An Efficient Distributed Group Key Management Algorithm

S. Rahul
Dept. of Computer Sc. & Automation
Indian Institute of Science
Bangalore - 560012
India
email: srahul@csa.iisc.ernet.in

R. C. Hansdah
Dept. of Computer Sc. & Automation
Indian Institute of Science
Bangalore - 560012
India
email: hansdah@csa.iisc.ernet.in

Abstract

A key agreement protocol is an important part of a secure group communication system(SGCS) which provides secure message passing services to its members. Among the various distributed key agreement protocols proposed in the literature, the tree-based group Diffie-Hellman(TGDH) protocol is the most efficient in terms of the number of keys that need to be maintained at each member and distribution of DH exponentiation operations among group members. In TGDH, on a group change, the group members need to perform between one and $O(\log_2 n)$ exponentiation operations [1] serially. Also, the messages that are passed during group key agreement must be authenticated using digital signatures. In this paper, we propose a new key agreement protocol which minimizes the number of exponentiation operations at each member. The member join operation requires only three members to perform one or two exponentiation operations each while the member leave operation requires only two or five group members to perform one or two exponentiation operations each. This is achieved at the cost of $O(\log_2 n)$ causal messages per member leave operation.

1 Introduction

Establishment of a common group key for encrypting group communication traffic is one of the most important functions of a SGCS. Key establishment protocols can be classified into two categories - key distribution protocols and key agreement protocols. Though centralized key distribution protocols[1, 2, 3, 4, 5, 6, 7, 8] can establish new group keys on change of group membership very efficiently and with minimum delay, distributed key agreement protocols are a better choice for SGCSs because of the inherent fault tolerant properties of these distributed algorithms.

Many distributed group key agreement protocols have been proposed in literature [9, 10, 11, 12, 13]. Most of these protocols make use of some extension of the two-party DH key agreement protocol to a group. The protocols GDH1, GDH2 and GDH3 [10] are direct extensions of the DH protocol to a group and in the following, they are referred to as GDH* protocols. These protocols are not very efficient for the following reasons.

- Average number of exponentiation operations performed by members during initial establishment of group key is $O(n)$.
- There is a large delay incurred during initial establishment of group key, since exponentiation operations at each member are performed only after it receives the result of an exponentiation from its previous member.
- The group leader will have to do $O(n)$ exponentiation operations on every membership change event. This causes a large delay in the formation of the new group key.

The TGDH protocol[9] solves many of the problems associated with the GDH* protocols. Each member participating in the secure group communication(SGC) maintains a binary key tree. The members occupy the leaf nodes. Every internal node *nd* of the binary tree represents a key shared by all members which are leaf nodes of the binary subtree rooted at *nd* and is computed by a single DH key agreement protocol between two groups of members occupying the leaf nodes of the two subtrees rooted at the two child nodes of *nd*. This protocol is more efficient than the GDH* protocols because of the following reasons.

- In the TGDH protocol, during initial establishment of group key, every member performs only

[1] If the tree is balanced

Proceedings of the Tenth International Conference on Parallel and Distributed Systems (ICPADS'04)
1521-9097/04 $ 20.00 IEEE

$O(\log_2 n)$ DH exponentiation operations.

- In the TGDH protocol, whenever the group membership changes, every member performs at most $O(\log_2 n)$ DH exponentiation operations.

Though the TGDH protocol is very efficient, it loads the members of the SGCS because of the $2D$ [2] serial exponentiation operations per membership change. This causes a lot of delay in resuming normal group communication. In this paper, we explore the possibility of further reducing the number of DH exponentiations required by a key agreement protocol. We present a distributed key management algorithm which reaches key agreement in $O(\log_2 n)$ rounds. The member-join operation requires at most four *concurrent* DH exponentiation operations. The member-leave operation requires at most six *concurrent* DH exponentiation operations and may require upto $O(\log_2 n)$ messages to be passed. Though there is an increase in the message complexity for handling member-leave events, because of reduction in the number of DH exponentiations, the increase in delay is small.

The rest of the paper is organized as follows. Section 2 contains an informal description of the algorithm. Section 3 contains the formal description of the algorithm along with correctness proofs and we conclude the paper in section 4.

2 Informal description of the algorithm

The following notations have been used to describe the algorithm.

n	The number of members in the group
$M_i (1 \leq i \leq n)$	The i^{th} member of the group.
T	The key tree
$root[T]$	Root node of T
$\{m\}_K$	Encryption of message m with key K
$\{c\}_{K^{-1}}$	Decryption of cipher text c using key K
p	The DH modulus. Both p and $\frac{p-1}{2}$ are prime.
g	The DH generator of order $p-1$ modulo p
α_i	Member M_i's long term private secret
$g^{\alpha_i} \bmod p$	Member M_i's public key

The heart of the algorithm involves maintaining a balanced binary key tree T at all members with the leaf nodes representing the group members and each internal node associated with a key shared between all those members which are at the leaves of the binary subtree rooted at the node having the key. Each internal node of the binary tree has exactly two children and is balanced in the sense that the difference in depths of any two leaf nodes is at most one, which is a much stronger requirement for balancing. The tree is securely built using a novel idea of a secure chain of leaf nodes which is established using DH key agreement between adjacent members in the chain. The performance of the group key management algorithm is ensured by using efficient algorithms for key management and keeping the tree balanced in the above sense.

Every node nd of T is associated with the following variables.

$left$ ($right$)	The left (right) child of nd (nil if nd is a leaf).
par	The parent node of nd.
key	The key associated with nd. It is nil if key is unknown or if nd is a leaf node.
$first$ ($last$)	The ID of the left (right) most leaf node of the subtree rooted at nd if nd is not a leaf node. Otherwise, it is the ID of nd.

A variable x associated with a node nd is referred to by the notation $x[nd]$.

2.1 Group key agreement

The algorithm proceeds in two phases. The members are arranged in a logical line. In the rest of the section, we will consider a group of nine members and describe the algorithm with respect to this group.

Phase 1 : In the first phase, every member M_i engages in a DH key agreement protocol with every other $M_j (\mid i-j \mid = 1)$. At the end of this phase, every pair of adjacent members $M_i, M_{i+1} (1 \leq i < n)$ will share a secret key. The two keys that a member M_i shares with its two neighbours are known locally as $leftkey$ and $rightkey$.

Figure 1: **Formation of the DH chain**

Phase 2 : In the second phase, a balanced binary tree is built in a distributed fashion with the result that every node knows only the keys at nodes along the path from itself to $root[T]$. In Figure 2, the darkened

[2] D is the depth at which a new member is added to the tree or an old member is removed from the tree

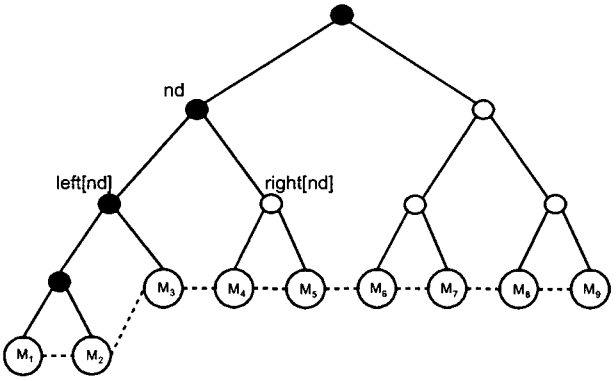

Figure 2: **The key tree**

nodes are the ones whose keys are known to members M_1 and M_2. The dashed lines represent the secure channels formed in stage 1.

The keys corresponding to the nodes of the tree are generated from bottom of the tree to the top, i.e , the key for a node nd is generated *after* generating the keys for $left[nd]$ and $right[nd]$ (unless nd is a leaf node). Consider the node nd in Figure 2. $key[nd]$ is generated after generating $key[left[nd]]$ and $key[right[nd]]$.

The member corresponding to the rightmost leaf node of the subtree rooted at $left[nd]$ selects a random value for $key[nd]$ and multicasts $\{key[nd]\}_{key[left[nd]]}$ to the members corresponding to the leaf nodes of the subtree rooted at $left[nd]$. It also sends $\{key[nd]\}_{rightkey}$ to the member corresponding to the leftmost leaf node of the subtree rooted at $right[nd]$. The leftmost leaf node of $right[nd]$ then decrypts it using its *leftkey* and multicasts $\{key[nd]\}_{key[right[nd]]}$ to the leaf nodes of the subtree rooted at $right[nd]$. Now, all leaf nodes of the subtree rooted at nd will know $key[nd]$.

2.2 Key change on member join

When a new member joins the group, first all members insert a node corresponding to the new member in their locally maintained trees. Then the keys along the path from the new member to the root node are modified in two phases.

2.2.1 Inserting a new member into the tree

While inserting a new member into the tree, our aim is to ensure that the resulting tree is as balanced as possible. Starting from the root node, we descend down the tree, at each node selecting the child node with minimum number of leaf nodes as its descendants(or the left node if the subtrees rooted at both child nodes have the same number of leaves). When we come to a leaf node, say L, we replace L by a new node, and make L the right child of the new node and M(node corresponding to new member) the left child of the new node. Then, all members are reassigned IDs such that the leaf nodes represent members $M_1 \cdots M_{n+1}$ from left to right.

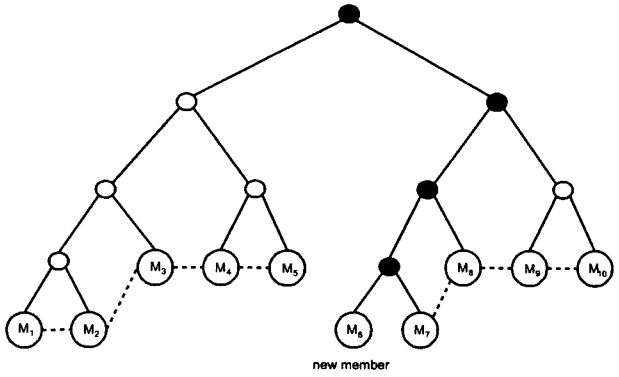

Figure 3: **Breaking of DH chain on member join**

The dark nodes in the tree shown in Figure 3 represent the nodes whose keys have to be changed to maintain backward secrecy. The establishment of these keys proceeds in two phases.

Phase 1 : If the addition of the new member M_i(ID i is assigned to the new member after inserting it into the tree) to the tree causes the chain of leaf nodes to break, the chain is completed using two DH key exchanges. If the chain is not broken (member is added at the beginning or end of the chain), then the chain is extended to include the new member by using one DH key exchange.

In Figure 3, DH keys have to be generated between members M_5 and M_6, and between members M_6 and M_7.

Phase 2 : In this phase, the keys of nodes along the path from the leaf node corresponding to the new member to the root are changed as follows.

Let N be the set of nodes such that the subtree rooted at each node $nd \in N$ contains the new member M_i as one of its leaf nodes. Every member $M_j (j \neq i)$ belonging to the subtree rooted at $k \in N$ replaces $key[nd]$ with its hash[3]. Since the right neighbour of the new member now has all new keys for nodes along the path from the node corresponding to the new member to the root, it can send these keys along with the logical tree to the new member securely.

[3] A strong one-way function like MD5 can be used

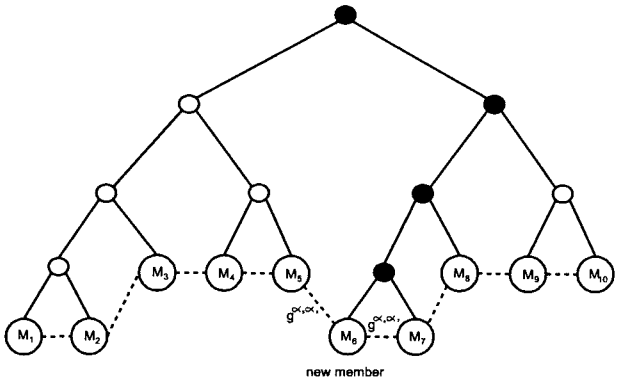

Figure 4: **Re-establishment of DH chain and the key tree**

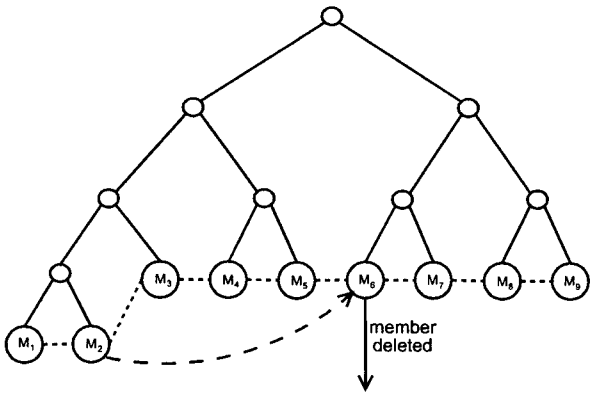

Figure 5: **Deletion of a member from the key tree**

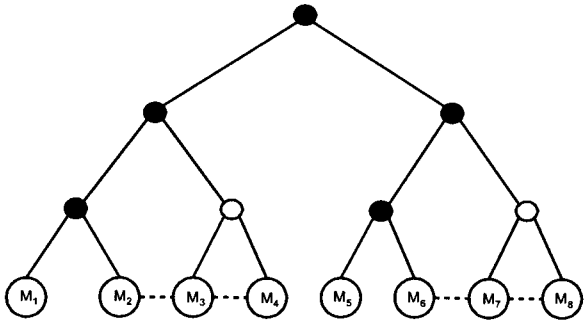

Figure 6: **Changing keys following member deletion**

2.3 Key change on member leave

When a member M leaves the group, first the node corresponding to the member is deleted from all the local trees maintained at other members of the group. If the deletion causes the tree to become unbalanced, balance is restored by moving a suitable leaf node from another part of the tree to occupy the position of the leaf node of the leaving member. After this rebalancing, the secure chain of leaf nodes is reestablished. Once the secure chain is reestablished, the keys of the following nodes must be changed.

- Nodes from leaving member's node to the root
- Nodes from the balancing member's node to the root(if a balancing is done)

The dark nodes in the tree shown in Figure 6 represent the nodes whose keys have to be changed. The number of such nodes is at most $2D - 1$ (when tree balancing is done after membership change) and at least $D - 1$(when tree remains balanced after the membership change). The generation of these node keys is similar to generation of node keys during group formation as explained in section 2.1. Also, in the figure, it can be seen that the DH chain is broken in three places. These keys should be generated before the keys corresponding to the internal nodes can be changed.

3 The algorithm

The following are some of the functions used in the algorithm.

- A sequence of numbers $i, \cdots, j$ can be divided into two groups as follows

$$\begin{aligned}&\mathbf{low}(\mathtt{i},\mathtt{j}) = (\mathtt{a},\mathtt{b}),\\ &\mathtt{where\ a} = \mathtt{i}, \mathtt{b} = \lfloor \mathtt{i} + (\mathtt{j}-\mathtt{i})/2 \rfloor\ ,\\ &\mathbf{high}(\mathtt{i},\mathtt{j}) = (\mathtt{a},\mathtt{b}),\\ &\mathtt{where\ a} = \lfloor \mathtt{i} + (\mathtt{j}-\mathtt{i})/2 \rfloor + 1, \mathtt{b} = \mathtt{j}.\end{aligned}$$

- The tuple $(first[nd], last[nd])$ associated with a node nd is referred to by the notation $\mathbf{id}(nd)$

In the following algorithm, we make use of a balanced binary tree T. The binary tree T is built independently by each member by calling the function CONSTRUCT_BT(1,n,T). The function CONSTRUCT_BT(i,j,st) is defined below.

```
CONSTRUCT_BT(i,j,st)
  if i = j then
    left[st], right[st] ← nil
    key[st] ← nil
  else
    left[st] ← NEW_NODE()
    right[st] ← NEW_NODE()
    par[left[st]], par[right[st]] ← st
    (x1, y1), id(left[st]) ← low(i, j)
    (x2, y2), id(right[st]) ← high(i, j)
    CONSTRUCT_BT(x1, y1, left[st])
```

CONSTRUCT_BT(x_2, y_2, right[st])
end if

3.1 Group key agreement

Let $M_i (1 \leq i \leq n)$ be the i^{th} member in the group. The aim of the algorithm is to build a balanced binary tree T with the members occupying the leaf nodes, every pair of adjacent members sharing a secret key and every internal node $node_{a,b}$ representing a key shared by members $M_i (a \leq i \leq b)$. There are three kinds of external events at each member M_i.

1. $\texttt{SEND}_{i,j}(\texttt{msg})$: Sending of a message msg from M_i to M_j.
2. $\texttt{MCAST}_{i,(a,b)}(\texttt{msg})$: Sending of a message msg from M_i to all M_j $(a \leq j \leq b)$.
3. $\texttt{RECV}_{j,i}(\texttt{msg})$: Receipt of a message msg at M_i from M_j.

The algorithm proceeds in 2 phases. In the first phase DH keys are established between pairs of members M_i, M_{i+1} and in the second phase, keys corresponding to all of the tree's internal nodes are generated in a distributed fashion.

Phase 1 :

- $M_i \longrightarrow M_{i+1} (1 \leq i < n) : g^{\alpha_i} \bmod p$
- $M_i \longrightarrow M_{i-1} (1 < i \leq n) : g^{\alpha_i} \bmod p$
- $M_i (1 \leq i < n) : rightkey \leftarrow g^{\alpha_i \alpha_{i+1}} \bmod p$
- $M_i (1 < i \leq n) : leftkey \leftarrow g^{\alpha_{i-1} \alpha_i} \bmod p$

Phase 2 : Every member M_i executes the following algorithm.

x ← par[M_i]
while x ≠ nil **do** {x is not the root node}
  l ← left[x]
  r ← right[x]
  if i = last[l] **then** {right most leaf node of l}
    key[x] ← RAND()
    $\texttt{MCAST}_{i,(\texttt{first[l]},\texttt{last[l]}-1)}(\{\texttt{key[x]}\}_{\texttt{key[l]}})$
    $\texttt{SEND}_{i,i+1}(\{\texttt{key[x]}\}_{\texttt{rightkey}})$
  else if first[l] ≤ i < last[l] **then**
    $\texttt{RECV}_{\texttt{last[l]},i}(\{\texttt{key[x]}\}_{\texttt{key[l]}})$
    key[x] ← $\{\{\texttt{key[x]}\}_{\texttt{key[l]}}\}_{\texttt{key[l]}^{-1}}$
  else if i = first[r] **then**
    $\texttt{RECV}_{\texttt{last[l]},i}(\{\texttt{key[x]}\}_{\texttt{leftkey}})$
    key[x] ← $\{\{\texttt{key[x]}\}_{\texttt{leftkey}}\}_{\texttt{leftkey}^{-1}}$
    $\texttt{MCAST}_{i,(\texttt{first[r]}+1,\texttt{last[r]})}(\{\texttt{key[x]}\}_{\texttt{key[r]}})$
  else if first[r] < i ≤ last[r] **then**
    $\texttt{RECV}_{\texttt{first[r]},i}(\{\texttt{key[x]}\}_{\texttt{key[r]}})$
    key[x] ← $\{\{\texttt{key[x]}\}_{\texttt{key[r]}}\}_{\texttt{key[r]}^{-1}}$
  end if
  x ← par[x]
end while

The above algorithm requires a total of $O(n)$ messages to be passed and requires $O(\log_2 n)$ rounds[4]. The keys corresponding to the internal nodes are generated from bottom to top. The key corresponding to the root node is generated at the end of the above algorithm. This is the secret group key which can be used to encrypt group communication traffic.

3.2 Group key change

The group key management(GKM) protocol has to change the group key whenever the group membership changes, and it is initiated on the occurrence of any one of the following two events.

- When a new member wants to join the group
- When an existing member has to be removed from the group

The algorithms for managing these events are explained next.

3.2.1 Member join

The aim of this algorithm is to include the node corresponding to the new member (say M) in the tree as a leaf node. M is added to T by using ADD(M, T). The function ADD is defined below. The function ADD makes use of a function SHIFT_RIGHT(nd) which increments the value of first[nd'] and last[nd'] for all nodes nd' in the subtree rooted at nd.

ADD(M,st)

st_size ← last[st] − first[st] + 1
if st_size = 1 **then** {Leaf node}
  new ← NEW_NODE()
  if st = left[par[st]] **then**
    left[par[st]] ← new
  else
    right[par[st]] ← new
  end if
  par[new] ← par[st]
  left[new] ← M
  right[new] ← st
  first[M], last[M], first[new] ← first[st]
  first[st], last[st], last[new] ← first[st] + 1

[4] An iteration of the while loop is a round

```
    par[M], par[st] ← new
  else {Non-leaf node}
    last[st] ← last[st] + 1
    l ← left[st]
    l_size ← last[l] − first[l] + 1
    r ← right[st]
    r_size ← last[r] − first[r] + 1
    if (r_size < l_size) then
      ADD(M, r)
    else
      ADD(M, l)
      SHIFT_RIGHT(r)
    end if
  end if
```

Lemma 1. *The execution of the member join algorithm in a balanced binary tree which has all leaf nodes at depth of* D *or* $D-1$ *results in a balanced binary tree.*

Proof. Assume that this is false, i.e , the leaf node corresponding to the joining member is added at a node n which is at a depth D resulting in a leaf node at a depth $D+1$ while there exists another member whose node nd is at a depth $D-1$. Consider the lowest node p such that both the node n (at a depth D) and a node n' (at a depth $D-1$) belong to the subtree rooted at p. The fact that during some iteration of the algorithm, the child node of p to whose subtree the new member was added was selected for adding the member, implies that the subtree had fewer leaf nodes than the subtree rooted at the other child node of p. Since we have chosen p to be the lowest node such that both the node n and the node n' (at a depth $D-1$) belong to the subtree rooted at p, all leaves of the subtree rooted at the child node of p to which n belongs are at height h. This means that a subtree of height h which has all leaves at depth h can have fewer members than a subtree of height h which has some leaves at depth $h-1$ which is a contradiction. □

Before M is added to the group, the old members of the group(n in number) are in a logical chain such that every adjacent pair of members have established a common key using DH key agreement protocol.

After M has been added to the tree at all members, the following two steps need to be taken.

Round 1 : The new member has to engage in DH key agreement protocol with its neighbours(at most two).

Round 2 : Each key from M to the root of the tree is replaced with its hash to preserve backward secrecy. The sibling of M sends its key tree to M encrypted with its $leftkey$.

When the above algorithm is being executed, the group multicast services have to be suspended. The delay before these activities can be resumed depends on the delay in executing the above algorithm. There are three types of delays incurred by the above algorithm.

1. Let the time taken for one large integer exponentiation operation be d. Let the maximum time required to reliably send a message be l. In the first round, two such exponentiation operations and one $SEND()$ operation are performed serially resulting in a maximum delay of $2d + l$.

2. Since the second round involves a single *SEND()* operation, the maximum delay in this round is l.

3. Other local calculations at each member introduce a delay which is negligible compared to the above two delays.

So, a member join operation causes a maximum delay of $2d + 2l$ and at most five messages are passed. But in the TGDH protocol, the member join operation requires one DH key agreement round and in the next round one member performs $2D$ serial exponentiations and broadcasts the modified key tree(with new values of blinded keys for the nodes along the path from the joining member's node to the root). So for the TGDH protocol, the delay is $d(2D + 3) + 2l$ and the messages passed include two unicasts and one broadcast.

3.2.2 Member leave

The aim of the GKM algorithm handling member leave events is to ensure that the tree remains balanced after the leave operation(difference in depth of any two leaves is at most one) and all the keys that the leaving member knew have to be changed to ensure forward secrecy.

The DELETE_NODE(M) function is used to delete a leaf node M from the tree T. The removal of a leaf node M might make the tree unbalanced. The tree is rebalanced as explained below. The DELETE_NODE(M) function makes use of the following functions.

DELETE(M) : This function deletes the node M from the tree T. After deletion, the $first$ and $last$ values of the nodes of T are re-adjusted.

INVALIDATE_KEYS(M) : This function sets the value of $key[nd]$ to nil for all nodes nd along the path from M to the root of the tree.

GET_BALANCER(M) : This function returns a node nd in the tree such that $depth(nd) = depth(M) + 2$.

DELETE_NODE(M)

```
(sib, LN) ← DELETE(M)
INVALIDATE_KEYS(sib)
if (height(T) = depth(sib) + 2)
  ∧(first[sib] = last[sib]) then {Tree has become
unbalanced}
  balancer ← GET_BALANCER(sib)
  INVALIDATE_KEYS(balancer)
  DELETE(balancer)
  p ← NEW_NODE()
  if LN = TRUE then {sib was left node before dele-
  tion of M}
    left[p] ← sib
    right[p] ← balancer
  else {sib was right node before deletion of M}
    left[p] ← balancer
    right[p] ← sib
  end if
  par[p] ← par[sib]
  if sib = left[par[sib]] then
    left[par[sib]] ← p
  else
    right[par[sib]] ← p
  end if
  par[sib], par[balancer] ← p

  {Renumbering the members}
  id ← first[sib]
  first[p], first[left[p]], last[left[p]] ← id
  last[p], first[right[p]], last[right[p]] ← id + 1
  x ← par[p]
  while x ≠ nil do
    last[x] ← last[x] + 1
    if x = left[par[x]] then
      SHIFT_RIGHT(right[par[x]])
    end if
    x ← par[x]
  end while
end if
```

Lemma 2. *The execution of the member leave algorithm in a balanced binary tree which has all nodes at depth of* D *or* $D-1$*, where* D *is the depth of the tree, results in a balanced binary tree.*

Proof. If the deletion of a leaf node causes an imbalance, it means that the sibling of the removed member's node is at a depth $D-2$ while another leaf node is at a height D. When this happens, a leaf node at height D is moved to take the place of the removed member's node. This ensures that the tree remains balanced. □

After M is deleted from the tree, the DH chain will be broken. The DH chain is completed, and then all the keys that were known to M and the member whose sibling was moved to another portion of the tree, have to be changed. These are changed by the following algorithm at each member M_i.

KEY_CHANGE_ON_LEAVE()

```
x ← par[M_i]
while x ≠ nil do
  l ← left[x]
  r ← right[x]
  if key[x] = nil then
    if i = last[l] then
      key[x] ← RAND()
      MCAST_{i,(first[l],last[l]-1)}({key[x]}_{key[l]})
      SEND_{i,i+1}({key[x]}_{rightkey})
    else if first[l] ≤ i < last[l] then
      RECV_{last[l],i}({key[x]}_{key[l]})
      key[x] ← {{key[x]}_{key[l]}}_{key[l]^-1}
    else if i = first[r] then
      RECV_{last[l],i}({key[x]}_{leftkey})
      key[x] ← {{key[x]}_{leftkey}}_{leftkey^-1}
      MCAST_{i,(first[r]+1,last[r])}({key[x]}_{key[r]})
    else if first[r] < i ≤ last[r] then
      RECV_{first[r],i}({key[x]}_{key[r]})
      key[x] ← {{key[x]}_{key[r]}}_{key[r]^-1}
    end if
  end if
  x ← par[x]
end while
```

The agreement algorithm for the key of any node requires two MCAST() operations and a SEND() operation. The first MCAST() operation and the SEND() operation can be executed concurrently. The SEND() operations of the different rounds can be executed concurrently in the beginning itself since the keys required to encrypt these messages are the DH keys which are already available. So even though the SEND() operation causally precedes the second MCAST() operation, it does not cause additional delay. Let the maximum time required to reliably send a message be l. As explained before, the authenticated DH key agreement in the first round requires at least $2d + l$ time to complete. Thus the member-leave operation of the GKM algorithm is completed within a time $2d + lD$.

In the TGDH protocol, the member which broadcasts the changed blinded keys has to perform $2(D-1)$ serial exponentiations and all other members have to perform at least one and at most $D-1$ exponentiations. So the time taken by the TGDH protocol to handle member-leave events is $2d(D-1) + l$. Thus, TGDH is faster than our algorithm only if the time required for sending a message is greater than the time

required for performing two exponentiations.

Our algorithm requires $O(log_2 n)$ messages to be sent for each member leave event. This is because all keys of the key tree from the leaving member to the root have to be changed and each key change requires two multicasts. If the underlying network does not support multicast, then multiple unicasts will have to be used for key agreement. In such a case, our algorithm would require at most five messages to be sent after a member join event and about $2n$ messages to be sent after a member leave event while the TGDH protocol would require about n messages to be sent after any membership change event.

4 Conclusion

In this paper we have presented a key management algorithm for secure group communication. The algorithm is fully distributed and secure, and it makes minimum use of Diffie-Hellman key agreement algorithm unlike other algorithms proposed in the literature which solely rely on the Diffie-Hellman protocol. Our protocol requires a fixed number (four for member join, and two or six for member leave) of concurrent exponentiation operations per membership change. In our algorithm, the group key change protocol messages are always authenticated because every member receives only encrypted messages from another member with whom it shares a secret key. So, the overhead of digitally signing protocol messages is absent. Our algorithm requires $O(log_2 n)$ messages to be sent for each member leave event. But since the computation overhead on group members per membership change is minimum, the algorithm is suitable for groups in which the members do not have the resources to frequently perform a number of Diffie-Hellman exponentiation operations.

References

[1] Marcel Waldvogel, Germano Caronni, Dan Sun, Nathalie Weiler, and Bernhard Plattner, "The versakey framework: Versatile group key management," *IEEE Journal on Selected Areas in Communications*, vol. 17, no. 9, pp. 1614–1631, September 1999.

[2] David A. McGrew and Alan T. Sherman, "Key establishment in large dynamic groups using one-way function trees," *IEEE transactions on software engineering*, vol. 29, no. 5, pp. 444–458, May 2003.

[3] P. McDaniel, P. Honeyman, and A. Prakash, "Lightweight secure group communication," Tech. Rep., University of Michigan, April 1998.

[4] A. Rowley and J. Dollimore, "Secure group communication for groupware applications," in *European Research Seminar on Advances in Distributed Systems (ERSADS)*, Zinal, Switzerland, 1997.

[5] Patrick McDaniel, "Secure high performance group communication," September 1997.

[6] Ohad Rodeh, Ken Birman, and Danny Dolev, "A study of group rekeying," Tech. Rep. TR2000-1791, Cornell University Computer Science, 2000.

[7] Chung Kei Wong, Mohamed G. Gouda, and Simon S. Lam, "Secure group communications using key graphs," in *Proceedings of the ACM SIGCOMM '98 conference on Applications, technologies, architectures, and protocols for computer communication.*, September 1998, pp. 68–79.

[8] Danny Dolev Ohad Rodeh, Kenneth P. Birman, "Using avl trees for fault-tolerant group key management," *International Journal of Information Security*, vol. 1, no. 2, pp. 84–99, February 2002.

[9] Yongdae Kim, Adrian Perrig, and Gene Tsudik, "Tree-based group key agreement," Tech. Rep. 2002/009, Department of Information and Computer Science, University of California at Irvine, CA, USA, 2002.

[10] Michael Steiner, Gene Tsudik, and Michael Waidner, "Diffie-hellman key distribution extended to group communication," in *Proceedings of the ACM Conference on Computer and Communications Security*, 1996, pp. 31–37.

[11] Mike Burmester and Yvo Desmedt, "A secure and efficient conference key distribution system," in *Advances in Cryptology - EUROCRYPT '94*, May 1994, pp. 275–286.

[12] Emmanuel Bresson, Olivier Chevassut, and David Pointcheval, "Provably authenticated group Diffie–Hellman key exchange — the dynamic case," *Lecture Notes in Computer Science*, vol. 2248, 2001.

[13] Yongdae Kim, Adrian Perrig, and Gene Tsudik, "Communication-efficient group key agreement," in *Proceedings of IFIP SEC 2001*, June 2001.