JOURNAL OF AEROSPACE COMPUTING, INFORMATION, AND COMMUNICATION
Vol. 3, October 2006

Stabilizing Certificate Dispersal

Mohamed G. Gouda and Eunjin (EJ) Jung*
The University of Texas at Austin, Austin, TX, U.S.A.

A certificate issued by a user u for another user v enables any user that knows the public key of u to obtain the public key of v. A *certificate dispersal* D assigns a set of certificates $D.u$ to each user u in the system so that user u can find a public key of any other user v without consulting a third party (provided that there are enough certificates in the system for user u to find the public key of user v). In this paper, we present a stabilizing certificate dispersal protocol that tolerates transient faults and changes in the certificate system. For example, when a certificate is issued or revoked, this change may lead the system into a state where the set of certificates assigned to each user no longer constitutes a certificate dispersal. Our "dynamic dispersal" protocol eventually brings the system back to a legitimate state where the set of certificates assigned to each user constitutes a certificate dispersal.

I. Introduction

In a distributed system, public-key cryptography is often used to provide security features such as authentication and authorization. For example, when a client wants to have assurance that he is communicating with the correct server, then the client can use the public key of the server for authentication. The client may pick up a random number and encrypt it with the public key of the server. When the server receives the encrypted message, the server decrypts the message with the matching private key and sends the number back to the client. When the client receives the correct number, the client can authenticate the server. In fact, this is how customers authenticate the web servers using Secure Socket Layer (SSL)[1] in the Internet. This use of public-key cryptography necessitates that the users know the public keys of other users in the system.

The public keys can be advertised through *certificates*. A certificate (u, v) issued by a user u for another user v contains the public key of user v and is signed with the private key of user u. Any user who knows the public key of user u can verify this certificate and obtain the public key of user v. A *certificate dispersal* D assigns a set of certificates $D.u$ to each user u in the system so that user u can find a public key of any other user v without consulting a third party (provided that there are enough certificates in the system for user u to find the public key of user v). In this paper, we show a stabilizing certificate dispersal protocol that tolerates transient faults and changes in the certificate system.

The concept of stabilization[2,3] was first introduced by Dijkstra.[4] His definition of a stabilizing system was "regardless of its initial state, it is guaranteed to arrive at a legitimate state in a finite number of steps." This concept is very useful in building a fault-tolerant system under a model of transient failures. For example, when a certificate is issued or revoked, this change may lead the system into a state where the set of certificates assigned to each user no longer constitutes a certificate dispersal. Our "dynamic dispersal" protocol eventually brings the system back to a legitimate state where the set of certificates assigned to each user constitutes a certificate dispersal. In Section V, we prove that our dynamic dispersal protocol is stabilizing.

Received 02 September 2005; revision received 28 June 2006; accepted for publication 12 July 2006.
*Department of Computer Sciences, University of Iowa, Iowa City, IA 52242, USA. E-mail: ejjung@cs.uiowa.edu

In the following sections, we give formal definitions of certificate systems and present our dynamic dispersal protocol. We prove that this protocol is stabilizing and discuss some events that may lead the system out of the legitimate states and show that the dynamic dispersal protocol eventually brings the system back to a legitimate state.

II. Certificate Systems

We consider a system where each user u has a private key $R.u$ and a public key $B.u$. In this system, in order for a user u to securely send a message m to another user v, user u needs to encrypt the message m using the public key $B.v$, before sending the encrypted message, denoted $B.v\{m\}$, to user v. This necessitates that user u knows the public key $B.v$ of user v.

If a user u knows the public key $B.v$ of another user v in this system, then user u can issue a certificate, called a certificate from u to v, that identifies the public key $B.v$ of user v. This certificate can be used by any user in the system that knows the public key of user u to further acquire the public key of user v. An example of such system is Pretty Good Privacy (PGP).[5]

A certificate from user u to user v is of the following form:

$$\langle u, v, B.v, \texttt{expr}, \texttt{sig}\rangle$$

This certificate is signed using the private key $R.u$ of user u, and it includes five items:

u	is the identity of the issuer,
v	is the identity of the subject,
$B.v$	is the public key of the subject v,
`expr`	is the expiration date, and
`sig`	is an encrypted message digest of this certificate.

`sig` is constructed by computing a message digest of all other four items in this certificate and encrypting the message digest with the private key $R.u$ of issuer u.

For simplicity, a certificate $\langle u, v, B.v, \texttt{expr}, \texttt{sig}\rangle$ is denoted (u, v). Any user x that knows the public key $B.u$ of user u can use $B.u$ to decrypt `sig` in (u, v). If the decrypted message matches the message digest of all other four items in the certificate and the current data is before `expr`, then user x can accept the key $B.v$ in certificate (u, v) as the public key of user v. A *valid* certificate (u, v) is an unexpired certificate with the correct signature.

Even though public-key cryptography has strong guarantees, a public key can be used only for a finite amount of time. (A dictionary attack will eventually succeed.) Therefore, each certificate has an expiration date and every certificate system requires some degree of clock synchronization. In practice, the expiration of certificates happens daily, and the lifetime of a certificate is often quite long, say a year, so the clock may be skewed by hours and this certificate system would still run correctly. As an alternative, we can also assume the clock rates of all users are the same. (In this case, we need to use version numbers instead of expiration dates.) All users will agree on the number of clock ticks as the lifetime of a certificate and use version numbers to verify the freshness of certificates. For simplicity, we assume that we have perfect clock synchronization in this paper. However, the protocol works as long as the clock skew is small enough that users will be able to detect expired certificates not too late.

The certificates issued by different users in a system can be represented by a directed graph, called the *certificate graph* of the system. Each node u in the certificate graph represents a user u and its corresponding public and private key pair $B.u$ and $R.u$. Each directed edge (u, v) from node u to node v in the certificate graph represents a certificate $\langle u, v, B.v, \texttt{expr}, \texttt{sig}\rangle$.

Fig. 1 shows a certificate graph for a system with five users: a, b, c, d, and e. According to this graph,

user a issued two certificates (a, b) and (a, d)
user b issued one certificate (b, c)
user c issued one certificate (c, e)

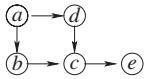

Fig. 1 A certificate graph example.

user d issued one certificate (d, c)
user e issued no certificates.

A simple path $(v_0, v_1), (v_1, v_2), \ldots, (v_{k-1}, v_k)$ in a certificate graph G, where the nodes $v_0, v_1, \ldots, v_k$ are all distinct, is called a *certificate chain* from v_0 to v_k in G of *length* k. Node v_0 in this chain can accept all the keys $B.v_1, \ldots, B.v_k$ in the certificates in this chain as the public keys of the users $v_1, \ldots, v_k$, respectively. For example, user a in Fig. 1 may use the certificate chain $(a, b)(b, c)$ to accept the public keys $B.b$ and $B.c$ of user b and user c.

III. Certificate Dispersal

In a certificate system, when a user u wants to securely communicate with another user v, u needs to find a certificate chain from u to v to obtain the public key of user v. Therefore, each user can store a subset of certificates in the certificate system to securely communicate with each other.

A *certificate dispersal* of a certificate graph G is a function that assigns a set of certificates $\texttt{CERT}.u$ to each user u in G such that the following condition holds. If there is a certificate chain from a user u to a user v in G, then u and v can find a chain from u to v using the certificates in the set $\texttt{CERT}.u \cup \texttt{CERT}.v$.

A certificate dispersal is *optimal* if and only if the average number of certificates stored in each user due to this dispersal is minimum.

For the certificate graph in Fig. 1, an optimal certificate dispersal is as follows:

$$\begin{aligned}
\texttt{CERT}.a &:= \{(a, d), (a, b), (b, c)\} \\
\texttt{CERT}.b &:= \{(b, c)\} \\
\texttt{CERT}.c &:= \{\} \\
\texttt{CERT}.d &:= \{(d, c)\} \\
\texttt{CERT}.e &:= \{(c, e)\}
\end{aligned}$$

Based on this dispersal, when user a wishes to securely communicate with user c, user a can use the two certificates (a, b) and (b, c) in $\texttt{CERT}.a$ to obtain the public key of user c. Also, when user b wishes to securely communicate with user e, user b can use the two certificates (b, c) in $\texttt{CERT}.b$ and (c, e) in $\texttt{CERT}.e$ to obtain the public key of user e.

In general, an optimal dispersal is hard to compute.[6] A certificate dispersal, that is not necessarily optimal, can be obtained by storing a "maximal reach tree" of certificates in each users. A *maximal reach* tree of a graph is a tree that contains all the reachable nodes in the graph from the root. Lemma 4 in[7] proves the following theorem.

Theorem 1. *A certificate dispersal of a certificate graph G is obtained by storing in each* $\texttt{CERT}.u$ *the certificates in a maximal reach tree rooted at u for each user u in G.*

For the certificate graph in Fig. 1, the certificate dispersal using reach trees is as follows:

$$\begin{aligned}
\texttt{CERT}.a &:= \{(a, d), (a, b), (b, c), (c, e)\} \\
\texttt{CERT}.b &:= \{(b, c), (c, e)\} \\
\texttt{CERT}.c &:= \{(c, e)\}
\end{aligned}$$

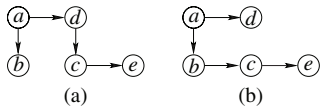

Fig. 2 Two possible reach trees.

$\texttt{CERT}.d \quad := \{(d, c), (c, e)\}$
$\texttt{CERT}.e \quad := \{\}$

Note that a maximal reach tree rooted at user u does not necessarily include all the users in the certificate graph. Each reach tree rooted at user u includes only the reachable users from u in the certificate graph. For example, the maximal reach tree rooted at user d includes only users d, c, and e. Also, there can be multiple reach trees in the certificate graph for the same root. For example, there are two possible maximal reach trees rooted at user a as shown in Fig. 2. CERT.a needs to contain the certificates of only one of the two reach trees. The example dispersal above contains the certificates from the reach tree in Fig. 2(b).

IV. Dynamic Dispersal

In the previous section, we discussed the concept of certificate dispersal. Gouda and Jung[7] showed how to compute a certificate dispersal for a "static" certificate graph, when the topology of the certificate graph does not change over time. However, in many certificate systems, certificate graphs do change due to issuing new certificates, adding new users, revoking old certificates, and removing old users. To maintain the certificate dispersal of a dynamic certificate graph, the changes in the graph need to be propagated to the appropriate users.

Fig. 3 shows the inputs and output of our dynamic dispersal protocol. The dynamic dispersal protocol running at each user has two inputs FORE and BACK. FORE in user u is the set of the certificates that have been issued by user u, and BACK in user u is the set of users that have issued certificates for u. Note that the two inputs FORE and BACK in all users define the certificate graph of the system. We assume that FORE and BACK are maintained by an outside protocol that issues new certificates and revokes old ones. We also assume that FORE and BACK are always *correct* and so they are always *consistent*. For example, if at any time a certificate (u, v) is in FORE.u of user u, then u is in BACK.v of user v at the same time.

The dynamic dispersal protocol maintains a variable CERT.u at each user u. At stabilization, the value of CERT.u is a maximal reach tree rooted at user u. Thus, by Theorem 1, the values of CERTs at stabilization constitute a certificate dispersal of the system.

The dynamic dispersal protocol in user u is shown in Protocol 1 below. Protocol 1 consists of three actions.

In the first action, when the timer of user u expires, user u uses its input FORE.u to update the variable CERT.u and sends a copy of CERT.u to each user v in BACK.u. Then u updates its timer to expire after ltime time units,

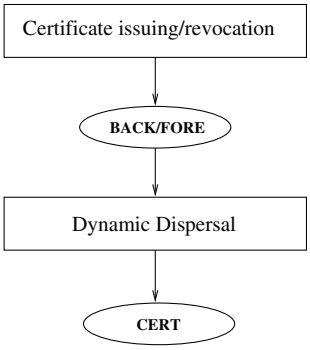

Fig. 3 Inputs and output of dynamic dispersal protocol.

and the cycle repeats. For convenience, we refer to CERT.u messages that user u has sent in this action as a round of gossip. If user u does not change its CERT.u and does not observe any change in its inputs FORE.u and BACK.u, then the time period between two consecutive rounds of gossip by u is ltime time units. The value ltime is expected to be in the range of days or months.

In the second action, user u receives a certificate tree sent by a user v (where u is in BACK.v). In this case, u updates its CERT.u using its input FORE.u, and then merges its CERT.u with the received certificate tree. If the update or merge operations change CERT.u then u reduces the value of its timer to at most stime time units. Note that the value stime is in the range of minutes or hours so it is much less than the value ltime. In other words, any change in the variable CERT.u causes u to initiate its next round of gossip after no more than stime time units.

In the third action, when user u observes that its inputs BACK.u or FORE.u has changed, then user u sets its timer to be at most stime time units. This change causes u to initiate its next round of gossip after no more than stime time units.

PROTOCOL 1 Dynamic dispersal

```
user u

const   stime, ltime                          //stime is a short time period
                                              //ltime is a long time period
                                              //ltime is greater than stime

input   BACK        : {x| x has issued a certificate (x,u)}
        FORE        : {(u,x) | u has issued a certificate (u,x)}

var     CERT        : a certificate tree rooted at u
        tree        : a certificate tree
        timer       : 0..ltime
        v           : any user other than u

begin
       timer=0 ->          update(CERT, FORE);
                           for each user v in BACK, send CERT to v;
                           timer:=ltime

    [] rcv tree from v -> update(CERT, FORE);
                          merge(CERT, tree);
                          if CERT has changed, timer:=min(timer, stime)

    [] BACK or FORE has changed -> timer:=min(timer,stime)

end
```

A. Issuing Certificates

When a user u issues a certificate (u, v), there are two events that need to occur. (Note that these two events occur outside the dynamic dispersal protocol.) The first event is to add (u, v) to FORE.u, and the second event is to add u to BACK.v. These events cause users u and v to execute the third action in the protocol and to reduce their timers to be at most stime time units. In stime time units, the timers in both users u and v will expire and then users u and v will execute the first action and update their CERTs and send a copy of the updated CERT to each user in their BACKs.

B. Revoking Certificates

When a user u wants to revoke a certificate (u, v) it has issued before, two events need to occur in users u and v. (Note that these two events occur outside the dynamic dispersal protocol.) The first event is to remove (u, v) from FORE.u, and the second event is to remove u from BACK.v.

When user u observes the change in FORE.u, u executes the third action and set its timer to be at most stime. When the timer expires, u will update CERT.u and send it to users in BACK.u. When a user x in BACK.u receives the newly updated CERT.u from user u, x will merge it with its own CERT.x. During this merge, the revoked certificate (u, v) and any path using that certificate will be removed from CERT.x.

C. `update` Procedure

Procedure update(CERT,FORE) is defined as follows.

PROCEDURE 1 update(CERT, FORE)

```
INPUT: a certificate tree CERT rooted at u and
       a set of certificates FORE issued by u
OUTPUT: a certificate tree CERT rooted at u

var tmp: a certificate tree rooted at u

begin

    add all the valid certificates in FORE to tmp;
    while there is a valid certificate (x,y) in CERT where
       x != u,
       x is in tmp, and
       y is not in tmp
    do add (x,y) to tmp;
    CERT:=tmp;

end
```

It is convenient to explain this procedure by an example. Consider user a where FORE.a in user a contains one certificate (a, b) and CERT.a contains two certificates (a, b), (b, c) as shown in Fig. 4(a). When user a issues a new certificate (a, c), FORE.a changes into $\{(a, b), (a, c)\}$. This change causes user a to execute its third action and then after at most stime time units to execute its first action. In the first action, procedure update(CERT.a, FORE.a) is executed. First, all the certificates in FORE.a are added to a certificate tree tmp and tmp becomes $\{(a, b), (a, c)\}$. Certificate (b, c) cannot be added to tmp because user c is already in tmp. In the last step, tmp is copied to CERT.a, and CERT.a becomes $\{(a, b), (a, c)\}$ as shown in Fig. 4(b).

D. `merge` Procedure

Procedure merge(CERT,tree) is defined as follows.

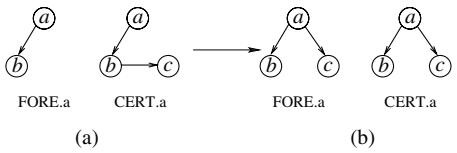

Fig. 4 `update` of `CERT`.a due to change in `FORE`.a.

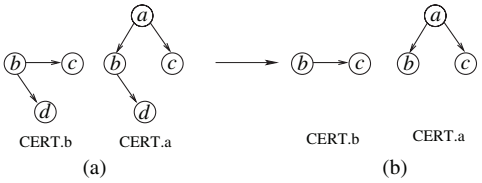

Fig. 5 `merge` of `CERT`.a due to change in `CERT`.b.

PROCEDURE 2 `merge(CERT, tree)`

```
INPUT: a certificate tree CERT rooted at u and
       a certificate tree ''tree'' rooted at t, where
       t != u
OUTPUT: a certificate tree CERT

begin

    if CERT has a certificate (u,t) ->
        remove from CERT the subtree rooted at t, if any;
        remove from tree every subtree rooted at a node, other than t,
        that occurs in CERT;
        while tree has a valid certificate (x,y) where
              x is in CERT and
              y is not in CERT
        do add y and certificate (x,y) to CERT;
    [] CERT has no certificate (u,t) ->
        skip
    fi

end
```

It is convenient to explain this procedure by an example. Consider user a where FORE.a contains two certificates (a, b), (a, c) and CERT.a contains three certificates (a, b), (a, c), (b, d) as shown in Fig. 5(a). When user b revokes certificate (b, d), FORE.b changes into $\{(b, c)\}$. This change causes user b to execute its third action and after at most stime time units to execute its first action. In the first action, user b updates its CERT.b to be $\{(b, c)\}$. User a still does not know about this revocation, so CERT.a remains the same as shown in Fig. 5(a). After stime time units, user b sends a copy of its CERT.b to user a. When user a receives the certificate tree $\{(b, c)\}$, user a executes its second action, and procedure `merge(CERT.a,tree)` is executed with CERT.a and the received tree $\{(b, c)\}$. Procedure `merge(CERT.a,tree)` first checks if there is certificate (a, b) in CERT.a. There is certificate (a, b), so the subtree rooted at user b, (b, d) in CERT.a is removed from CERT.a. Then, certificate (b, c) is considered, but is not added to CERT.a because c is already in CERT.a. In result, CERT.a becomes $\{(a, b), (a, c)\}$ as shown in Fig. 5(b).

V. Stabilization of Dynamic Dispersal

The dynamic dispersal algorithm in Section IV is based on a message passing model. Gouda and Multari[8] showed that it is hard to design stabilizing protocols in the traditional message passing model where there are channels between users. In this paper, we use a non-conventional model of communication. A state consists of the values of timer and CERT of all the users in the system. As mentioned in Section IV, we assume that FORE and BACK of each user remain correct and consistent in every state. In one state transition, only one user can execute its first action. Furthermore, in the same transition, each user v in BACK.u receives the same copy of this message and executes its second action. In other words, we have no messages in transit, so there is no need for channels in the state description.

There are two reasons that we adopted this model. First, this model allows the proofs to be easier to follow. Second, this model is sensible, given that the time it takes for the timer in each user to expire is very large compared to the time each state transition takes. `stime` is in the range of minutes and hours, and each state transition takes only milliseconds, so we can assume that no two timers expire at the same time.

For the proofs of convergence and closure, we define a *computation* to be a sequence of states of the system where along with this computation `FORE` and `BACK` of all the users remain unchanged. In the following theorems, we show that the dynamic dispersal protocol eventually stabilizes into a legitimate state, where the values of `CERT`s of all users constitute a certificate dispersal of the certificate graph of the system. Using the same proof technique as Arora and Gouda[9], we show the convergence and the closure of this protocol to prove its stabilization.

Theorem 2. *(Convergence) Each computation of the dynamic dispersal protocol has a state where the value of each `CERT`.u in the protocol is a maximal reach tree rooted at u in the certificate graph of the protocol (as defined by the two inputs `FORE` and `BACK` of all users in the protocol).*

Proof sketch. To prove that `CERT`.u eventually becomes a maximal reach tree rooted at node u of the certificate graph G, we first prove that `CERT`.u eventually becomes a tree rooted at u, and then prove that every node that is reachable from u in G is reachable in `CERT`.u.

There are two procedures, `update(CERT.u,FORE.u)` and `merge(CERT.u,tree)`, that can change `CERT`.u. The procedure `update(CERT.u,FORE.u)` constructs a tree by starting from the certificates in `FORE`.u. All the certificates in `FORE`.u are issued by user u, so the resulting tree from `update(CERT.u,FORE.u)` is rooted at u. Similarly, the procedure `merge(CERT.u,tree)` adds certificates in the received `tree` to `CERT`.u, a certificate tree rooted at u. Therefore, the resulting tree from `merge(CERT.u,tree)` is also rooted at u. Based on these observations, after a state transition in this computation, `CERT`.u in user u becomes a tree rooted at u.

Now we prove that `CERT`.u is a maximal reach tree, i.e. any node that is reachable from node u in G is also reachable in `CERT`.u. Assume that there is a path from u to another node v in G, $(u, u_1)(u_1, u_2) \cdots (u_k, v)$. Node u_k has the certificate (u_k, v) in its `FORE`, so the certificate (u_k, v) is in its `CERT`. Node u_k sends its `CERT` periodically to node u_{k-1}, so node u_{k-1} will have a path from itself to node v in its `CERT`. Repeatedly, each node on the path will send its `CERT` to the previous node in the path and node u will have a path from itself to node v in its `CERT`. Therefore, every node v that is reachable from node u in G is also reachable in `CERT`.u. ■

Note that our dynamic dispersal protocol is different from stabilizing spanning tree algorithms. The spanning tree algorithms[10–12] build a single spanning tree for the whole system that covers every process in the system, and build one tree rooted at a special process (usually referred as a leader). In particular, the algorithm by Afek and Bremler[13] builds a spanning tree in an unidirectional network, where a communication link between nodes is unidirectional. This unidirectional network is similar to our directed graph model of certificate systems, as our certificates are directed from one user to another. The routing tree building algorithm by Cobb and Gouda[14] also builds a single incoming spanning tree to the network root in a directed network. Each process in these algorithms stores the parent node identifier, the distance from the root, and possibly the root identifier. On the other hand, in our dynamic dispersal protocol, there is no leader, and each user u maintains a maximal reach tree rooted at u. Also, the maximal reach tree stored by our dynamic dispersal protocol does not necessarily cover every user in the system. Each maximal reach tree stored in each user may cover different set of users, and even for the same set of users the trees will have different root nodes.

Theorem 3. *(Closure) Executing any step of the dynamic dispersal protocol starting from a state, where the value of each variable `CERT`.u in the protocol is a maximal reach tree rooted at u, leaves the values of all `CERT` variables unchanged.*

Proof sketch. In a computation, the inputs `BACK` and `FORE` remain unchanged. Therefore, only two types of steps can be executed: time propagation and the first action. Time propagation cannot change the value of `CERT`. When the time propagation causes the timer in user u to expire, the first action in the dynamic dispersal protocol will be executed. When the timer expires, user u updates its `CERT`.u with `FORE`.u, but `CERT`.u remains the same since

FORE.u remains unchanged. Now user u sends a copy of its CERT.u to each user v in BACK.u. User v receives a tree and merge it with its own CERT.v. Since CERT.u is the same, merge(CERT, tree) will not change CERT.v. Therefore, when the certificate graph of the system does not change, CERT.u in each user u, a maximal reach tree rooted at u, remains unchanged. ■

VI. Time Complexity

In this section, we compute the time that takes to bring the system to stabilization in terms of the timer ltime. Note that each state transition is triggered by a timer expiration in a user, so any user will execute the first action of dynamic dispersal algorithm at least once in ltime time units. Also, the time that takes for a state transition is very small compared to ltime. Therefore, in ltime time units, we can assume that all users have executed the first action at least once.

Theorem 4. *In each computation of the dynamic dispersal protocol, the protocol reaches a legitimate state in at most T time units, where*

$$T = ltime \times (2p - 1),$$

where p is the length of the longest path in the certificate graph.

Proof sketch. A legitimate state of the dynamic dispersal protocol is one where the value of CERT.u of every user u in the system is a maximal reach tree rooted at u.

Consider a certificate (x, y) that is not in the certificate graph, but in some CERT.u of user u in the beginning of the computation. This certificate disappears from CERT of any user in the system in ltime $\times$ p. After the first ltime time units in the computation, user x updates CERT.x with FORE.x and remove the certificate (x, y) from CERT.x, if there was (x, y) in CERT.x. After the second ltime time units, any user in BACK.x receives CERT.x and removes the certificate (x, y) from its CERT, if there was (x, y) in its CERT. In other words, any user that had (x, y) in the second level of the tree in CERT removes (x, y) from its CERT. The cycle repeats, and after (ltime $\times$ p), any user that had (x, y) in its CERT removes (x, y) from its CERT.

Consider a certificate (v, w) that is in every possible reach tree rooted at some user u in the certificate graph, but not in CERT.u in the beginning of the computation. After the first ltime time units in the computation, user v updates CERT.v with FORE.v and add the certificate (v, w) to CERT.v if it was not in CERT.v already. For the next (ltime $\times$ $(p - 1)$) time units, a user in BACK.v may have node w in its CERT through a incorrect certificate and not add (v, w) to its CERT. However, any incorrect certificate will be removed from CERT of any user in (ltime $\times$ p) time units as shown above. Therefore, after (ltime $\times$ $(p + 1)$) time units since the beginning of the computation, any user in BACK.v adds (v, w) to its CERT, if it was not there already. In other words, any user that should have (v, w) in the second level of the tree in CERT adds (v, w) to its CERT. The cycle repeats, and after (ltime $\times$ $(2p - 1)$) time units, any user that should have (v, w) in its CERT adds (v, w) to its CERT.

As shown above, in (ltime $\times$ $(2p - 1)$) time units, any certificate that is not in the certificate graph disappears from CERT of every user, and any certificate that is in every possible reach tree of user u appears in CERT.u. Therefore, in (ltime $\times$ $(2p - 1)$), CERT.u becomes a maximal reach tree rooted at u. ■

We believe that the upper bound on the convergence span described in Theorem 4 is quite loose. It is an interesting problem to compute a tight upper bound of the convergence span.

VII. Dispersal in Client/Server Systems

This dynamic dispersal protocol is useful in any dynamic certificate system. Consider a client/server system, where there are much fewer servers than clients in the system. We can run the dynamic dispersal protocol among the servers and let any server issue a certificate for a client. Each server will have an maximal reach certificate tree in its CERT, so each server will be able to find a certificate chain from itself to any client that has a certificate issued by and other serve.

For example, many coffee shops offer free Internet connection for their customers. To prevent free-riders that are not customers, coffee shops may require the customers to register. For convenience, a customer needs to register

only once at any coffee shop (the coffee shop issues a certificate for the customer), and the customer can use the free connection at all coffee shops that are participating in this membership without logging in or getting temporary authorization each time he or she goes to a coffee shop, since any coffee shop has a certificate chain from itself to the customer. The authentication using the certificate chain does not require any interaction with the customer, so once the customer registers to get a certificate from one coffee shop, the customer does not need to know how he or she gets authenticated and authorized for the Internet connection.

Also, this client/server system can help two clients authenticate each other. A client $c1$ has issued a certificate for a server $s1$ and $s1$ issued a certificate for $c1$. A client $c2$ has issued a certificate for a server $s2$ and $s2$ issued a certificate for $c2$. When client $c1$ wants to securely communicate with client $c2$, client $c1$ can ask server $s1$ for a certificate chain from $s1$ to $s2$ and use the chain and the certificates $(c1, s1)$ and $(s2, c2)$ to find the public key of client $c2$.

The hierarchical certificate authority used in Lotus Notes[15] is a special case of such client/server system. In a system with a hierarchical certificate authorities, the certificate graph between certificate authorities constitutes a star graph, where the root certificate authority has issued a certificate for each non-root certificate authority and each non-root certificate authority has issued a certificate for the root certificate authority. In such a system, when a client $c1$ who has issued a certificate for a certificate authority $ca1$ wants to securely communicate with another client $c2$ who has issued a certificate for a certificate authority $ca2$, $c1$ can contact $ca1$ for certificates $(ca1, root)(root, ca2)$. In Lotus Notes, $ca1$ also finds the certificate $(ca2, c2)$ from $ca2$ so that $c1$ can use the public key of $c2$ safely without communicating with $c2$.

VIII. Concluding Remarks

Public-key cryptography is often used to provide security features in a distributed system. For users to use public-key cryptography, they need to know the public keys of other users. Certificates are useful to advertise public keys to other users. In particular, when a user u wishes to securely communicate with another user v, user u needs to find a certificate chain from u to v. A certificate dispersal D assigns a set of certificates $\texttt{CERT}.u$ to each user u so that user u can find such a chain in $\texttt{CERT}.u \cup \texttt{CERT}.v$.

We present the dynamic dispersal protocol, which eventually stabilizes a certificate system into a legitimate state where the set of certificates assigned to each user constitutes a certificate dispersal when a certificate graph of the certificate system is dynamic. We prove the convergence and the closure of the protocol, and show the time complexity of the convergence.

Acknowledgments

We are thankful to Shlomi Dolev, Ted Herman, and the anonymous reviewers for their valuable comments.

References

[1]Dierks, T. and Rescorla, E., The TLS Protocol Version 1.1. Internet Draft (draft-ietf-tls-rfc2246-bis-08.txt) 2004.

[2]Dolev, S., *Self-Stabilization*, MIT Press, 2000.

[3]Herman, T., "A Comprehensive Bibliography on Self-Stabilization," *Chicago Journal of Theoretical Computer Science*, 1996.

[4]Dijkstra, E.W., "Self-Stabilization in spite of Distributed Control," *ACM Communications*, Vol. 17, 1974, pp. 643–644.

[5]Zimmerman, P., *The Official PGP User's Guide*, MIT Press, 1995.

[6]Jung, E., Elmallah, E.S., and Gouda, M.G., "Optimal Dispersal of Certificate Chains," *Proceedings of the 18th International Symposium on Distributed Computing (DISC '04)*, Springer-Verlag, 2004.

[7]Gouda, M.G. and Jung, E., "Certificate Dispersal in ad-hoc networks," *Proceedings of the 24th International Conference on Distributed Computing Systems (ICDCS '04), IEEE*, 2004.

[8]Gouda, M.G. and Multari, N., "Stabilizing Communication Protocols," *IEEE Transactions on Computers, Special Issue on Protocol Engineering*, Vol. 40, 1991, pp. 448–458.

[9]Arora, A. and Gouda, M.G., "Closure and Convergence: A Foundation of Fault-Tolerant Computing," *IEEE Transactions on Software Engineering*, Vol. 19, 1993, pp. 1015–1027.

[10]Dolev, S., Israeli, A., and Moran, S., "Self-Stabilization of Dynamic Systems," *Proceedings of the 9th Annual ACM Symposium on Principles of Distributed Computing, ACM*, 1990.

[11]Arora, A. and Gouda, M.G., "Distributed Reset," *Proceedings of the 22nd International Conference on Fault-Tolerant Computing Systems.*, 1990.

[12]Chen, N.S., Yu, H.P., and Huang, S.T., "A Self-Stabilizing Algorithm for Constructing Spanning Trees," *Inf. Process. Lett.*, Vol. 39, 1991 pp. 147–151.

[13]Afek, Y. and Bremler, A., "Self-Stabilizing Unidirectional Network Algorithms by Power-Supply," *Chicago Journal of Theoretical Computer Science*, 1998.

[14]Cobb, J.A. and Gouda, M.G., "Stabilization of Routing in Directed Networks," *Proceedings of the 5th International Workshop on Self-Stabilizing (WSS 2001), LNCS 2194*, Springer-Verlag, 2001.

[15]Nielsen, S.P., Dahm, F., Lüscher, M., Yamamoto, H., Collins, F., Denholm, B., Kumar, S., and Softley, J. Lotus Notes and Domino r5.0 Security Infrastructure Revealed, 1999.

Shlomi Dolev
Associate Editor