributed fault-tolerant way are currently known.

The description here usually assumes that up to a certain fraction of all servers fail. This implies a threshold failure model, which is appropriate for independent failures, but not for maliciously induced faults. However, our protocols are not limited to threshold failure assumptions. As shown in [5], it is possible to use *generalized adversary structures*, which are also adequate for faults that represent malicious acts of an adversary. They can accommodate a strictly more general class of failures than any weighted threshold failure assumption.

In addition to the servers, there is an unspecified number of clients, which may be corrupted by an adversary well.

3 A Certification Authority

This section describes the *MAFTIA distributed certification authority* or *DCA* for short. *DCA* does not store its secret signing key at a single location, which might be compromised by an attacker. Instead, it uses threshold cryptography and secure replication protocols to distribute the power of issuing a certificate among a group of servers, which may only be connected by an asynchronous network like the Internet. *DCA* issues certificates for encryption public keys and for digital signature (verification) public keys, for encryption schemes that are secure against adaptive chosen-ciphertext attacks [17] and for digital signature schemes that are unforgeable against adaptive chosen-message attacks [13].

This chapter describes the components of the *DCA* (Section 3.1), how clients interact with the *DCA* (Section 3.2), how the validity of certification requests is determined (Section 3.3), and how the *DCA* is implemented (Section 3.4).

3.1 Components

DCA runs on a group of n servers, of which up to $t < n/3$ may be faulty or corrupted by a malicious adversary. They are linked by an asynchronous point-to-point network (providing authenticated asynchronous channels). The servers may also communicate with clients over asynchronous channels, but clients are anonymous.

In the context of the coordination protocols used by *DCA*, the servers do not have access to synchronized clocks and no assumptions are made about their relative speed of execution. Thus, they operate in a fully asynchronous network, which rules out all time-based attacks on the protocol level.

Of course, the servers may need access to a common notion of time in order to determine the validity period of a certificate. But this presumably needs much less accuracy than clocks for synchronizing network protocols, so that this assumption does not invalidate the assumption of an asynchronous communication network.

DCA needs to be initialized in a trusted way. This means that an administrator generates the signing key of *DCA* on a central server to obtain n initialization files. These files must then be copied to the servers in a secure way (e.g., by using manual distribution via a floppy disk, by encrypted email, or through an encrypted login session). The *DCA*'s public key should be made available to all users.

When the *DCA* thread is started on every server, the *DCA* starts to operate. It will accept requests from clients to issue and update certificates as described in Section 3.2. The validity of requests for certificates is determined according to the *certificate issuing* *mode* described in Section 3.3.

3.2 Client Operations

Clients access the *DCA* to *issue* a new certificate, to *update* an existing certificate, to *revoke* an existing certificate, or to *retrieve* one or more certificates. Clients are assumed to know the *DCA* public key and the identities of all n *DCA* servers; they can communicate with all servers through an asynchronous communication network. Each *DCA* operation is described below.

3.2.1 Issuing a new certificate

A certificate issue request is sent by a client who wants to obtain a new certificate on a particular public key that it controls (i.e., for which it knows the corresponding secret key) and a particular name. In single-server certification authorities, the validity of such a request is usually determined by a trusted application, which may also include a "registration authority." Because this is not possible in our distributed model, where some of the *DCA* servers may be corrupted themselves, a *key confirmation* step is mandatory for every certificate issue and update request.

To start issuing a new certificate, the client sends the message

$$(\texttt{issue-request}, \mathit{rid}, \mathit{key}, \mathit{name}, \mathit{credentials})$$

to one or to a quorum of $2t + 1$ *DCA* servers (the difference between contacting one and a quorum of $2t + 1$ servers is explained in Section 3.2.5). The string *rid* is the *request identifier*, which should be chosen as a unique value by the client. *key* is an encryption or signature public key, *name* contains the name and attributes to be bound to the key, and *credentials* contains any further information that may be necessary to determine the validity of the request.

If the *DCA* has determined that the certificate issue request is valid, the client will receive several (at least $t + 1$) messages of the form

$$(\texttt{issue-challenge}, \mathit{rid}, s)$$

from distinct *DCA* servers, where s is an arbitrary-length bit string. If *key* is an encryption public key, the client must interpret s as a ciphertext, decrypt it using the corresponding secret key, and return the decryption d in a message

$$(\texttt{issue-answer}, \mathit{rid}, d)$$

to the server from which it received the `issue-challenge` message. If *key* is a signature verification key, the client must sign s using the corresponding signing key and return the resulting signature d in an `issue-answer` message.

This step proves to the *DCA* servers that the client controls the secret key belonging to the public key on which it requests a certificate. Although it is not a proof of knowledge in the technical sense [12] and it is not clear what the implications of the verification for an application are, in our context this protocol serves to authorize the certificate request for the *DCA*, which is distributed. (We do not want to invoke a non-distributed "registration authority" that determines which keys are bound to which names as this could become a single point of failure.) In any case, if a client could forge correct answers to the requests without having access to the secret key, both the encryption scheme and the signature scheme would be considered insecure.

The client then waits until it receives a message

$$(\texttt{issue-cert}, \mathit{rid}, \mathit{cert})$$

where *cert* contains a valid certificate under the *DCA*'s public key on the *key* and *name* supplied in the `issue-request` message.

3.2.2 Updating a certificate

A certificate update request is sent by a client who wants to change the binding of a *key* to a *name*, which was established in one or more certificate(s) issued previously by the *DCA*. Again, the client must prove to the *DCA* that it controls the corresponding secret key in a mandatory *key confirmation* step.

To start updating a certificate, the client sends the message

$$(\texttt{update-request}, \mathit{rid}, \mathit{key}, \mathit{name}, \mathit{credentials})$$

to one or to a quorum of $2t + 1$ *DCA* servers (the difference between contacting one and a quorum of $2t + 1$ servers is explained in Section 3.2.5). The string *rid* is the *request identifier*, which should be chosen as a unique value by the client. *key* is an encryption or signature public key, *name* contains the name and attributes to be bound to the key, and *credentials* contains any further information that may be necessary to determine the validity of the request.

If the *DCA* has determined that the certificate update request is valid, the client will receive several (at least $t + 1$) messages of the form

$$(\texttt{update-challenge}, \mathit{rid}, s)$$

from distinct *DCA* servers, where s is an arbitrary-length bit string. If *key* is an encryption public key, the client must interpret s as a ciphertext, decrypt it using the corresponding secret key, and return the decryption d in a message

(`update-answer`, *rid*, d)

to the server from which it received the `update-challenge` message. If *key* is a signature verification key, the client must sign s using the corresponding signing key and return the resulting signature d in an `update-answer` message.

This step proves to the *DCA* servers that the client controls the secret key belonging to the public key on which it requests a new certificate.

The client then waits until it receives a message

(`update-cert`, *rid*, *cert*)

where *cert* contains a valid certificate under the *DCA*'s public key on the *key* supplied in the `update-request` message.

Note that the new certificate resulting from an update action (or from an issue request) is self-verifying since the client can determine itself if its request has been executed properly.

3.2.3 Revoking a certificate

A certificate revocation request is sent by a client who wants to remove the binding of a *key* to a *name*, which was established in one or more certificate(s) issued previously by the *DCA*.

To revoke a certificate, the client sends the message

(`update-request`, *rid*, *key*, *name*, *credentials*)

to one or to a quorum of $2t + 1$ *DCA* servers (the difference between contacting one and a quorum of $2t + 1$ servers is explained in Section 3.2.5), where *name* may be left empty in *key-major* mode and *key* may be left empty in *name-major* mode. The string *rid* is the *request identifier*, which should be chosen as a unique value by the client. *key* is an encryption or signature public key, *name* contains name and attributes, and *credentials* contains any further information that may be necessary to determine the validity of the request.

3.2.4 Retrieving certificates

DCA stores all valid (i.e., non-updated and non-revoked) certificates. A client who wants to obtain a certificate for a particular public key or a particular name may do this using the following protocol for retrieving certificates.

The client sends a message

$$(\texttt{retrieve}, \mathit{rid}, \mathit{key}, \mathit{name})$$

to one or to a quorum of $2t + 1$ *DCA* servers (the difference between contacting one and a quorum of $2t+1$ servers is explained in Section 3.2.5). The string *rid* is a *request identifier*, which should be chosen as a unique value by the client, and either *key* or *name* may be empty, but not both. The idea is that a client can retrieve all certificates for a given *key*, for a given *name*, or for a given combination of *key* and *name*.

The client then waits to receive at least $2t + 1$ messages of the form

$$(\texttt{retrieve-cert}, \mathit{rid}, \ell, \mathit{cert}_1, \mathit{cert}_2, \ldots, \mathit{cert}_\ell)$$

from distinct *DCA* servers such that their *content is the same.* $\mathit{cert}_1, \mathit{cert}_2, \ldots, \mathit{cert}_\ell$ are all certificates known to *DCA* pertaining to the supplied *key* or *name*.

It is necessary for the client to obtain $2t + 1$ consistent answers from different *DCA* servers because the client must be sure that the answers represent the current contents of the certificate database. Note that it may be the case that up to t servers are corrupted and send back invalid, outdated, or incomplete results. Moreover, also up to t honest servers may send incomplete answers because they are slow and have not yet completed the most recent update operation(s). Thus, their answers may contain certificates that have been revoked or updated by the honest majority in the mean time. But receiving $2t+1$ answers that agree on their contents ensures that the answer was also sent by at $t+1$ honest servers, at least one of them knows the current state of the *DCA*, and therefore the answer is consistent.

It is interesting to note that only the certificate retrieval operation needs this extra redundancy. The certificate issuing and update operations, in contrast, have self-verifying results since the client can determine itself if a request has been executed properly.

3.2.5 Contacting one or a quorum of $2t + 1$ servers

As mentioned above, a client may send a request to one *DCA* server or to a quorum of $2t+1$ servers. These are two different modes of contacting the *DCA* and their difference is as follows.

In the first case, where a request is sent to only one *DCA* server, this server becomes responsible for broadcasting the request to the group of all n *DCA* servers. Should this server crash or become corrupted, it is possible that the client's request is lost and never processed by *DCA*. In order to prevent this, we require that a client, who has not received any answer to a request sent previously, must resend this request to a *different DCA* server with the same *rid*. How long a client may wait before "timing out" is left to the particular application. If a client proceeds like this but receives no answer for an extended period of time, it will have sent the request to at least $2t + 1$ *DCA* servers, which brings us to the second case.

In the second case, the server has sent the request to at least $2t + 1$ different *DCA* servers. The servers are running an atomic broadcast protocol to disseminate the requests, which guarantees fair delivery of a request only if at least $t+1$ honest servers *send* it [5, 22]. But by the assumption on the number of failures, at least $t+1$ of the servers who receive the request are honest, which ensures that it is eventually delivered by the atomic broadcast and processed by *DCA*. The *DCA* servers identify requests through the combination of the name of the client (e.g., its IP address) and the *rid* chosen by the client. The *DCA* implementation must filter out duplicate requests with the same identification.

3.3 Certificate Issuing Modes

DCA handles certificate update requests in one of two modes: *key-major* or *name-major* mode. They correspond to the difference between binding a name to a key and binding a key to a name.

For the purpose of this discussion, assume *key* and *name* parameters in requests to *DCA* are represented by Java objects of type

```
public interface Key implements java.lang.Comparable
```

and

```
public interface Name implements java.lang.Comparable
```

respectively. In other words, *DCA* knows how to compare and to sort keys and names.

In *key-major* mode, an update request binds the new *name* to an existing *key* and invalidates any previously issued certificate that may have bound a different name to *key*. The validity of such a request is determined using the *credentials* supplied with the request.

This corresponds to the example of a public key (for encryption or signatures) that a user posts on his or her web page, when the user obtains a new email address.

More precisely, when a valid update request is processed by a *DCA* server, it removes all those certificates from its certificate store that match *key* using the operations in `java.lang.Comparable`. Then it signs the new binding of *name* to *key*, stores the resulting certificate, and also sends it to the client.

In *name-major* mode, conversely, an update request binds a new *key* to an existing *name* and invalidates any previously issued certificate that may have bound a different key to *name*. The validity of such a request is determined using the *credentials* supplied with the request. This corresponds to a public key associated with an email address that is posted on a web page, and where a new public key is generated and bound to the existing address in case the owner believes the old key has been compromised.

More precisely, when a valid update request is processed by a *DCA* server, it removes all those certificates from its certificate store that match *name* using the operations in `java.lang.Comparable`. Then it signs the new binding of *key* to *name*, stores the resulting certificate, and also sends it to the client.

3.4 Implementation

The certificates issued by *DCA* are based on RSA signatures [19]. The current format is proprietary, but future extensions will optionally allow to produce standard X.509-type certificates. Certificates are produced as threshold signatures using the scheme of Shoup [21, 5, 22, 8].

Every *DCA* server has a certificate store that contains all valid certificates (i.e., all certificates that have not been revoked or updated).

The initialization data of every server contains a key share of an $(n, t+1)$-threshold signature scheme $\mathcal{S}$.

The *DCA* servers use an *atomic broadcast* protocol to distribute all requests that are sent to *DCA* [5, 22, 8]. The common atomic broadcast channel is started when the servers have been initialized and begin to operate.

The only payload carried by the atomic broadcast channel are client requests. The details of the atomic broadcast implementation used by *DCA* can be found in [5]. This protocol requires the dealer to generate a key pair of a digital signature scheme for every party and to include the public keys of all parties in the initialization data.

The atomic broadcast channel implementation uses the sender's identity and a separate sequence number for each sender for identifying the payload messages. This is in contrast to the abstract protocol description in [5], where payloads are identified by their bit-string representation. This change seems unavoidable for any reasonably efficient implementation since one would otherwise have to store a complete history of payloads. Thus, the abstract *integrity* property [5] that every payload message is delivered at most once, no matter which parties sent it, has to be changed for *DCA*: Here it means that a message, consisting of a bit string, is delivered at most once for every time that an honest party sent that bit string. See also [8] for a discussion of this issue.

Because of this fact, *DCA* must include an additional layer to exploit the fairness property while maintaining integrity, i.e., so that more than one server (i.e., $2t+1$ servers) may send the same client request on the broadcast channel but the request is executed at most once. This extra layer on top of atomic broadcast filters out the duplicate client requests. Every *DCA* server maintains a history of the client requests that have been delivered on the atomic broadcast channel. (In practice, it will be necessary to limit the size of this history, but the bigger it can grow, the better the performance of the protocol.)

When a *DCA* server receives a request from a client, it first checks if the request is already present in the history of delivered client requests (using the client identity together with *rid* to identify requests in the history). If yes, it discards the request; otherwise, it sends it on the atomic broadcast channel.

When a request is delivered on the atomic broadcast channel, a *DCA* server also checks if the request is already present in the history of delivered client requests. If yes, the server discards the request. If not, the server adds the request to the history and processes it.

Requests are processed as described next.

3.4.1 Processing issue and update requests

Suppose an issue or update request r for *key* and *name* is ready to be processed and assume that *DCA* operates in *key-major* mode (the operation is analogous for *name-major* mode). Then the server marks all certificates in the certificate store that contain *key* as "in transition" and any future client request pertaining to *key* is buffered until r has been processed.

At this time, the server spawns a separate thread to handle request r and may continue to process requests that are delivered on the atomic broadcast channel.

The server uses the current *policy* and the *credentials* supplied with r in order to determine if the request should be processed. If yes, the server sends the appropriate `challenge` request to the client (i.e., either an encryption under *key* of a randomly chosen nonce or a random nonce). Then it waits until the client returns the suitable `answer` message and checks the validity of the answer using *key*.

Next the server starts a binary Byzantine agreement protocol [5, 22] to determine if r should be fulfilled or not. The binary agreement instance is identified by client's identity and by the *rid* from r. The initial vote of the server is set to "yes" if and only if both of the above tests were valid (i.e., r satisfied the *policy* and the client answered the challenge in the correct way).

The outcome of the binary agreement protocol determines how the server proceeds. If the outcome is "no," the server unmarks all certificates in the certificate store that contain *key* (those marked "in transition" above). Then it terminates the processing of r.

If the outcome is "yes," the server generates a share of a threshold signature for the requested certificate and sends it to all *DCA* servers. Then it waits for receiving enough $(t+1)$ shares of the signature from other servers, and assembles the certificate once it gets them.

If r is an update request, the server removes all certificates in the certificate store that contain *key* (those marked "in transition" above).

In any case, the server then adds the new certificate to its certificate store and sends the new certificate in a message of type `cert` to the client. This terminates the processing of r.

All requests that may have been buffered because some certificates related to r were marked as "in transition" are processed next. They are processed in the same order in which they have been queued up.

3.4.2 Processing a revocation request

Suppose a certificate revocation request r for *key* (and perhaps *name*) is ready to be processed and assume that *DCA* operates in *key-major* mode (the operation is analogous for *name-major* mode).

If the server is concurrently processing a request containing *key* (and has marked all certificates containing *key* "in transition" as described above), then the request is buffered and processed later.

Otherwise, the server uses the current *policy* and the *credentials* supplied with r in order to determine if the request should be processed. If yes, it removes all certificates from the certificate store that contain *key* (and perhaps *name*, if *name* is not empty).

3.4.3 Processing a retrieval request

Suppose a retrieval request r for *key* and/or *name* is ready to be processed. Recall that retrieval requests contain at least a value for either *key* or *name*. Assume further that *DCA* operates in *key-major* mode (the operation is analogous for *name-major* mode).

If the server is concurrently processing a request containing *key* (and has marked all certificates containing *key* "in transition" as described above), then the request is buffered and processed later.

Otherwise, the server proceeds to answer the request immediately. No separate thread is started for this operation. All certificates matching *key* and/or *name* are retrieved from the certificate store. If only *name* is supplied in r, the certificates that match *name* and are marked as "in transition" may either be included as well or may be ignored (this choice is implementation-dependent). The resulting certificates $cert_1, cert_2, \ldots, cert_\ell$ are returned to the client in a `retrieve-cert` message.

3.5 Discussion

This specification does not ensure that a client who sends an invalid or even a bogus request maintains liveness. In fact, such a client may be waiting forever for an answer from *DCA*. But any implementation would most likely include additional answer messages with suitable error codes.

The only other proposal for a distributed certification authority that we are aware of is COCA [23], the Cornell On-line Certification Authority. It does not use atomic broadcast but imposes an application-specific ordering for update requests that modify certificates for the same key.

COCA does not have the flexibility of *DCA* for configuring its operation mode, but it executes potentially faster because its protocols are less involved. Another difference is that COCA may execute certificate update and retrieval operations concurrently and therefore a certificate query operation sometimes does not return the most recent certificate. In contrast, *DCA* serializes update and retrieval operations through atomic broadcast such that retrieval operations always return the most current certificate.

4 A Trusted Party for Optimistic Fair Exchange

This section describes the *MAFTIA distributed optimistic fair exchange service* or *DFE* for short.

The *fair exchange problem* lies at the basis of may commercial interactions between two parties: how the participants can exchange two valuable tokens in such a way that either both get the item they bargained for or neither does. Many protocols have appeared in the literature to solve this problem, and they all use the mechanism of a trusted third party in some way (at least all potentially practical protocols do so). Perhaps the most efficient algorithms are those which go under the name of *optimistic fair exchange* [1], where the third party is only involved when the transaction fails, either to abort a transfer when the initiating party is not releasing her valuable item, or to force a conclusion of the transaction if the first party has released her good but the second is trying to avoid the promised payment—or simply if some of the protocol messages are lost or deleted by a malicious network.

DFE implements the third party by a group of servers of which some might be corrupted themselves and collaborate with corrupted clients. *DFE* uses a distributed signature scheme and secure coordination protocols to tolerate such faults.

This chapter describes first the fair exchange protocols used by *DFE* (Section 4.1), the components of *DFE* (Section 4.2), how clients interact with *DFE* (Section 4.3), and how *DFE* is implemented (Section 4.4). There are some modifications with respect to the preliminary description of *DFE* in [5], which are explained below.

4.1 Protocols

The fair exchange protocol used by *DFE* is the asynchronous protocol proposed by Asokan, Shoup and Waidner [2]. It provides an exchange of digital signatures. This protocol has the advantage of being extremely flexible, so that it can operate on all commonly used signature schemes and can be easily adapted for the exchange of digital content or certified e-mail, for example. The communication model used in [2] is the same as used here, relying only on asynchronous communication on an untrusted network. We shall describe here the special case from [2] of a protocol for the electronic signing of contracts, because it is one that can very easily take advantage of the communication primitives we have developed.

Let us denote by $[\alpha]_X$ the bit string α concatenated with a signature on α under X's public key. Then the protocol for optimistic fair exchange of digital signatures on a contract m between two parties A and B, with dispute resolution by the trusted party T, proceeds as follows:

1. A sends B the message
$$(\texttt{promiseA}, [m, A, B, T]_A)$$
2. B receives the message and verifies the signature; if successful, it replies with
$$(\texttt{promiseB}, [m, A, B, T]_B),$$
otherwise B quits;
3. A receives and verifies this message; if successful, it sends to B the message
$$(\texttt{commitA}, \sigma_A),$$
where $\sigma_A = [m, A, B]_A$, otherwise A requests an *abort* from T;
4. B receives the message containing σ_A from A and verifies the signature; if successful, it sends to A the message
$$(\texttt{commitB}, \sigma_B),$$
where $\sigma_B = [m, A, B]_B$, and accepts the pair (σ_A, σ_B) as the exchange contract; otherwise B requests a *resolve* from T;
5. A receives this message and checks the signature from B; if successful, it accepts the pair (σ_A, σ_B) as the exchange contract; otherwise, A requests a *resolve* from T.

In this scheme, a *valid contract* is a string of the form
$$\Big([m, A, B]_A, [m, A, B]_B\Big)$$
or, in the case that T intervened,
$$\Big[[m, A, B, T]_A, [m, A, B, T]_B\Big]_T;$$
the latter is called a *proxy signature* and results from an exchange where T intervened on behalf of a party that sent a *resolve* request. We are assuming that the social infrastructure is in place to enforce it legally as completely as a normal contract.

There are two requests the trustworthy T must be able to handle: an *abort* from A and *resolves* from A or B. The *abort* is essentially a request from A that all future *resolves* from B on the contract m be disallowed. If, however, B has already *resolved*, T can and does directly deliver the proxy signature to A.

Either A or B may attempt to *resolve* by sending T the message

$$m_{resolve} \;=\; (\texttt{resolve}, [m, A, B, T]_A, [m, A, B, T]_B),$$

to which T replies with $[m_{resolve}]_T$ if no abort has yet been processed.

In this optimistic protocol, it is expected that A and B will only turn to the TTP for conflict resolution—in which case T must always be able to respond reliably.

4.2 Components

DFE runs on a group of n servers, of which up to $t < n/3$ may be faulty or corrupted by a malicious adversary. They are linked by an asynchronous point-to-point network.

The servers that make up the *DFE* must be initialized in a trusted way. In other words, an administrator generates the necessary cryptographic keys for the secure agreement protocols run by the *DFE* servers and for the digital signature scheme of the *DFE*. Then every server receives its initial keys in a secure way.

The *DFE* starts to operate when the *DFE* thread is started on every server. It will not start any protocols unless a client request is received.

4.3 Operation

Compared to the description above, the fair exchange protocols only have to be changed in the *abort* and *resolve* sub-protocols. In particular, T maintains some state information which must be kept in a consistent fashion across the separate *DFE* servers.

In the course of normal, friendly interactions, A and B will not contact T at all, and hence there need be no change whatsoever in the above protocol for their personal communication.

Let *tid* be a bit string agreed upon by A and B to uniquely identify their transaction (such as a hash of the contract itself, $tid = H(m)$) and assume that all *DFE* servers have key shares for a non-interactive $(n, t+1, t)$-threshold signature scheme $\mathcal{S}$ [21]. Its public key must be know to all clients.

4.3.1 Sub-protocol *abort*

If A invokes the abort protocol, it sends a message

$$(\texttt{abort}, \mathit{tid}, [m, A, B, \texttt{abort}]_A)$$

to all *DFE* servers. Then it waits to receive an answer from at least $n - t$ *DFE* servers that is of the form

$$(\texttt{answer}, \mathit{tid}, b, s)$$

where $b \in \{\texttt{aborted}, \texttt{resolved}\}$.

If $b = \texttt{resolved}$ then A verifies that s contains an $\mathcal{S}$-signature share on the message $[m, A, B, T]_A, [m, A, B, T]_B$ and assembles these to a proxy-signature if successful (note that $t + 1$ shares suffice for this).

4.3.2 Sub-protocol *resolve*

If A (or B) invokes the resolve protocol, it sends a message

$$(\texttt{resolve}, \mathit{tid}, [m, A, B, T]_A, [m, A, B, T]_B)$$

to all *DFE* servers. Then it waits to receive an answer from at least $n - t$ *DFE* servers that is of the form

$$(\texttt{answer}, \mathit{tid}, b, s)$$

where $b \in \{\texttt{aborted}, \texttt{resolved}\}$.

If $b = \texttt{resolved}$ then A (or B) verifies that s contains an $\mathcal{S}$-signature share on the message $[m, A, B, T]_A, [m, A, B, T]_B$ and assembles these to a proxy-signature if successful (note that $t + 1$ shares suffice for this).

4.4 Implementation

This section describes how the *DFE* servers implement the trusted party for fair exchange.

We assume that the digital signatures issued by clients of the service (A, B, ...) are verifiable by all *DFE* servers. This can be implemented in several ways, most likely it will be a public-key infrastructure that certifies the digital signature public keys of all clients (for example by running a certification according to Section 3).

Suppose that all servers keep a local *transaction database* $\mathcal{T}$ that contains entries of the form

$$(\mathit{tid}, b, s)$$

with $b \in \{\texttt{aborted}, \texttt{resolved}\}$ and s an arbitrary string.

A *DFE* server operates as follows. When it receives an `abort` or `resolve` request from a client pertaining to a particular *tid*, it first checks if any entry (tid, b, s) is present in $\mathcal{T}$. If yes, it returns a message

$$(\texttt{answer}, \mathit{tid}, b, s)$$

with b and s from $\mathcal{T}$.

If (one of) the digital signature(s) present in the request do not verify properly, the request is ignored.

Otherwise, it checks if there is an active request with *tid* around (which means that such a request has been received, but no corresponding `answer` has been sent yet). If so, it postpones answering the request until the handling of the active request with *tid* has terminated; then it reads the answer from the transaction database $\mathcal{T}$ as above.

Otherwise, no valid request pertaining to *tid* has been received yet. In this case, it starts a *binary validated Byzantine agreement* protocol [5, 22, 8] with transaction identifier set to *tid*. Its initial vote of 0 or 1 is determined by the contents of the client request and is 1 if and only if the message from the client contained a properly justified `resolve` request (i.e., where the signatures by A and by B are correct). If the initial vote is 1, the corresponding validation data in the Byzantine agreement consists of the client request.

The validation predicate of the binary validated agreement verifies that all initial votes of 1 are accompanied by properly justified `resolve` requests.

When the binary agreement protocol has decided for 0, the *DFE* server adds the tuple

$$(\mathit{tid}, \texttt{aborted}, -)$$

to $\mathcal{T}$. Should the binary agreement protocol decide for 1 (in which case it also returns validation data d), the *DFE* server computes an $\mathcal{S}$-signature share s on the message

$$[m, A, B, T]_A, [m, A, B, T]_B$$

which may be taken from d if the server did not receive a `resolve` request. It adds the tuple

$$(\mathit{tid}, \texttt{resolved}, s)$$

to $\mathcal{T}$.

Then the server returns a message

$$(\texttt{answer}, \mathit{tid}, b, s)$$

to the client who sent the request, with b and s from the entry (tid, b, s) that has just been added to $\mathcal{T}$.

4.5 Discussion

The present implementation of the trusted party for fair exchange corresponds to the original protocol for distributing the trusted party given in [7]. It relies only on a binary Byzantine agreement protocol.

The implementation given earlier in [5] is less efficient because it builds on an atomic broadcast protocol and multi-valued Byzantine agreement, which potentially involves many instances of binary agreement.

5 Other Applications

The approach of using a layer of Byzantine-fault tolerant replication middleware for maintaining critical services in wide-area distributed systems has been explored by several independent research groups recently. We briefly review three such applications here, a time-stamping service, a global data storage, and a serverless distributed file system.

5.1 Identiscape

Identisacpe [16], developed in a project at Stanford University, manages various identities that people take on during their Internet lifetime. It links old, potentially invalidated identities and public keys to new identities. This requires a *naming history service*, which is implemented using a trusted time-stamping service for public keys. Any time-stamped document signed under an outdated public key can thus still be verified. The time-stamping service is implemented in a distributed way, and Identiscape proposes to use protocols developed in the context of MAFTIA [5, 22, 6] for this purpose, including the protocols for threshold signatures, verifiable consistent broadcast, common coin (distributed shared random number generation) and multi-valued Byzantine agreement.

These protocols have been "implemented" in the *Narses* protocol simulator, where the impact of cryptographic operations and network latency are simulated by appropriate delays. As identity updates are considered to be rare events, Identiscape prioritizes safety over speed; the authors report simulations of systems with up to 148 servers — far more than ever intended for our protocols. They achieve a performance about 500 seconds per update (i.e., one multi-valued agreement) for a group of 148 participants [16]. In spite of the large number of participants and the fact that the unoptimized basic protocols were used, the authors consider this to be acceptable.

5.2 OceanStore

The goal of *OceanStore* [18, 14] developed at the University of California in Berkeley (`http://oceanstore.cs.berkeley.edu/`) is to provide highly available durable storage in a distributed way. To this end, a large number of servers redundantly store encrypted files provided by clients.

A file is stored by a group of servers, which may be different for every file. This group consists of two parts, a *primary ring* and a *secondary ring*.

The servers in the *primary ring* store the information that is necessary to locate all fragments of the file and maintain this information in a consistent way during updates. The primary ring employs fault-tolerant replication protocols for atomic broadcast and can tolerate malicious behavior by up to one third of its members. This corresponds to one of our distributed trusted services.

The servers in the *secondary ring* store redundant copies of fragments of every file. They may also serve as caches and proxies for other clients. No particular group structure is imposed on them, and they correspond to the clients in our model.

The two main protocols needed by the primary ring are:

Update serialization: Ensures that all updates to a file in the distributed storage are performed in the same order such that it remains in a consistent state. This functionality corresponds directly to an atomic broadcast. The authors propose to use the protocol of Castro and Liskov [10], which provides the same service in a weaker model as the atomic broadcast of MAFTIA [6].

Document certification: The primary ring may also sign the stored objects before distributing them to the secondary ring. This functionality is implemented using threshold signatures.

In contrast to the trusted third-party services in MAFTIA, the primary ring is very dynamic; in essence, every user of OceanStore may define its own set of trusted servers for the primary ring as well as the corresponding failure thresholds (within obvious bounds implied by the protocols). Thus, it is important to generate new sets of cryptographic keys efficiently and to distribute them on the fly without manual intervention. The MAFTIA protocols do currently not offer this service in a secure fashion.

So far, OceanStore does not directly use protocols developed by MAFTIA, but they are under consideration as part of the security architecture for providing consistency [3]. In any case, the techniques used in OceanStore are very related to our approach.

5.3 Farsite

Farsite [4] is a serverless distributed file system developed at Microsoft Research (`http://research.microsoft.com/sn/Farsite/`) relying on otherwise unused disk space in a large number of desktop machines. It should provide security, availability, and reliability by distributing multiple encrypted replicas of each file among the client machines.

Farsite encrypts the stored files under a key provided by the user. In order to save space when multiple users store copies of same data, their encryption technique apparently allows to detect and to map together ("coalesce") identical files, even when these files are encrypted with separate keys.

The directory component in Farsite includes a directory service that is implemented in a distributed fashion, such that the data for each directory is replicated among several client machines. Whereas the integrity of file data is guaranteed by digital signatures (one only needs enough file replicas to ensure a high degree of availability), the integrity of file meta-data depends on the integrity of the parent directory, which might be undetectably compromised by the machines that house the directory replicas. Therefore, the number of directory replicas is significantly higher than that of file replicas, and the directory replicas communicate using an atomic broadcast protocol that tolerates Byzantine faults, which protects them from attacks by a fraction of the machines holding the replicas.

The authors propose to use the fault-tolerant replication protocol of Castro and Liskov [10], which provides the same service in a weaker model as the atomic broadcast developed in the context of MAFTIA [6].

5.4 Conclusion

Identiscape, OceanStore, and Farsite differ in their goals and thus in the requirements put on them. But it is interesting to note that all of them rely on a central component that is implemented by a group of replicated servers using replication protocols tolerating Byzantine faults. This provides direct support for the approach followed by MAFTIA.

Two of these projects cite the approach of Castro and Liskov [10], which achieves a quite good performance, with update times below one millisecond in a high-speed LAN environment. It remains to be seen how this approach maps to the Internet, where latencies are significantly higher, and how it scales to a larger number of servers. Our own preliminary results show that infrequent update operations are feasible also on the Internet [8], even though our atomic broadcast protocol is actually one order of magnitude more expensive than the one of Castro and Liskov. (The reason is that they have to make a timing assumption, whereas our protocol is fully asynchronous.)

Recently, Kursawe and Shoup [15] have developed a protocol in the context of MAFTIA that combines the approach of Castro and Liskov with a multi-valued Byzantine agreement in a fully asynchronous model. It essentially achieves the same performance as the one of Castro and Liskov, which is near to optimal for the given model.

It also seems that key management, in particular the distributed generation of shared keys, is a problem that has to be addressed before such systems are widely deployed.

All of the mentioned applications assume that group membership is dynamic, but the only secure group membership and key distribution protocols known so far work in synchronous environments [9, 11]. Further research is needed to solve this problem.

Bibliography

[1] N. Asokan, M. Schunter, and M. Waidner, "Optimistic protocols for fair exchange," Research Report RZ 2858, IBM Research, 1996. (Extended abstract in 4th ACM Conference on Computer and Communications Security, Zurich 1997).

[2] N. Asokan, V. Shoup, and M. Waidner, "Optimistic fair exchange of digital signatures," *IEEE Journal on Selected Areas in Communications*, vol. 18, pp. 591–610, Apr. 2000.

[3] D. Bindel and S. C. Rhea, "The design of the OceanStore consistency mechanism." Manuscript, available from `http://www.cs.berkeley.edu/~dbindel/`, May 2000.

[4] W. J. Bolosky, J. R. Douceur, D. Ely, and M. Theimer, "Feasibility of a serverless distributed file system deployed on an existing set of a desktop PCs," in *Proc. Joint International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS 2000)*, 2000.

[5] C. Cachin, ed., *Specification of Dependable Trusted Third Parties.* Deliverable D26, Project MAFTIA IST-1999-11583, Jan. 2001. Also available as Research Report RZ 3318, IBM Research.

[6] C. Cachin, K. Kursawe, F. Petzold, and V. Shoup, "Secure and efficient asynchronous broadcast protocols (extended abstract)," in *Advances in Cryptology: CRYPTO 2001* (J. Kilian, ed.), vol. 2139 of *Lecture Notes in Computer Science*, pp. 524–541, Springer, 2001. Full version available as Cryptology ePrint Archive, Report 2001/006, `http://eprint.iacr.org/`.

[7] C. Cachin, K. Kursawe, and V. Shoup, "Random oracles in Constantinople: Practical asynchronous Byzantine agreement using cryptography," in *Proc. 19th ACM Symposium on Principles of Distributed Computing (PODC)*, pp. 123–132, 2000. Full version available from Cryptology ePrint Archive, Report 2000/034, `http://eprint.iacr.org/`.

[8] C. Cachin and J. A. Poritz, "*Hydra*: Secure replication on the Internet," Research Report RZ 3393, IBM Research, Jan. 2002.

[9] R. Canetti, R. Gennaro, A. Herzberg, and D. Naor, "Proactive security: Long-term protection against break-ins," *RSA Laboratories' CryptoBytes*, vol. 3, no. 1, 1997.

[10] M. Castro and B. Liskov, "Practical Byzantine fault tolerance," in *Proc. Third Symp. Operating Systems Design and Implementation (OSDI)*, 1999.

[11] R. Gennaro, S. Jarecki, H. Krawczyk, and T. Rabin, "Secure key generation for discrete-log based cryptosystems," in *Advances in Cryptology: EUROCRYPT '99* (J. Stern, ed.), vol. 1592 of *Lecture Notes in Computer Science*, pp. 295–310, Springer, 1999.

[12] O. Goldreich, *Foundations of Cryptography: Basic Tools*. Cambridge University Press, 2001.

[13] S. Goldwasser, S. Micali, and R. L. Rivest, "A digital signature scheme secure against adaptive chosen-message attacks," *SIAM Journal on Computing*, vol. 17, pp. 281–308, Apr. 1988.

[14] J. Kubiatowicz, D. Bindel, Y. Chen, S. Czerwinski, P. Eaton, D. Geels, *et al.*, "OceanStore: An architecture for global-scale persistent storage," in *Proc. Ninth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2000)*, Nov. 2000.

[15] K. Kursawe and V. Shoup, "Optimistic asynchronous atomic broadcast." Cryptology ePrint Archive, Report 2001/022, Mar. 2001. `http://eprint.iacr.org/`.

[16] P. Maniatis, T. Giuli, and M. Baker, "Enabling the long-term archival of signed documents through time stamping," Technical Report arXiv:cs.DC/0106058, Computer Science Department, Stanford University, Stanford, CA, USA, June 2001. Available at http://www.arxiv.org/abs/cs.DC/0106058.

[17] C. Rackoff and D. R. Simon, "Non-interactive zero-knowledge proof of knowledge and chosen ciphertext attack," in *Advances in Cryptology: CRYPTO '91* (J. Feigenbaum, ed.), vol. 576 of *Lecture Notes in Computer Science*, pp. 433–444, Springer, 1992.

[18] S. Rhea, C. Wells, P. Eaton, D. Geels, B. Zhao, H. Weatherspoon, and J. Kubiatowicz, "Maintenance-free global data storage," *IEEE Internet Computing*, vol. 5, pp. 40–49, September/October 2001.

[19] R. L. Rivest, A. Shamir, and L. Adleman, "A method for obtaining digital signatures and public-key cryptosystems," *Communications of the ACM*, vol. 21, pp. 120–126, Feb. 1978.

[20] F. B. Schneider, "Implementing fault-tolerant services using the state machine approach: A tutorial," *ACM Computing Surveys*, vol. 22, pp. 299–319, Dec. 1990.

[21] V. Shoup, "Practical threshold signatures," in *Advances in Cryptology: EUROCRYPT 2000* (B. Preneel, ed.), vol. 1087 of *Lecture Notes in Computer Science*, pp. 207–220, Springer, 2000.

[22] P. Veríssimo and N. F. Neves, eds., *First Specification of APIs and Protocols for the MAFTIA Middleware*. Deliverable D24, Project MAFTIA IST-1999-11583, Jan. 2001. Also available as Research Report RZ 3365, IBM Research.

[23] L. Zhou, F. B. Schneider, and R. van Renesse, "COCA: A secure distributed online certification authority," Technical Report 2000-1828, Department of Computer Science, Cornell University, Dec. 2000.