Stabilizing Certificate Dispersal

Mohamed G. Gouda and Eunjin (EJ) Jung

Department of Computer Sciences,
The University of Texas at Austin,
Austin, TX U.S.A.
{gouda,ejung}@cs.utexas.edu

Abstract. 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. 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.

1 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. 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 states 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 5, we prove that our dynamic dispersal protocol is stabilizing.

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.

2 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 know 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

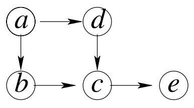

Fig. 1. A certificate graph example

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

A simple path (v_0, v_1), (v_1, v_2), $\cdots$, (v_{k-1}, v_k) in a certificate graph G, where the nodes v_0, v_1, $\cdots$, 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 $B.v_1 \cdots B.v_k$ in the certificates in this chain as the public keys of the users $v_1 \cdots v_k$, 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.

3 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 $CERT.u$ 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 $CERT.u \cup CERT.v$.

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:

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

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 $CERT.a$ 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 $CERT.b$ and (c,e) in $CERT.e$ 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 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* 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:

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

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).

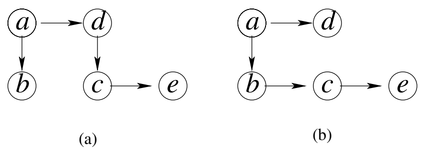

Fig. 2. Two possible reach trees

4 Dynamic Dispersal

In the previous section, we discussed the concept of certificate dispersal. Algorithms in [7] show how to compute a certificate dispersal for a "static" certificate graph, i.e. 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.

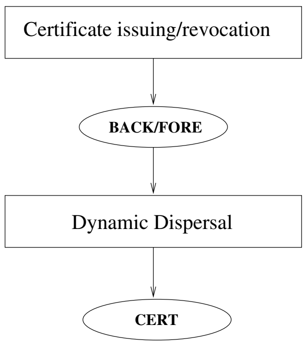

Fig. 3. Inputs and Output of Dynamic Dispersal Protocol

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, 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.

4.1 Issuing certificates

When a user u issues a certificate (u, v), there are two events that need to occur. (Note that these two events happen outside the dynamic dispersal protocol.) The first event is to add (u, v) to FORE.u, and the second action 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 accordingly and send a copy to the users in their BACKs.

4.2 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 happen outside the dynamic dispersal protocol.) The first event is to remove (u, v) from FORE.u, and the second action 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 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.

4.3 Expired Certificates

We assume that when a certificate (u, v) expires, it is removed from FORE.u and u is removed from BACK.v in user v. This triggers user u to set its timer to be at most

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
```

`stime` and user u will update its `CERT`.u accordingly and send a copy of `CERT`.u to users in `BACK`.u. Similarly to the case of certificate revocation, when a node x in `BACK`.u receives `CERT`.u, then x will update `CERT`.x and remove (u, v) from it.

4.4 `update` Procedure

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

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

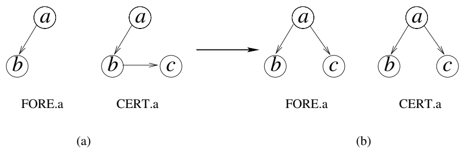

Fig. 4. `update` of `CERT`.a due to change in `FORE`.a

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
       v is not in tmp
    do add (u,v) to tmp;
    CERT:=tmp;

end
```

(a, c), FORE.a changes into $\{(a, b), (a, c)\}$. This change causes user a to execute its third action and then after 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).

4.5 merge Procedure

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

It is convenient to explain this procedure by an example. Consider user a where FORE.a contains two certificate $(a, b), (a, c)$ 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 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

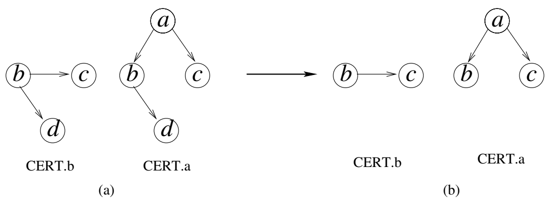

Fig. 5. merge of CERT.a due to change in CERT.b

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 all the certificates in the subtree rooted at t from 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
```

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).

5 Stabilization of Dynamic Dispersal

The dynamic dispersal algorithm in Section 4 is based on a message passing model. In [8], it is shown to be 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 4, 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. Following the proof technique in [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 `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 in [10,11,12] 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). Each process in these algorithms stores the parent node identifier, the distance from the root, and possibly the root identifier. On the other hand, our dynamic dispersal protocol stores a maximal reach tree in each user, which does not necessarily cover every user in the system. Also, in our dynamic dispersal protocol, there is no leader, and each user u maintains a maximal reach tree rooted at u.

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. ■

6 Time Complexity

In this section, we compute the time in terms of the two timers `stime` and `ltime` that takes to bring the system to stabilization. Note that each state transition is triggered by a timer expiration in a user, so the time between any two state transitions may be between 0 to `ltime`. Every state transition but the first one towards stabilization is triggered by a timer whose value is at most `stime`, which is shown below.

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

$$T = \mathtt{ltime} \times \textit{the length of the longest path in the certificate graph -1}$$

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.

After the first `ltime` time units in the computation, each `CERT`.u is a tree rooted at u, and the first two levels of this tree are correct. After the second `ltime` time units, each user sends a copy of its `CERT` to the users in the `BACK`, so the top three levels of each `CERT` are correct. The cycle repeats, and after `ltime` $\times$ (the length of the longest path in the certificate graph-1) time units, all the levels of each tree `CERT` are correct, so `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.

7 Dispersal in Client/Server Systems

This dynamic dispersal protocol is useful in any dynamic certificate systems. 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 an authenticated server.

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 $c2$.

A hierarchical certificate authorities used in Lotus Notes [13] 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 $c2$.

8 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 $CERT.u$ to each user u so that user u can find such a chain in $CERT.u \cup CERT.v$.

We present the dynamic dispersal protocol, which eventually stabilizes a certificate system into the legitimate states 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.

References

1. Dierks, T., 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 **17** (1974) 643–644
5. Zimmerman, P.: The Official PGP User's Guide. MIT Press (1995)
6. Jung, E., Elmallah, E.S., Gouda, M.G.: Optimal dispersal of certificate chains. In: Proceedings of the 18th International Symposium on Distributed Computing (DISC '04), Springer-Verlag (2004)
7. Gouda, M.G., Jung, E.: Certificate dispersal in ad-hoc networks. In: Proceedings of the 24th International Conference on Distributed Computing Systems (ICDCS '04), IEEE (2004)
8. Gouda, M.G., Multari, N.: Stabilizing communication protocols. EEE Transactions on Computers, Special Issue on Protocol Engineering **40** (1991) 448–458
9. Arora, A., Gouda, M.G.: Closure and convergence: A foundation of fault-tolerant computing. IEEE Transactions on Software Engineering **19** (1993) 1015–1027
10. Dolev, S., Israeli, A., Moran, S.: Self-stabilization of dynamic systems. In: Proceedings of the 9th Annual ACM Symposium on Principles of Distributed Computing, ACM (1990)
11. Arora, A., Gouda, M.G.: Distributed reset. In: Proceedings of the 22nd International Conference on Fault-Tolerant Computing Systems. (1990)
12. Chen, N.S., Yu, H.P., Huang, S.T.: A self-stabilizing algorithm for constructing spanning trees. Inf. Process. Lett. **39** (1991) 147–151
13. Nielsen, S.P., Dahm, F., Lüscher, M., Yamamoto, H., Collins, F., Denholm, B., Kumar, S., Softley, J.: Lotus notes and domino r5.0 security infrastructure revealed (1999)