SECURE GROUP REKEYING USING BALANCED BINARY TREE IN DISTRIBUTED ENVIRONMENT

A DISSERTATION

Submitted in partial fulfillment of the requirements for the award of the degree of

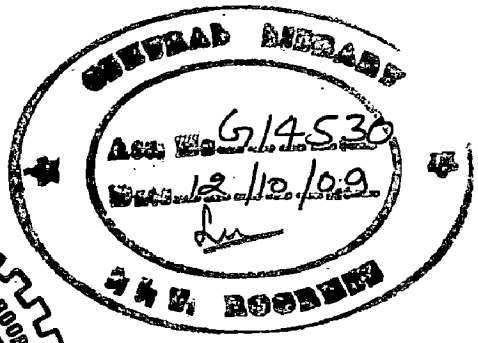
MASTER OF TECHNOLOGY

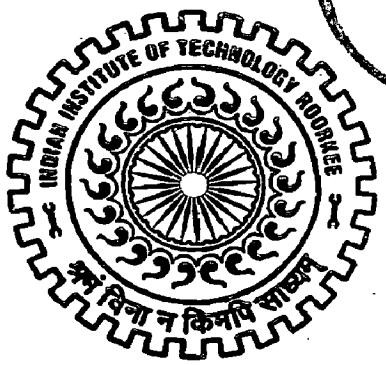
in

COMPUTER SCIENCE AND ENGINEERING

By

RANAJIT SENKO





DEPARTMENT OF ELECTRONICS AND COMPUTER ENGINEERING
INDIAN INSTITUTE OF TECHNOLOGY ROORKEE
ROORKEE-247 667 (INDIA)

JUNE, 2009

Candidate's Declaration

I hereby declare that the work being presented in the dissertation report titled "**Secure Group Rekeying Using Balanced Binary Tree in Distributed Environment**" in partial fulfillment of the requirement for the award of the degree of Master of Technology in Computer Science and Engineering, submitted in the Department of Electronics and Computer Engineering, Indian Institute of Technology Roorkee, is an authentic record of my own work carried out under the guidance of Dr. Kuldip Singh, Professor, in Department of Electronics and Computer Engineering, Indian Institute of Technology Roorkee. I have not submitted the matter embodied in this dissertation report for the award of any other degree.

Dated: 30-06-09 **(Ranajit Senko)**

Place: IIT Roorkee

Certificate

This is to certify that above statements made by the candidate are correct to the best of my knowledge and belief.

Dated: 30-06-09

Place: IIT Roorkee.

Dr. Kuldip Singh,
Professor,
Department of Electronics
and Computer Engineering,
IIT Roorkee, Roorkee,
247667 (India).

ACKNOWLEDGEMENTS

I am thankful to Indian Institute of Technology Roorkee for giving me this opportunity. It is my privilege to express thanks and my profound gratitude to my supervisors Dr. Kuldip Singh, Professor for their invaluable guidance and constant encouragement throughout the dissertation. I was able to complete this dissertation in time due to the constant motivation and support received from them.

I am also grateful to Mr. Sandeep Sood for helping me to understand some basic and important concepts explored in the dissertation work. I am grateful to Mr. Tirumalesh .C, Mr. Sandip Lokhande, Mr. Manoj Gupta and to all my friends who helped me directly and indirectly in completing this dissertation. Most importantly, I would like to extend my deepest appreciation to my family for their love, encouragement and moral support.

(Ranajit Senko)

ABSTRACT

Secure group communication is important for building distributed applications that work in dynamic network environments and communicate over insecure networks such as the global Internet. They can be used to building fault tolerant distributed applications. They can also provide support for distributed operating systems, distributed transactions and load balancing. The design and implementation of these distributed applications can be simplified by the group communication system. In distributed group communication systems, there is no single point of failure unlike centralize systems.

Though a large number of methods have been proposed and implemented for secure group communication in distributed environment, but they differ from each other in many aspects. These are computational overhead, storage requirement at each member, communicational overhead, distribution of load among group members and delay due to group membership change in new group key agreement. Also, there is no method to reduce the average waiting time for joining members. A lot of methods have already been proposed to solve these problems, but still there are no unique solutions to these problems.

In this Dissertation entitled "Secure Group Rekeying using Balanced Binary Tree in Distributed Environment", a solution is proposed for distributed key agreement algorithm which minimizes the computational overhead of group members, storage requirement at each group members and also reduces the average waiting time for joining members. It also ensures the message authentication very easily without the overhead of digitally signed messages. It also reduces the message delivery delay which makes group communication operation faster.

The implementation of the proposed work is done using java based event simulator JiST (Java in Simulation Time). The results of the proposed method are compared with the existing methods and improvement is obtained on computational overhead, number of rekeying messages, message delivery delay and average waiting time for the joining members.

CONTENTS

1.1 Introduction

A distributed system consists of a collection of autonomous computers, connected through a network, which enables computers to coordinate their activities and to share the resources of the system. As a result, users realize the system as a single integrated computing facility. Group communication is an important service in distributed system which is provided by a component of a distributed system, called Group Communication System. A Group Communication System facilitates multipoint to multipoint communication among a group of processes in a distributed system. The Group Communication Systems are useful for building real world distributed systems. They can be used to build fault tolerant distributed applications. They can also provide support for distributed operating systems, distributed transactions and load balancing. Recently, they are also being used for collaborative computing applications like video conferencing. The explosive growth of the internet has increased both the number and the popularity of applications that require a reliable group communication infrastructure. Secure group communication is important for building distributed applications that work in dynamic network environments and communicate over insecure networks such as the global Internet. Key management is the main issue for providing common security services (data secrecy, authentication and integrity) for group communication.

There are several different approaches to group key management and are divided into three main classes: centralize, decentralized and distributed group key management protocols [1]. In distributed key management approach, there is no group controller. Moreover, an advantage of distributed protocols over the centralized protocols is the increase in system reliability because the group key is generated in a shared and contributory fashion and there is no single point of failure. In distributed group key management system, group key is generated in a contributory fashion, where all members contribute their own share to the computation of the group key. When a member join or leave the group, group key needs to be refresh in order to provide backward or forward

secrecy, which is called individual rekeying. But individual rekeying has two problems: inefficiency and out of sync problem between key and data. To solve these two problems, a periodic batch rekeying method was proposed [2, 3 and 4]. But most of the approaches do not satisfy all attributes that make the group communication system more practicable. These attributes are computational overhead to each member, communicational overhead, and storage requirement of each member and distribution of load among group members.

1.2 Motivation

Centralized system has a problem of single point of failure. But in distributed system, there is no such problem. Moreover, it increases the system reliability as the group key is generated in a shared and contributory fashion. Whenever membership gets changed, group key needs to be refreshed in order to provide forward and backward secrecy. When it occurs with respect to a single member join or leave request, it is called individual rekeying. But it has two problems: inefficiency and out-of-sync problem. To overcome these two problems a periodic batch rekeying method is used. It is efficient because it reduces the number of rekey messages. But there is no unique solution which takes care of all the following problems in batch rekeying: computational overhead to each member, communicational overhead, and storage requirement of each member and distribution of load among group members, as well as average waiting time of joining members.

1.3 Problem Statement

The aim of this dissertation work is to design an algorithm for efficient implementation of secure distributed group communication. The main objective of the proposed algorithm is to minimize the cost and storage requirement of each members of the group, to minimize the computational overhead at each member and to minimize the average waiting time for joining of new members in group.

1.4 Organization of the Report

The report is divided into six chapters including this introductory chapter. Rest of the report is organized as follows:

Chapter 2 gives an over view of Group Communication System, classification of Group Communication System and different techniques available in each class briefly.

Chapter 3 presents brief description about the work done in the field of distributed group key agreement.

Chapter 4 describes the details of the proposed algorithm.

Chapter 5 discusses simulation results obtained from simulation.

Chapter 6 concludes the dissertation and provides directions for the future work.

BACKGROUND CHAPTER 2

2.1 Overview

The explosive growth of the internet has increased both the number and the popularity of applications that require a reliable group communication infrastructure. Some of them are Pay-per-view, stock quote distribution, voice and video conferencing, white boards and distributed simulations. Secure group communication is important for building distributed applications that work in dynamic network environments and communicate over insecure networks such as the global Internet. Key management is the main thing for providing common security services (data secrecy, authentication and integrity) for group communication.

The messages send among the group members are protected by encryption using the chosen key, which in the context of group communication, is called the group key. Only those who know the group key are able to extract the original message. The group may require the group key to be refreshed whenever the membership changes in order to preserve forward and backward secrecy. Group key management approaches can mainly be divided into three main classes [1]:

- Centralized Group Key Management Protocol
- Decentralize Group Key Management Protocol
- Distributed Group Key Management Protocol

2.2 Key Management Role

This section represents the common goal of three classes just mentioned above. Key management plays an important role in enforcing access control on the group key. It supports the establishment and maintenance of key relationships between valid parties according to a security policy being enforced on the group [5]. It encompasses techniques and procedures that can carry out:

- *Providing member identification and authentication.* Authentication is important in order to prevent an intruder from impersonating a legitimate group member. In addition, it is important to prevent attackers from impersonating key managers. Thus, authentication mechanisms must be used to allow an entity to verify whether another entity is really what it claims to be.
- *Access control.* After a party has been identified, its join operation should be validated. Access control is performed in order to validate group members before giving them access to group communication.
- *Generation, distribution and installation of key material.* It is necessary to change the key at regular intervals to make safe its secrecy. Additional care must be taken when choosing a new key to guarantee key independence. Each key must be completely independent from any previously used and future keys; otherwise; compromised keys may reveal other keys.

The key secrecy can be extended to membership changes. When a group requires backward and forward secrecy [7], the key must be changed for every membership change. Backward secrecy is used to prevent a new member from decoding messages exchanged before it joined the group. If a new key is distributed for the group when a new member joins, it is not able to decipher previous messages even if it has recorded earlier messages encrypted with the old key. Forward secrecy is used to prevent a leaving group member to continue accessing the group's communication. If the key is changed as soon as a member leaves, that member will not be able to decipher group messages encrypted with the new key.

As multicast is being used for group transmission, it is generally assumed that multicast should also be used to rekey the group. It is not reasonable to consider transmitting data using a scalable multicast communication and rekeying the members under a non scalable peer-to-peer communication. If the group has thousands of members, sending them a new key one by one would not be efficient. Although rekeying a group after the join of a new member is trivial, rekeying the group after a member leaves is far more complicated. The old key cannot be used to distribute a new one, because the leaving member knows the

old key. A group key distributor must therefore provide other mechanisms to rekey the group using multicast messages while maintaining the highest level of security possible.

2.3 Centralized Group Key Management Protocol

In a centralized system, there is only one entity controlling the whole group. Therefore, there is a problem of single point of failure. The entire group will be affected if there is a problem with the controller. The group privacy is dependent on the successful functioning of the single group controller. When the controller is not working, the group becomes vulnerable because the keys, which are the base for the group privacy, are not being generated/regenerated and distributed. Furthermore, the group may become too large to be managed by a single party, thus raising the issue of scalability. The group key management protocol used in a centralized system seeks to minimize the requirements of both group members and Key Distribution Center (KDC) in order to augment the scalability of the group management. The efficiency of the protocol can be measured by:

- *Storage requirements.* The number of key encryption keys (KEKs) that group members and the KDC need to keep.
- *Size of messages.* Measure by the number of bytes requires for a rekey message for adding and removing members. The protocol can combine unicast and multicast messages to achieve the best results. Note that the usage of unicast channels implies establishing a secure channel, hence increasing the total cost of the protocol.
- *Backwards and forward secrecy.* The capability of a protocol to provide secrecy despite changes to the group membership.
- *Collusion.* Evicted members must not be able to work together and share their individual piece of information to regain access to the group key.

2.3.1 Group Key Management Protocol

The Group Key Management Protocol (GKMP) [8, 9] enables the creation and maintenance of a group key. In this approach, the Key Distribution Center (KDC) helped by the first member to join the group creates a Group Key Packet (GKP) that contains a group traffic encryption key (GTEK) and a group key encryption key (GKEK). When a

new member wants to join the group, the KDC sends it a copy of the GKP. When a rekey is needed, the GC generates a new GKP and encrypts it with the current GKEK ($\{GTEK\}_{GKEK}$). As all members know the GKEK, there is no solution for keeping the forward secrecy when a member leaves the group except to recreate entirely new group without that member.

2.3.2 Logical Key Hierarchy

Wong et al. [10] proposed the use of a Logical Key Hierarchy (LKH). In this approach, a KDC maintains a tree of keys. The nodes of the tree hold key encryption keys. The leaves of the tree correspond to group members and each leaf holds a Key Encryption Key (KEK) associated with that one member. Each member receives and maintains a copy of the KEK associated with its leaf and the KEKs corresponding to each node in the path from its parent leaf to the root. The key held by the root of the tree is the group key. For a balanced tree, each member stores at most $(\log_2 n) + 1$ keys, where $(\log_2 n)$ is the height of the tree. For example, as shows in Figure 2.1, member u_1 knows k_1, k_{12}, k_{14} and k. A joining member is associated with a leaf and the leaf is included in the tree. All KEKs in the nodes from the new leaf's parent in the path to the root are compromised and should be changed (backward secrecy). A rekey message is generated containing each of the new KEKs encrypted with its respective node's children KEK. The size of the message produced will be at most $2 * (\log_2 n)$ keys long. Figure 2.1 shows an example of the KEKs being affected. The new member u_3 receives a secret key k_3 and its leaf is attached to the node k_{34}. The KEKs held by nodes k_{34}, k_{14} and k, which are the nodes in the path from k_3 to k, are compromised. New KEKs (k'_{34}, k'_{14} and k') are generated as shown in Figure 2.2. Finally, the KEKs are encrypted with each of its respective node's children KEK ($\{k'_{34}\}_{k3, k4}$; $\{k'_{14}\}_{k12, k'34}$; and $\{k\}_{k'14, k58}$ (as shown in Figure 2.2). The size of a rekeying message for a balanced tree has at most $2 * (\log_2 n)$ keys. Removing a member follows a similar process. When a member leaves (or is evicted from) the group, its parent node's KEK and all KEKs held by nodes in the path to the root are compromised and should be updated (forward secrecy). A rekey message is generated containing each of the new KEKs encrypted with its respective node's children KEK. The exception is the parent node of the leaving member's leaf. The KEK held by this node is encrypted only

with the KEK held by the remaining member's leaf. As the key held by the leaving member was not used to encrypt any new KEK, and all its known KEKs were changed, it is no longer able to access the group messages.

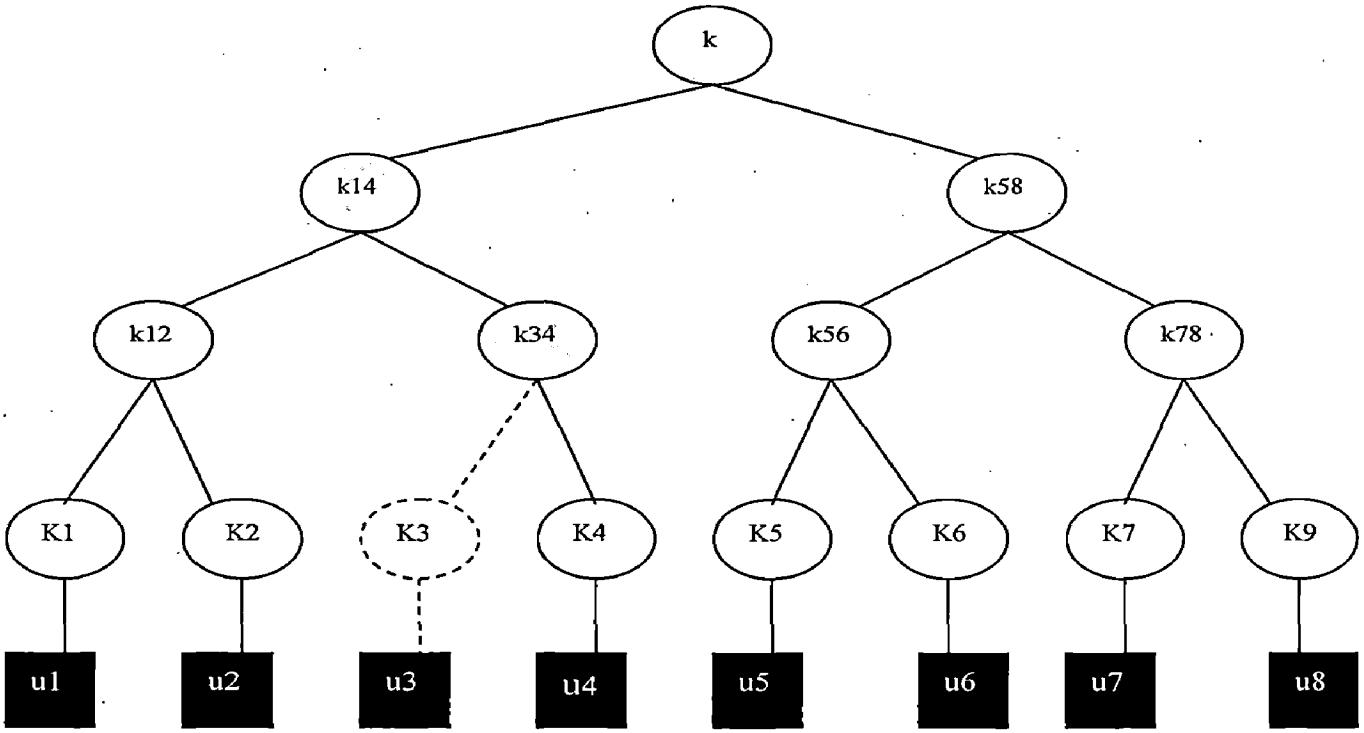
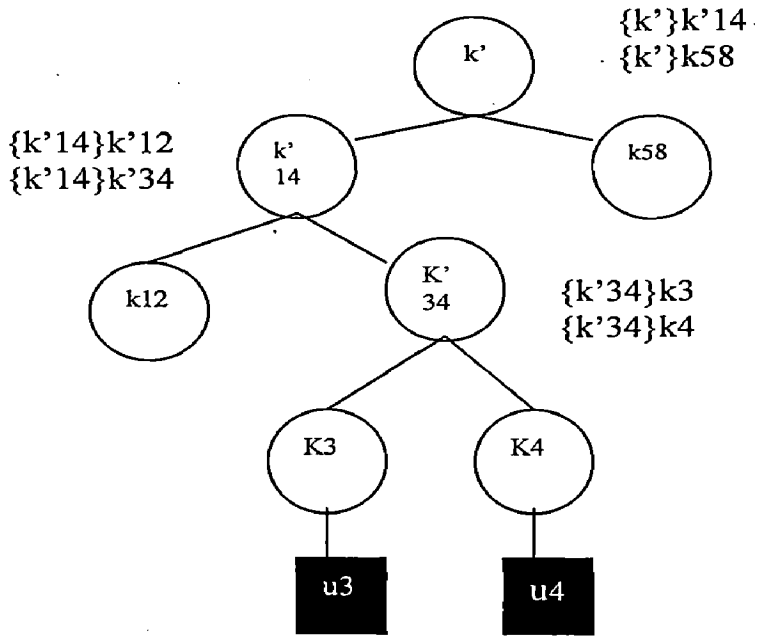


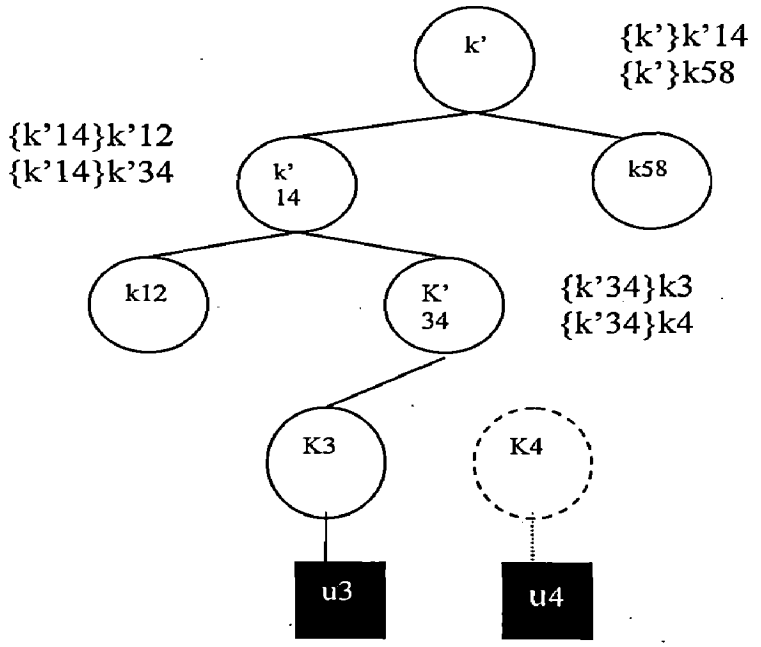
Figure 2.1 KEKs affected when a member joins the tree

Figure 2.3 presents what happens when a member leaves. Member u_4 is leaving the group and it knows KEKs k_{34}, k_{14} and k. KEKs k'_{34}, k'_{14} and k' are updated and encrypted with each of its respective children's KEKs. An exception is made for the k'_{34}. This KEK is encrypted only with k3, which is the key of the remaining member of n_{34}. There has another same approach like LKH except for joining operations. Instead of generating fresh keys and sending them to members already in the group, all keys affected by the membership change are passed through a one way function. Every member that already knew the old key can calculate the new one. Hence, the new keys do not need to be sent and every member can calculate them locally. This algorithm is known as LKH+.



{x}k means x has been encrypted with k

Figure 2.2 Necessary encryption when a member joins the tree in the basic LKH.



{x}k means x has been encrypted with k

Figure 2.3 Necessary encryption when a member is removed in the basic LKH.

2.3.3 One-Way Function Tree

An improvement over hierarchical binary tree approach is a one-way function tree (OFT) [11]. This scheme reduces the size of the rekeying message from 2 * ($\log_2 n$) to only ($\log_2 n$). Here a node's KEK is generated rather than just attributed. The KEKs held by a node's children are blinded using a one way function and then mixed together using a mixing function. The result of this mixing function is the KEK held by the node. This is represented by the following formula:

$$k_i = f\ (\ g\ (\ k_{left(i)}),\ g\ (\ k_{right(i)})\ \ .....(1)$$

Where left(i) and right(i) denote the left and right children of node i respectively. The function g is one way, and f is a mixing function: ancestors of a node are those nodes in the path from its parent node to the root. The set of ancestor of a node is called ancestor set and the set of siblings of the nodes in ancestor set are called sibling set (as shows in Figure 2.4). Each member receives the key (associated to its leaf node), its sibling's blinded key and the blinded keys corresponding to each node in its sibling set. For a balanced tree, each member stores $\log_2 n + 1$ keys. For example, in Figure 2.4, member u_4 knows key k_4 and blinded keys k_3^B (its sibling's blinded key) and k_{12}^B and k_{58}^B (blinded keys in u_4's sibling set). Putting these values in equation1, member u_4 is able to generate all keys in its ancestor set (k_{34}, k_{14} and k). The message size reduction is achieved because in the standard scheme, when a node's key changes, the new key must be encrypted with its two children's keys, and in the OFT scheme, the blinded key changed in a node has to be encrypted only with the key of its sibling node. Figure 2.5 shows an example of this scheme. Member u_3 joins the group at node n_{34}. It requires keys k_{34}, k_{14} and k to be changed. The only values that must be transmitted are the blinded KEKs k_3^B, $k'_{34}{}^B$ and $k'_{14}{}^B$. And all encrypted with k_4, k_{12} and k_{58} respectively. The new KEKs can be calculated by every group member: $k'_{34} = f\ (g\ (k_3),\ g\ (k_4))$, $k'_{14} = f\ (g\ (k_{12}),\ g\ (k'_{34}))$ and $k' = f\ (g\ (k'14),\ g\ (k_{58}))$.

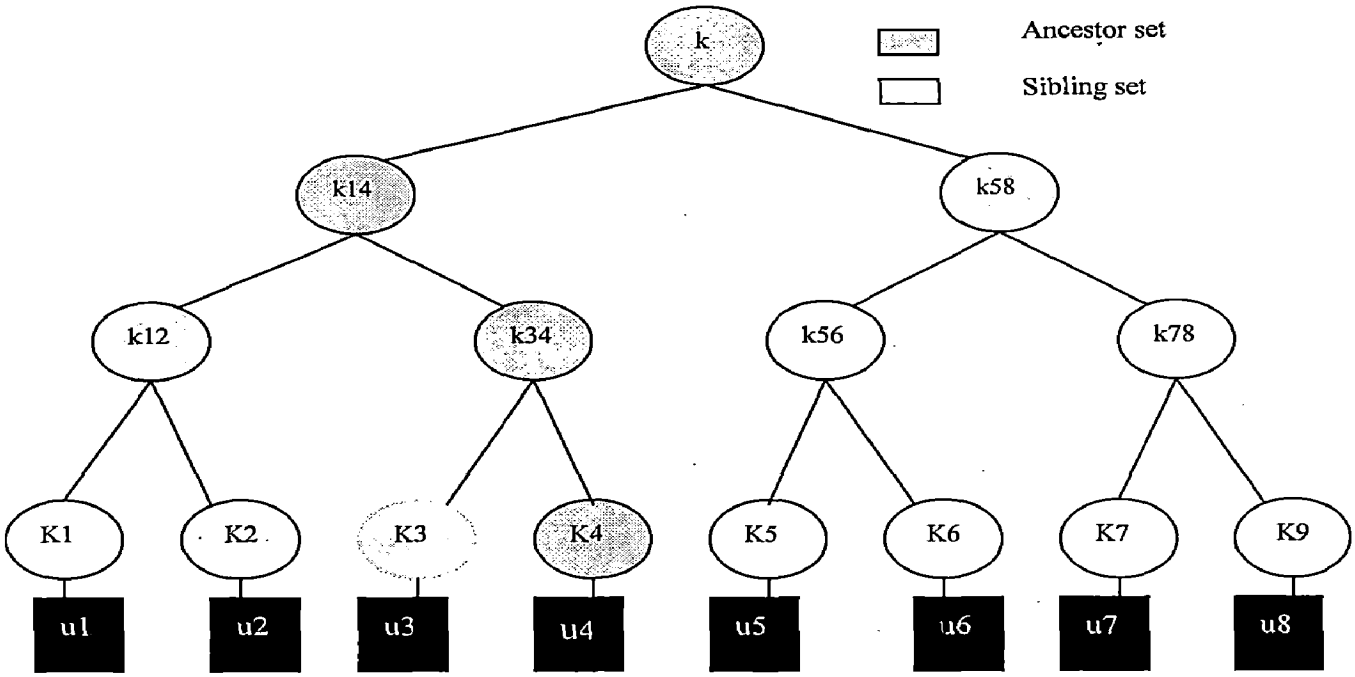


Figure 2.4 Ancestor and sibling sets of member

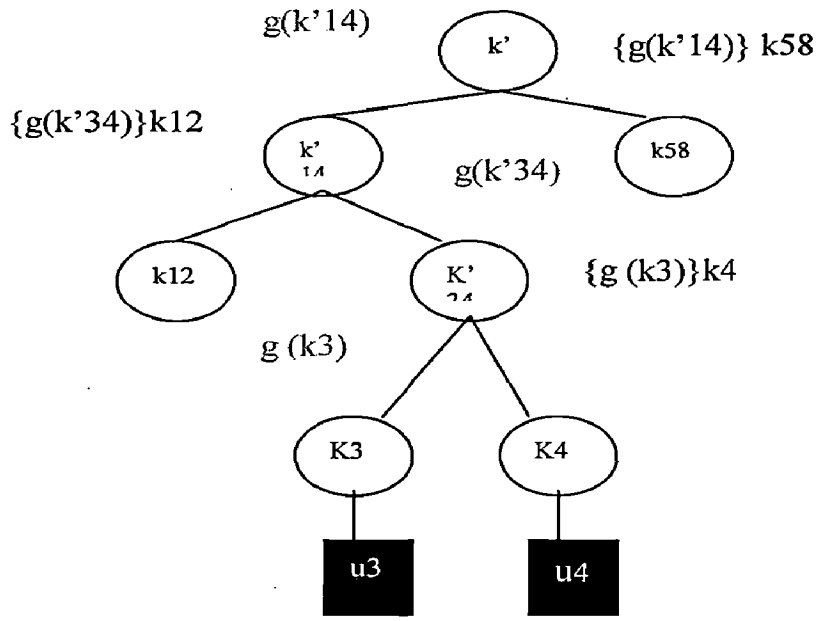


Figure 2.5 Necessary encryption when u_3 joins the tree in the improved LKH.

2.3.4 One-Way Function Chain Tree

Canetti et al. [12] proposed a slightly different approach that achieves the same communication overhead as in case of one way function tree. This scheme uses a pseudo random generator to generate the new KEKs rather than a one way function and it is

applied only on user's removal. This scheme is known as the *one way function chain tree.* The pseudo random generator, G(x), doubles the size of its input (x), and there are two functions, L(x) and R(x), that represent the left and right halves of the output of G(x) (i.e., G(x) = L(x)R(x), where | L(x) | = | R(x) | = | x |). When a user u leaves the group, the algorithm to rekey the tree goes as follows:

- A new value r_v is associated to every node v from u to the root of the tree using $r_{p(u)} = r$ for the first node and $r_{p(v)} = R(r_v)$ for all other v (where p(v) denotes the parent of v).
- The new keys are generated as $k'_v = L(r_v)$.
- Each $r_{p(v)}$ is encrypted with key $k_{s(v)}$ (where s(v) denotes the sibling of v) and sent off. From r_v, one can compute all keys k'_v, $k'_{p(v)}$, $k'_{p(p(v))}$ up to the root node key. Taking into account the example of Figure 2.1, if u_1 leaves the group (Figure 2.6), nodes n_{12}, n_{14} and n_0 will be associate respectively with r, R(r) and R(R(r)) and these values will be encrypted for n_2, n_{34} and n_{58}, with their respective KEKs (k_2, k_{34} and k_{58}). Finally, the new KEKs k_{12}, k_{14} and k will be L(r), L(R(r)) and L(R(R(r))).

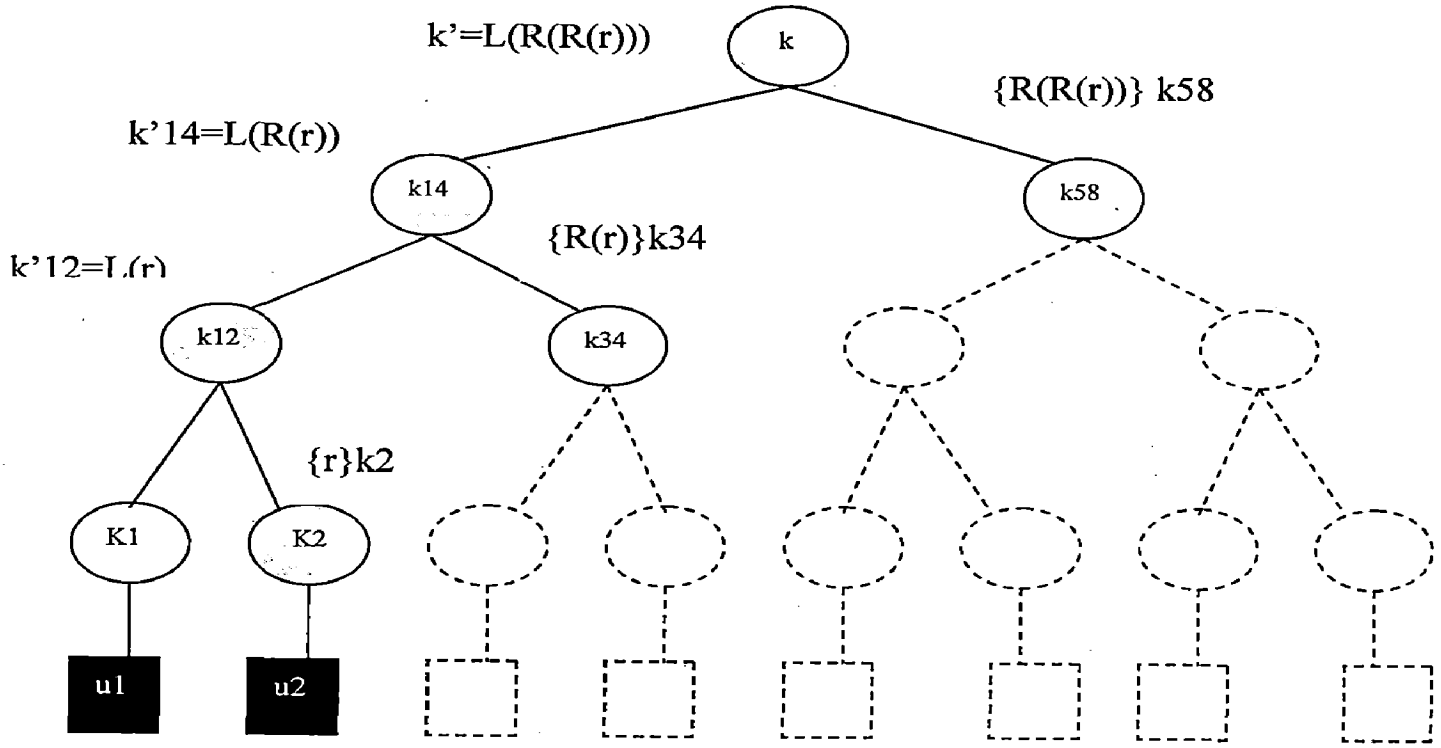


Figure 2.6 New key r is attributed to leaf K_2

2.3.5 Centralized Flat Table

Waldvogel et al.[13] extended their own solution and proposed to change the hierarchical tree for a flat table (FT) with the effect of decreasing the number of keys held by the KDC. The table has one entry for the Traffic Encryption Key (TEK) and 2w more entries for KEKs, where w is the number of bits in the member id. There are two keys available for each bit in the member id, one associated with each possible value of the bit (Table 2.1 shows an example with w = 4). A member knows only the key associated with the state of its bit. In total, each member holds w + 1 keys. For example, a member with id 0101 knows KEK0.0, KEK1.1, KEK2.0 and KEK3.0 (as shown in Table 2.1).

When a member leaves the group, all keys known by it are changed and the KDC sends out a rekey message containing two parts. The first part has the new TEK encrypted with each unchanged KEK (any member with an id with at least one single bit of difference from the leaving member's id can recover the TEK). In the second part, each of the new KEKs is encrypted with its old KEK and with the new TEK (as shown in Table 2.2). This way, every remaining member can update its old KEKs without gaining further knowledge about the KEKs other members had. This scheme is susceptible to collusion attacks. A set of evicted members, which have IDs with complementary bits, may combine their sets of keys to recover a valid set of keys and hence are able to have unauthorized access to group communication.

Table 2.1.
Flat ID assignment

	TEK	
ID Bit #0	KEK 0.0	KEK 0.1
ID Bit #1	KEK 1.0	KEK 1.1
ID Bit #2	KEK 2.0	KEK 2.1
ID Bit #3	KEK 3.0	KEK 3.1
	Bit 0	Bit 1

Table 2.2
Message to execute member 0101

TKE		
(KEK 0.0_{new}) TEK_{new}	(TEK_{new}) KEK 0.1	ID Bit #0
(TEK_{new}) KEK 1.0	(KEK 1.1_{new}) TEK_{new}	ID Bit #1
(KEK 2.0) TEK_{new}	(TEK_{new}) KEK 2.1	ID Bit #2
(TEK_{new}) KEK 3.0	(KEK 3.1_{new}) TEK_{new}	ID Bit #3
Bit 0	Bit 1	

2.4 Decentralized Group Key Management Protocol

In a decentralized subgroup approach, the large group is split into smaller subgroups. Different controllers are used to manage each subgroup thereby minimizing the problem of concentrating the work on a single place. In this approach, more entities are allowed to fail before the whole group is affected. We use the following attributes to evaluate the efficiency of decentralized frameworks:

- *Key Independence.* Each key must be completely independent from any previous used and future keys; otherwise compromised keys may reveal other keys.
- *Decentralized controller.* A centralizing manager should not manage the subgroup managers. The central manager raises the same issues as the centralized systems, namely if the centralizing manager is unavailable, the whole group is compromised.
- *Local rekey.* Membership changes in a subgroup should be treated locally, which means that rekey of a subgroup should not affect the whole group. This is also known as the 1-affects-n problem.
- *Key vs. data.* The data path should be independent of the key management path, which means that rekeying the subgroup should not impose any interference or delays to the data communication.
- *Rekey per membership.* Related to backward and forward secrecy.

- *Type of communication.* Group with a single data source are said to use 1-to-n communication, and groups with several or all members being able to transmit are characterized by using n-to-n communication.

2.4.1 Scalable Multicast Key Distribution

Ballardie [14] proposed the scalable multicast key distribution (SMKD) protocol, which uses the tree built by the Core Based Tree (CBT) multicast routing protocol [15, 16] to deliver keys to multicast group members. In the CBT architecture, the multicast tree is rooted at a main core. Also, cores can exist eventually. The main core creates an access control list (ACL). Group key and key encryption key (KEK) are used to update the group key. The ACL, the group key and the key encryption key are transmitted to secondary cores and other nodes, when they join the multicast tree after their authentication. Any router or secondary core authenticated with the primary core can authenticate joining members and use the ACL to distribute the keys, but only the main core generates those keys. The SMKD protocol does not provide the forward secrecy when a member leaves the group. It has to execute afresh each time when a member departs.

2.4.2 Intra-Domain Group Key Management Protocol

T. Hardjono et al. [17] proposed the Intra-domain Group Key Management Protocol GKMP. Architecture divides the network into administratively scoped areas. There is a Domain Key Distributor (DKD) and many Area Key Distributors (AKDs). Each AKD is responsible for one area. Figure 2.7 exemplifies this architecture. The group key is generated by the DKD and is propagated to the group members through the AKDs. The DKD and AKDs belong to a multicast group called All-KD-Group. The DKD uses this group to transmit rekey messages to the AKDs who rekey in turn their respective areas. This architecture suffers from a single point of failure, which is the DKD that is the entity responsible for generating the group key. Besides, in case of an AKD failure, members belonging to the same area will be not able to access the group communication.

2.4.3 Hydra Protocol

Rafeli et al. [18] proposed Hydra protocol, wherein the group is organized into smaller subgroups and a server called the Hydra server (HSi) controls each subgroup i. If a membership change occurs at subgroup i, the corresponding HSi generates the group key and sends it to the other HSj involved in that session. In order to have the same group key distributed to all HSs, a special protocol is used to ensure that only a single valid HS is generates the new group key whenever required. Figure 2.8 depicts the Hydra architecture.

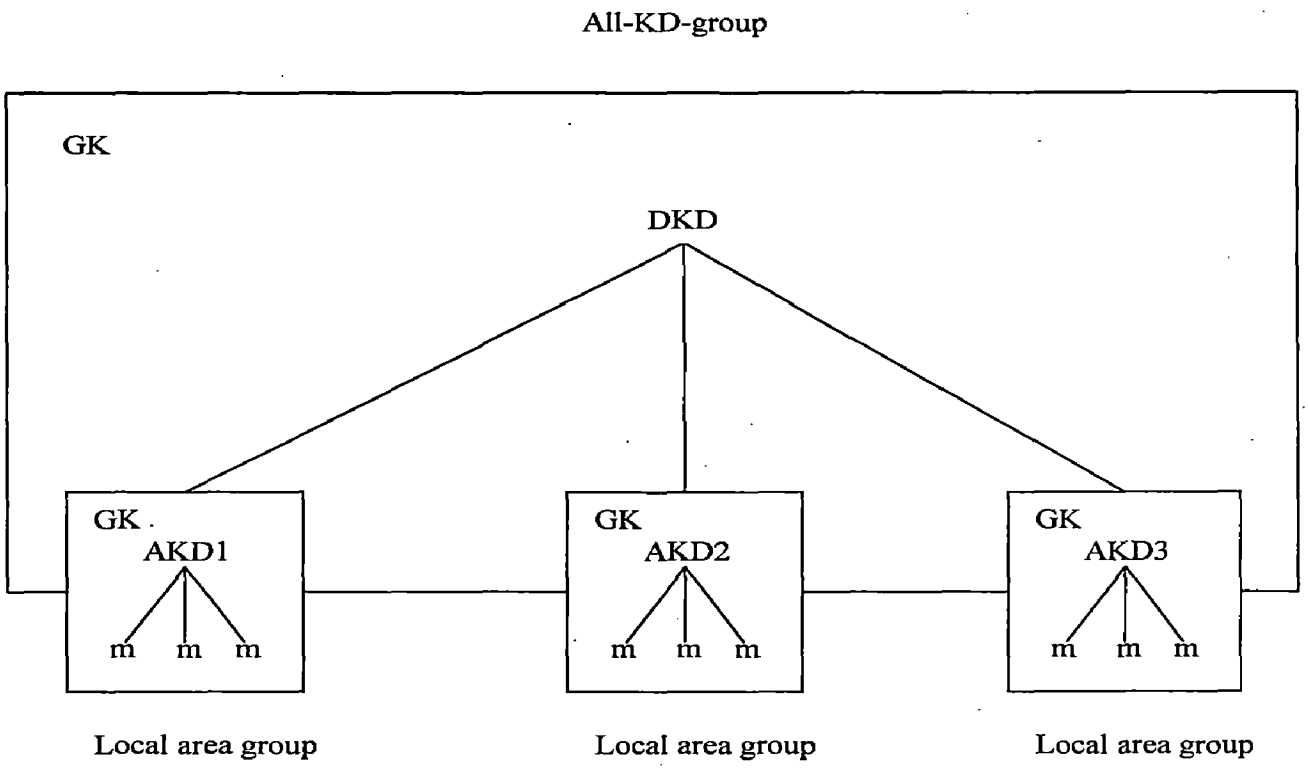


Figure 2.7 Intra-domain Group Key Management Protocol Architecture

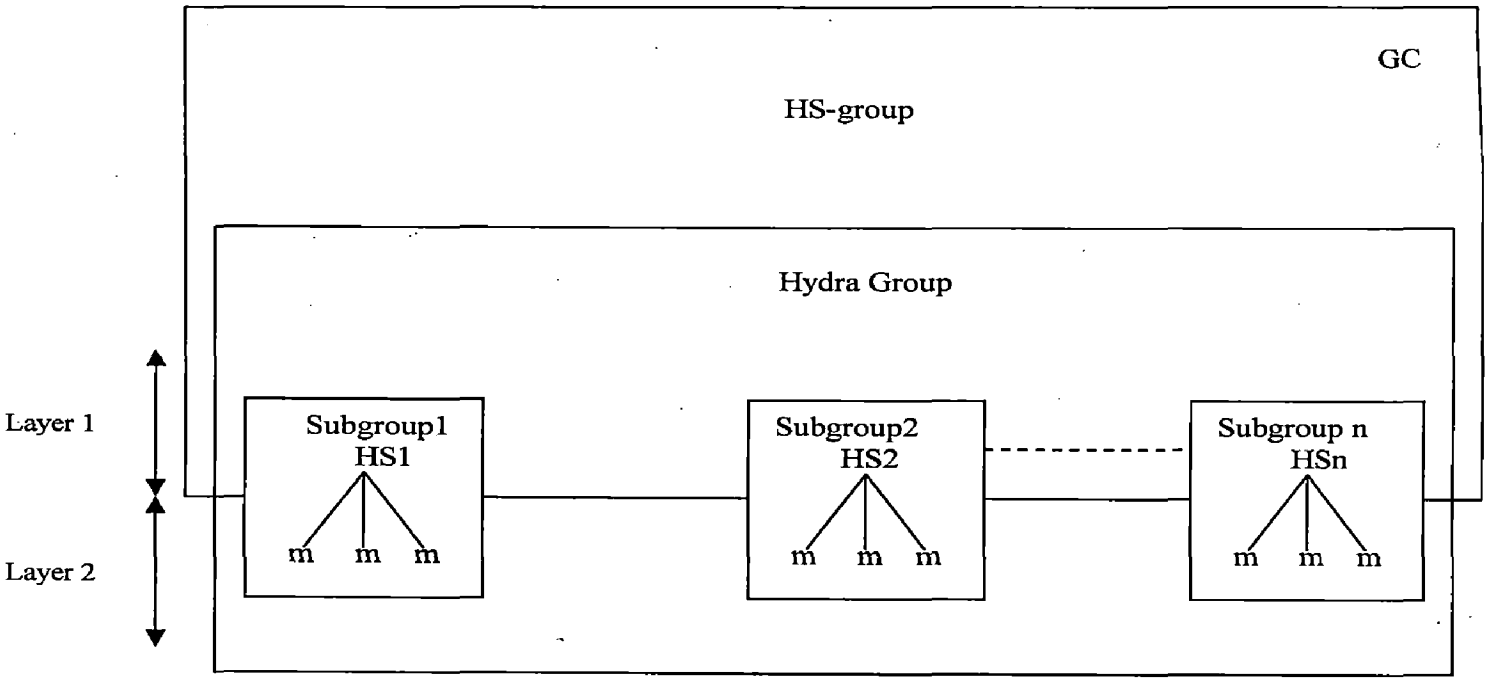


Figure 2.8 Hydra Architecture

2.4.4 Baal Protocol

Chaddoud et al. [19] proposed a protocol that is known as Baal protocol, which defines three entities:

- The Group Controller (GC): It maintains a participant list and creates and distributes the group key to group members via local controllers.
- Local Controllers (LC): The GC delegates a LC to each subnet (generally a local network) to manage the keys within its subnet. When a LC receives a new group key, it distributes it to the members connected to its subnet. Besides, a LC can play the role of the GC by generating and distributing new group keys after membership changes following some coordination rules.
- Group member: It belongs to participation list. When a membership change occurs at a subnet, the corresponding LC can generate a new group key and distribute it to its subnet and to the other members via their LCs. To assure that a single LC generates a new group key at a time, the GC assigns a priority to each LC and when many LCs

distribute simultaneously a new group key, the LCs are instructed to commit to the group key issued by the LC having the highest priority.

2.4.5 MARKS Protocol

In MARKS protocol, Briscoe [20] suggest slicing the time length to be protected (such as the transmission time of a TV program) into small portions of time and using a different key for encrypting each slice. The encryption keys are leaves in a binary hash tree that is generated from a single seed. The internal nodes of the tree are also called seeds. A blinding function, such as MD5, is used on the seed to create the tree in the following way:

- First, the depth D of the tree is chosen. The depth, D, defines the total number (N) of keys ($N = 2^D$).
- Then, the root seed, $S_{0,0}$, is randomly chosen. In $S_{i,j}$, i represents the depth of the seed in the tree, and j is the number of that key in level i.
- After that, two intermediate seeds (left and right) are generated. The left node is generated by shifting $S_{0,0}$ one bit to the left and applying the blinding function on it (S1, 0 = b(ls($S_{0,0}$))). The right node is generated by shifting $S_{0,0}$ one bit to the right and applying the blinding function on it (S1, 1 D b(rs($S_{0,0}$))).
- The same algorithm is applied to the following levels until the expected depth is reached.

Users willing to access the group communication receive the seeds needed to generate the required keys. For example, Figure 2.9 shows a binary hash tree of depth 3. If a user wants to participate in the group from time 3 to 7, it would be necessary to have only two seeds: $S_{3,3}$, as K_3, and $S_{1,1}$, to generate K_4 till K_7. This system cannot be used in situations when a membership change requires change of the group key, since the keys are changed as a function of the time.

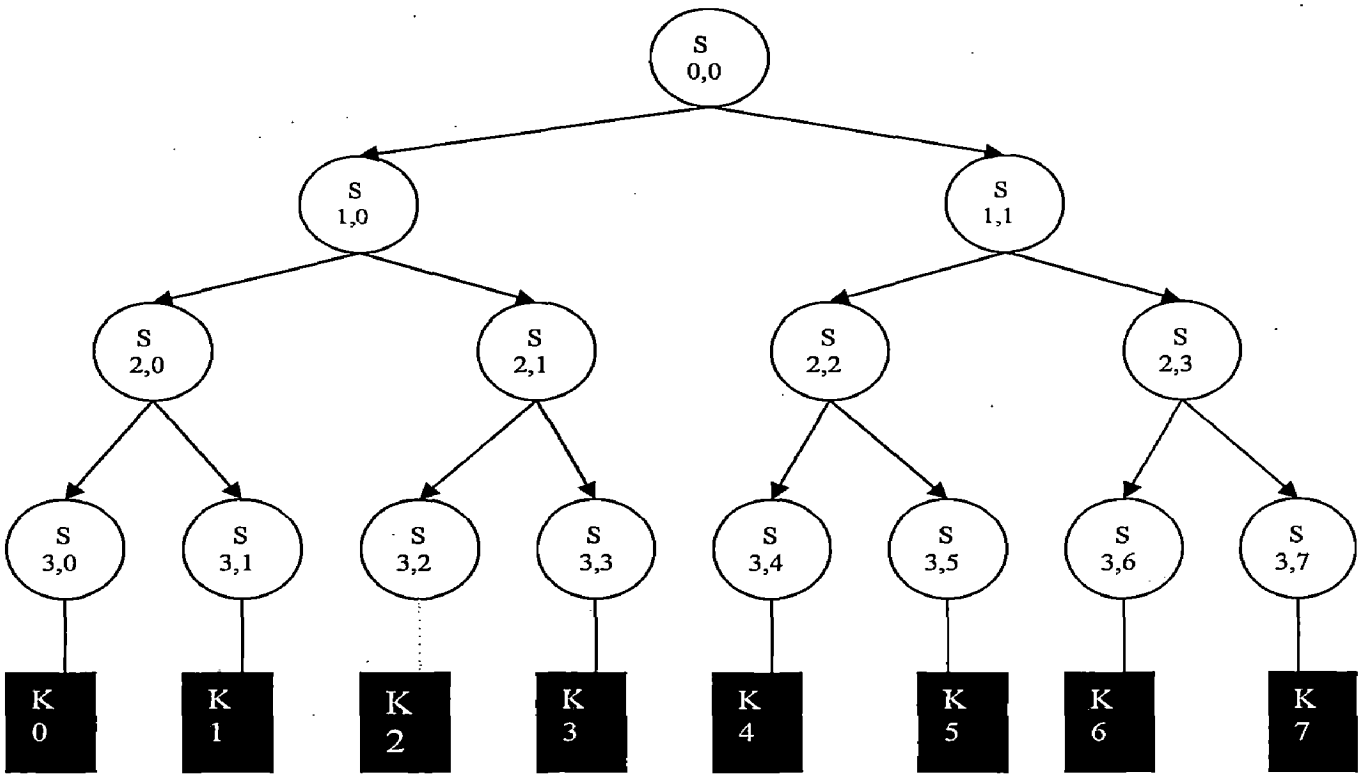


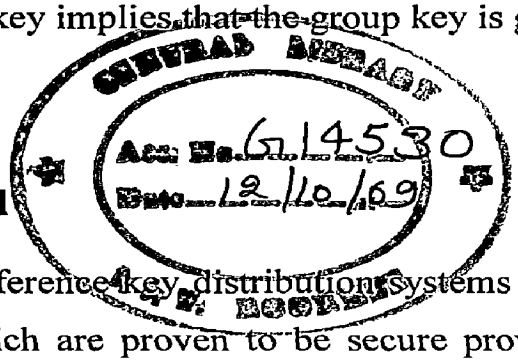
Figure 2.9 Binary Hash tree

2.5 Distributed Group Key Management Protocol

The distributed key management approach is characterized by having no group controller. The group key can be either generated in a contributory fashion, where all members contribute their own share to computation of the group key, or generated by one member. In the latter case, although it is fault tolerant, it may not be safe to leave any member to generate new keys since key generation requires secure mechanisms, such as random number generators, that may not be available to all members. Moreover, in most contributory protocols (apart from tree-based approaches), processing time and communication requirements increase linearly in term of the number of members. Additionally, contributory protocols require each user to be aware of the group membership list to make sure that the protocols are robust.

The following attributes are used to evaluate the efficiency of distributed key management protocols:

- Number of rounds: The protocol should try to minimize the number of iterations among the members to reduce processing and communication requirements.
- Number of messages: The overhead introduced by every message exchanged between members produces unbearable delays as the group grows. Therefore, the protocol should require a minimum number of messages.
- Processing during setup: Computations needed during setup time. Setting up the group requires most of the computation involved in maintaining the group, because all members need to be contacted.
- DH key: Identifying whether the protocol uses Diffie–Hellman (DH) to generate the keys. The use of DH to generate the group key implies that the group key is generated in a contributory fashion.



2.5.1 Burmester and Desmedt Protocol

Burmester et al. [21] proposed a practical conference key distribution systems based on public keys, which authenticate the users which are proven to be secure provided the Diffie-Hellman problem is intractable. A certain number of interactions is needed but the overall cost is low. But there is a complexity tradeoff. Depending on the network used, it either have a constant (in the number of conference participants) number of rounds or a constant communication and computation overhead. It is a very efficient protocol that executes in only three rounds:

1. member mi generates its random exponent r_i and broadcasts $Z_i = \alpha^{ri}$;
2. member m_i computes and broadcasts $X_i = (Z_{i+1} / Z_{i-1})^{ri}$;
3. member mi can now compute key $k = Z^{nri}_{i-1} \cdot X^{n-i}_{\ i} \cdot X^{n-2}_{\ i+1} \ldots X_{i-2} \bmod p$.

The BD protocol requires n + 1 exponentiations per member and in all but one the exponent is at most n - 1. The main drawback is the requirement of 2n broadcast messages.

2.5.2 Group Deffie-Hellman Key Exchange

Group Diffie–Hellman key exchange [2] is an extension of the DH key agreement protocol that supports group operations. The DH protocol is used for two parties to agree on a common key. In this protocol, instead of two entities, the group may have n members. The group agrees on a pair of primes (q and fi) and starts calculating in a distributive fashion the intermediate values. The first member calculates the first value (α^{x1}) and passes it to the next member. Each subsequent member receives the set of intermediary values and raises them using its own secret number generating a new set. A set generated by the i^{th} member will have i intermediate values with i - 1 exponents and a cardinal value containing all exponents.

3.1 Related Work

Different approaches have been proposed for distributed batch rekeying for group communications. Wong et al. [22] proposed the key tree approach for secure group communications. They suggested to associate keys in a hierarchical tree and rekey at every join or leave event. They provided this key tree approach for the solution to the scalability problem of the group key management. Later Steve et al. [23] introduced the concept of batch rekeying to enhance system efficiency since the rekeying workload is independent of membership dynamics. All the above approaches rely on a centralized key server, which is responsible for generating and distributing new keys.

Steiner et al. [2] used first to address dynamic membership issues in group key agreement and proposed a family of Group Diffie-Hellman (GDH) protocols (GDH1, GDH2 and GDH3) based on straightforward extensions of the two parties Diffie-Hellman (DH). GDH provides contributory authenticated key agreement, key independence, key interiority and resistance to known key attacks. These protocols are not very efficient for the following reasons. First, 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. And the last one is the group leader will have to do O(n) exponentiation operations on every membership change events. This causes a large delay in the formation of the new group key.

The TGDH protocol [3] 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. Though the TGDH protocol is very efficient, it loads the members of the SGCS because of the 2D (D is the depth at which a new member is added to the tree or an old member is removed from the tree) serial exponentiation operations per membership change. This causes a lot of delay in resuming normal group communication.

Lee et al. [4] described three interval based distributed rekeying algorithms: Rebuild algorithm, the Batch algorithm and the Queue-batch algorithm. The use of interval based rekeying aims to maintain good rekeying performance independent of the dynamics of joins and leaves. These algorithms were based on the following assumptions: The key tree of TGDH is used as a foundation of all the algorithms, the rekeying operations are carried out at the beginning of every rekey interval and When a new member sends a join request, it should also include its individual blinded key. The first two algorithms perform rekeying at the beginning of every rekey interval, which can result in a high processing load during the update instance and therefore delay the start of the secure group communication.

3.2 Research Gaps

As per the reviewing of different existing methods, we have found the following research gaps which we will address in our dissertation work.

1. In [2], 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.
2. In [2], the group leader will have to do O(n) exponentiation operations on every membership change events. This causes a large delay in the formation of the new group key.
3. In [3], it loads the members of the SGCS because of the 2D (D is the depth at which a new member is added to the tree or an old member is removed from the tree) serial exponentiation operations per membership change. This causes a lot of delay in resuming normal group communication.
4. Also, there is no solution for minimizing the average waiting time for the joining and leaving group members.

4.1 Introduction

In this chapter, we give details of our proposed algorithm for efficient implementation of secure GCSs. The proposed algorithm for implementation of a group communication system is based on the balanced binary tree. One of the main issues of this algorithm is to maintain the balance of the binary tree, so that each member needs to store minimum number of keys, which is equal to the depth of the binary tree, in order to minimize the storage requirement of each member. We form a secure chain among the group members using DH protocol where each member M_i share its secret with its adjacent neighbours in the secure chain and compute the shared key called *leftkey* and *rightkey*. Using this method, we can provide message authentication because every member receives only encrypted messages from another member with whom it share a secret key. In our method, we compute keys corresponding to the tree nodes in a bottom-up manner in such a way that the loads at each member are properly distributed. The algorithm enables the group members to agree upon a common group key with minimum computation overhead. All group communication traffic is encrypted with this common key so that only group members can recover original group messages. There are two kinds of group key management algorithm: key distribution and key agreement. In key distribution algorithms, a trusted entity securely distributes the group key to the group members. In key agreement algorithms, there is no trusted entity for generating the group key. The group members agree upon a common group key by communicating among themselves, as due to their fault tolerant nature, there is no need for a trusted third party. Here main goal of this dissertation are:

1) To built a fully distributed group key management system;
2) To minimize the number of messages exchange among group members during the group key generation;
3) To minimize the storage requirement at each member by properly maintaining the balance of the key tree. If the tree becomes unbalanced then the storage requirement

among the group members are varies (as shown in Figure 4.1) and it will become large when the group is associated with a large number of members.

4) To minimize the computational requirement by maintaining the balance of the tree. When the tree becomes unbalanced the numbers of decryptions are large if any of its siblings leave (as shown in Figure 4.1).
5) To minimize the average waiting time of the joining member in the group by replacing the departed members with the joining members within the rekeying interval.
6) To minimize the cost by maintaining the balance of the tree.

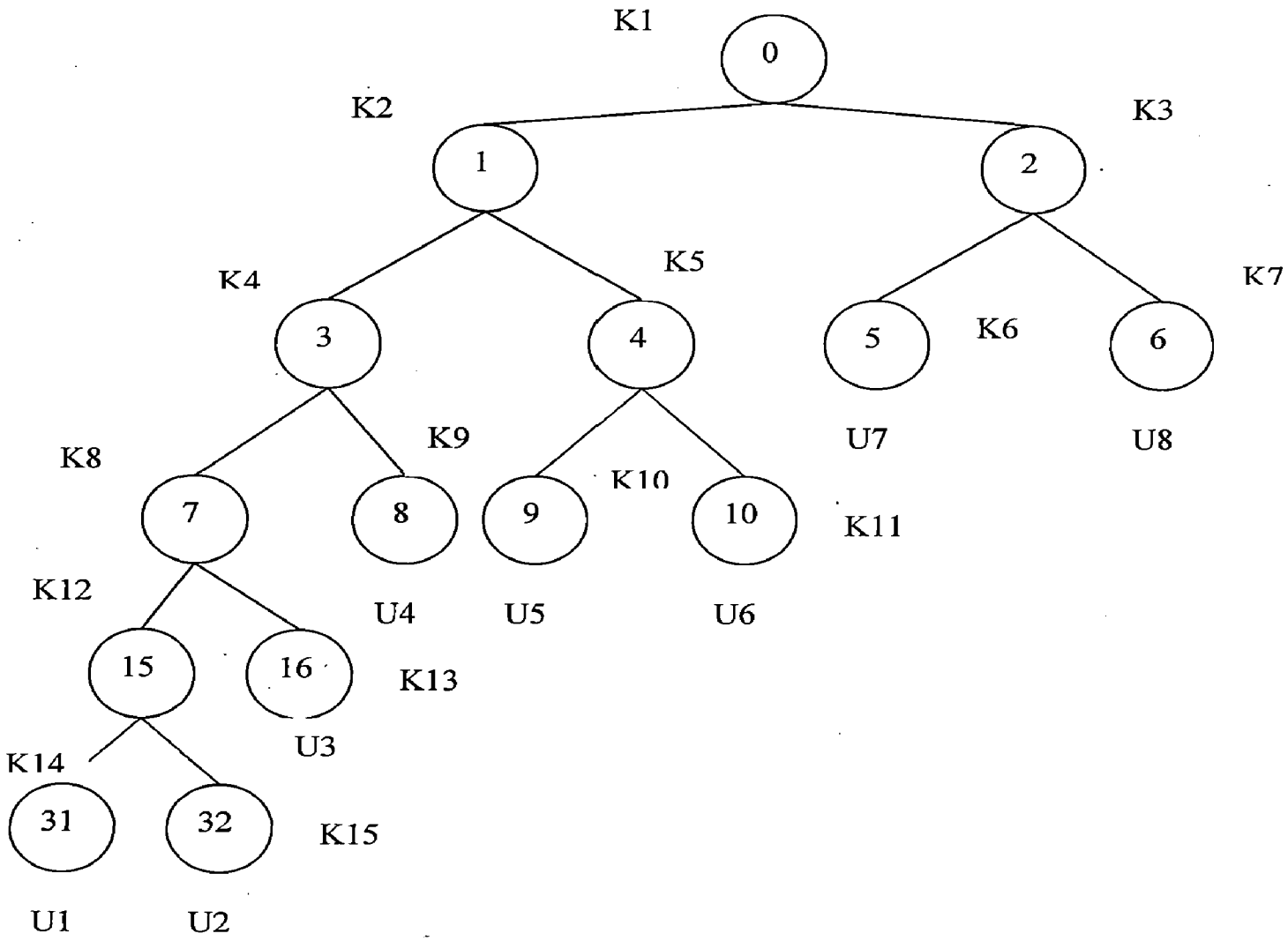


Figure 4.1 Unbalanced key tree

In the above Figure 4.1, it can be seen that key storage among the group members varies from three to six rather than four in a balanced key. Also it can be seen that member U1 and U2 needs five decryptions if any of its siblings leaves rather than three decryptions in a balanced key tree. Finally, the rekeying cost is nine messages when U1 or U2 departs since K1, K2, K4, K8 and K12 need to be changed. In this example, the difference between balanced and unbalanced key tree varies slightly as the multicast group is small. In scenario such as pay-per-view where the multicast group membership varies from thousands to millions of members, unbalanced key tree might lead to significant computation efforts for both the members.

4.2 Informal Description of Algorithm

The notations used in this section are listed in Table 4.1.

Table 4.1

List of notations

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 K
$\{c\}_K^{-1}$	Decryption of cipher text c using key K
P	The DH modulus. Both p and (p-1) / 2 are prime
g $(g < p)$	The DH generator of order p-1 modulo p
α_i	Member M_i 's long term private secret
$g^{\alpha i}$ mod p	Member M_i 's public key
H_{MIN}	Minimum height of leaf node in key tree
H_{MAX}	Maximum height of leaf node in key tree
H_{INSERT}	H_{MIN} of ST_A - H_{MAX} of ST_B
$H_{MIN\_ST\_A}$	H_{MIN} of ST_A
$H_{MAX\_ST\_A}$	H_{MAX} of ST_A

$H_{MIN\_ST\_B}$	H_{MIN} of ST_B
$H_{MAX\_ST\_B}$	H_{MAX} of ST_B

The main issue of the algorithm is to maintain a balance binary key tree T at all members. The leaf nodes of the tree 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 this key. Each internal node of the binary tree has exactly two children. The tree is balanced in the sense that the difference in depths of any two leaf nodes is at most one. It is a much stronger requirement for balancing the tree. The tree is securely built using the 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.

Every node nd of T is associated with the following variables listed in Table 4.2.

Table 4.2

List of 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].

4.3 Group Key Agreement

The algorithm proceed in two phase. The members are arranged in a logical line.

Phase 1 In the first phase, every member M_i engages in a DH key agreement protocol with every other M_j ($| i - j | = 1$). At the end of this phase, every pair of adjacent members M_i, M_{i+1} ($1 \leq i \leq n$) will share a secret key. The two keys a member Mi shares with its two neighbours are known locally as leftkey and rightkey as shown in Figure 4.2.

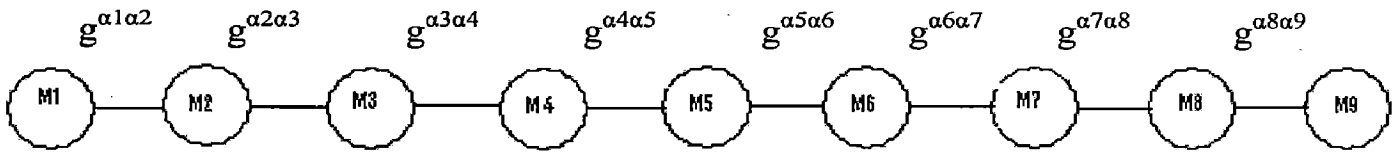


Figure 4.2 Formation of the DH chain

Phase 2 In the second phase, a balanced binary tree is built in a distributed fashion in such a way that every node knows only the keys at nodes along the path from itself to root [T]. In Figure 4.3, the darkened nodes are the ones whose keys are known to members M_1 and M_2. The dashed line represents the secure channels formed in stage 1.

The key 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). For example, for node nd in Figure 4.3, key[nd] is generated after generating key[left[nd]] and key[right[nd].

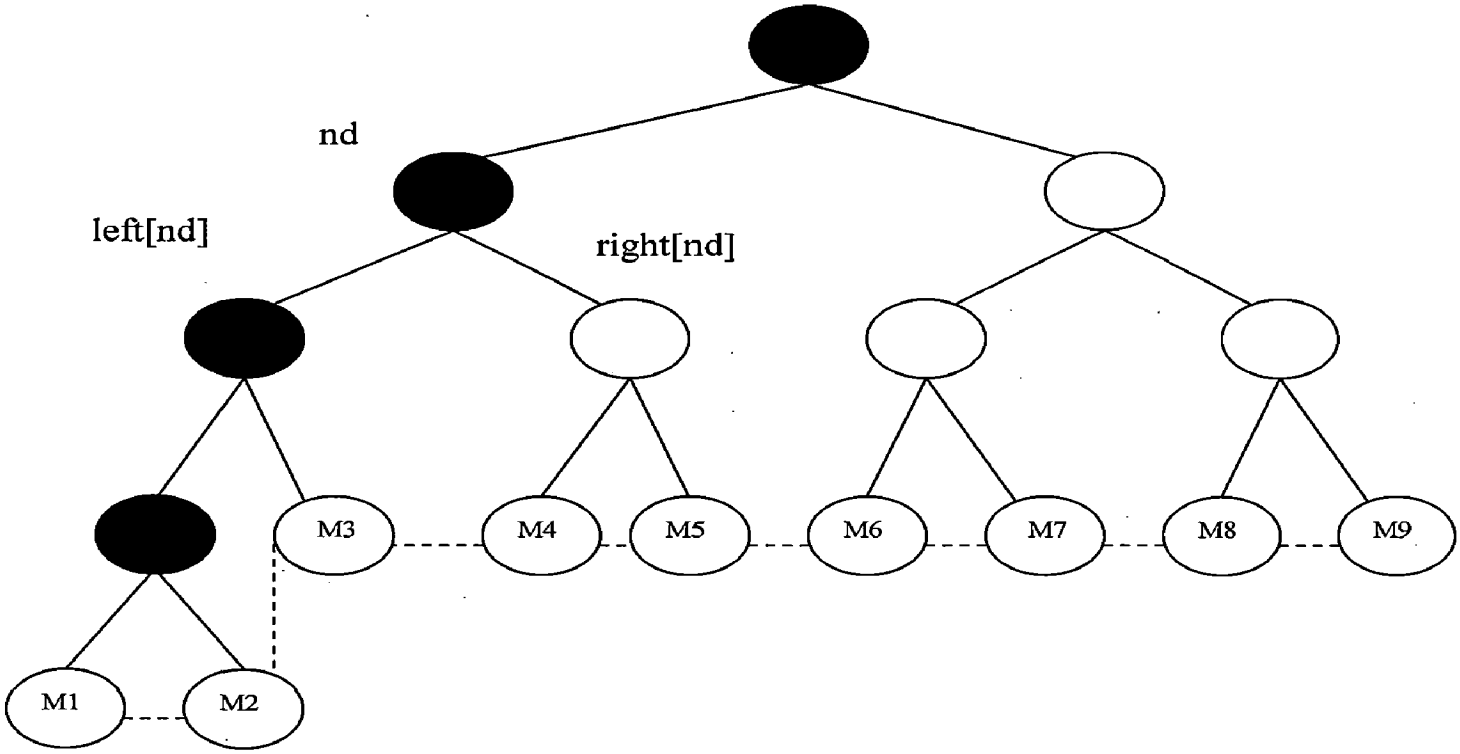


Figure 4.3 The key tree

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 decrypt it using its leftkey and multicast {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].

4.4 Merging of Two balanced Tree

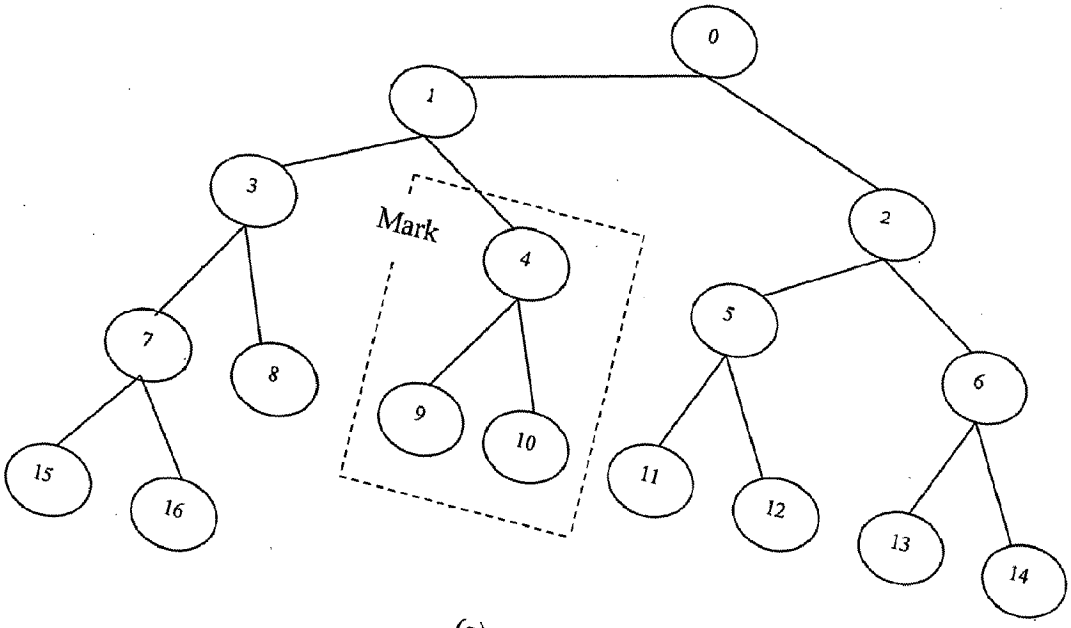
We have used two merging algorithms [24], which are suitable for a batch join event. Both merging algorithms insert the joining members at suitable height to create a balanced key tree. We first assume that we need to combine two key trees, ST_A and ST_B, where ST_A has its height greater than ST_B.

Method 1 This algorithm is only used when the difference in the maximum height between the two key trees, ST_A and ST_B, is greater or equal to one. The algorithm works as follow:

If the difference between HMAX_ST_A and HMIN_ST_B is greater than one and the difference between HMAX_ST_A and HMAX_ST_B is greater or equal to one, the algorithm calculates HMAX_ST_B level up from HMIN_ST_A, provided the resultant height is greater than zero. If the resultant height is zero, the child key node of the root is selected. The selection of the key node at H_{INSERT} is based on the one with the most number of leaf nodes on the minimum height from that particular key node to the leaf nodes. If more than one node in the H_{INSERT} has same number of leaf nodes in the minimum height, then we select the left most node among them. Marking is done on the selected key node. The algorithm creates a new key node at the old location of the marked key node and inserts the marked key node and ST_B as its children.

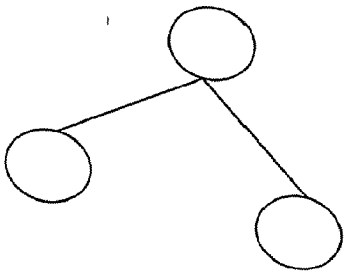
Figure 4.4 shows an example, where we have a balanced key tree ST_A (as shown in Figure 4.4 (a)) with 9 members. Assume that two member wish to join in the group. These two members form a new key tree (ST_B) as shown in Figure 4.4(b). Here, the ST_B will be adding to the 2 level of ST_A. Finally, it creates a new node with node ID 4 and inserts the marked key tree and the new key tree as shown in Figure 4.4(C).

Figure 4.4 (a) shows the original key tree (ST_A). Let two members want to join the group. We first form a tree ST_B using these two joining members as shown in Figure 4.4 (b). According to the Method 1 the ST_B key tree will be inserted at the level 2 of ST_A. Here nodes 4, 5, and 6 have the same numbers of maximum leaf nodes at lowest height of ST_A. In this situation, we chose the node with lowest node ID. Insert a new node at this place and make the subtree rooted at node 4 as its left child and new tree ST_B as its right child.

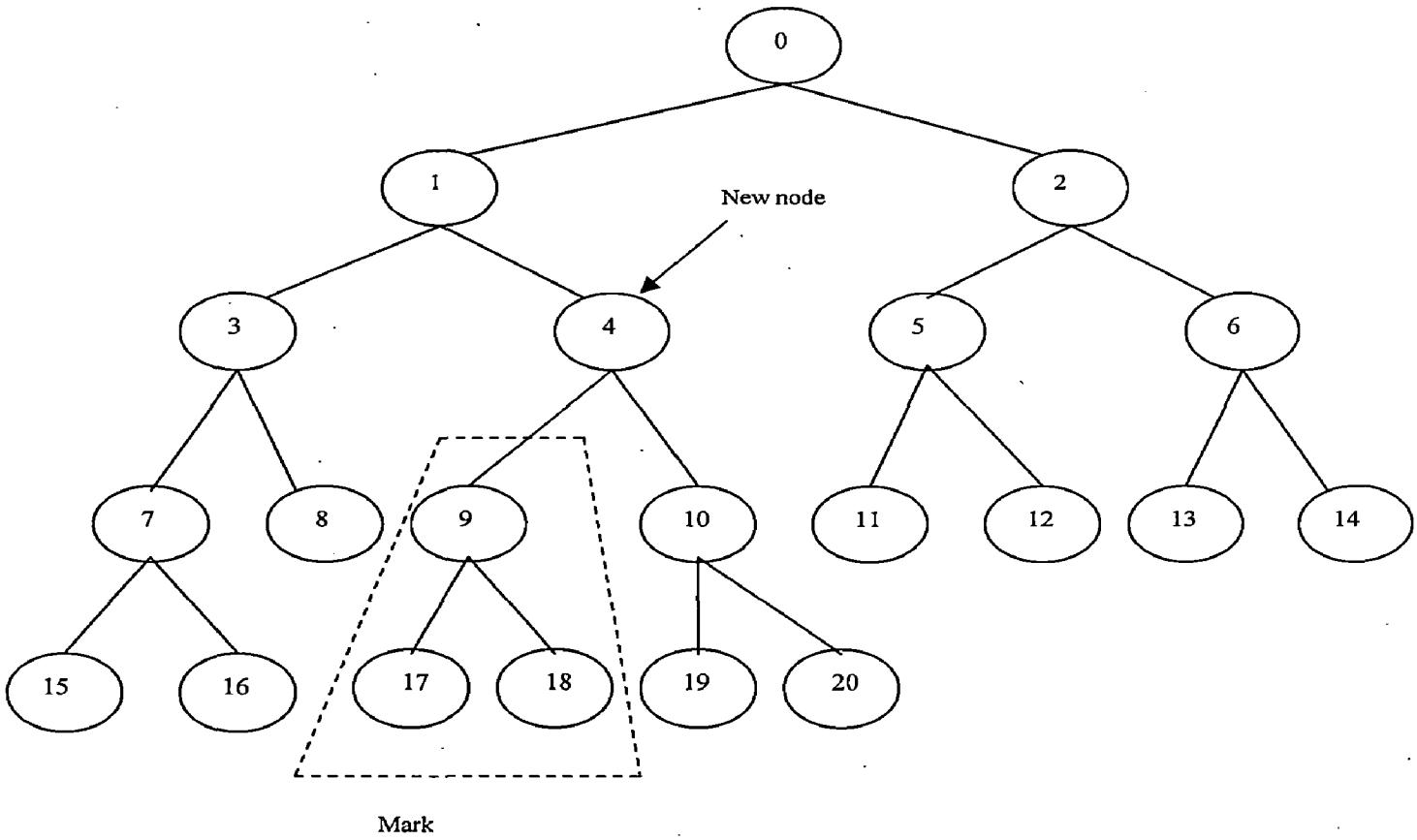


(a) ST_A key tree

+



(b) ST_B key tree



(c) Resultant key tree

Figure 4.4 (a) ST_A key tree, (b) ST_B key tree, and (c) Resultant key tree

Method 2 This algorithm is only used for combining key trees with the same maximum height or the difference in maximum height between them is not more than one. If the difference between $H_{MAX\_ST\_A}$ and $H_{MIN\_ST\_B}$ and the difference between $H_{MAX\_ST\_A}$ and $H_{MAX\_ST\_B}$ are similar or equal to one, then the Method 2 is suitable for combining them together. Finally, it creates a new key node at the root and inserts ST_A and ST_B as its children.

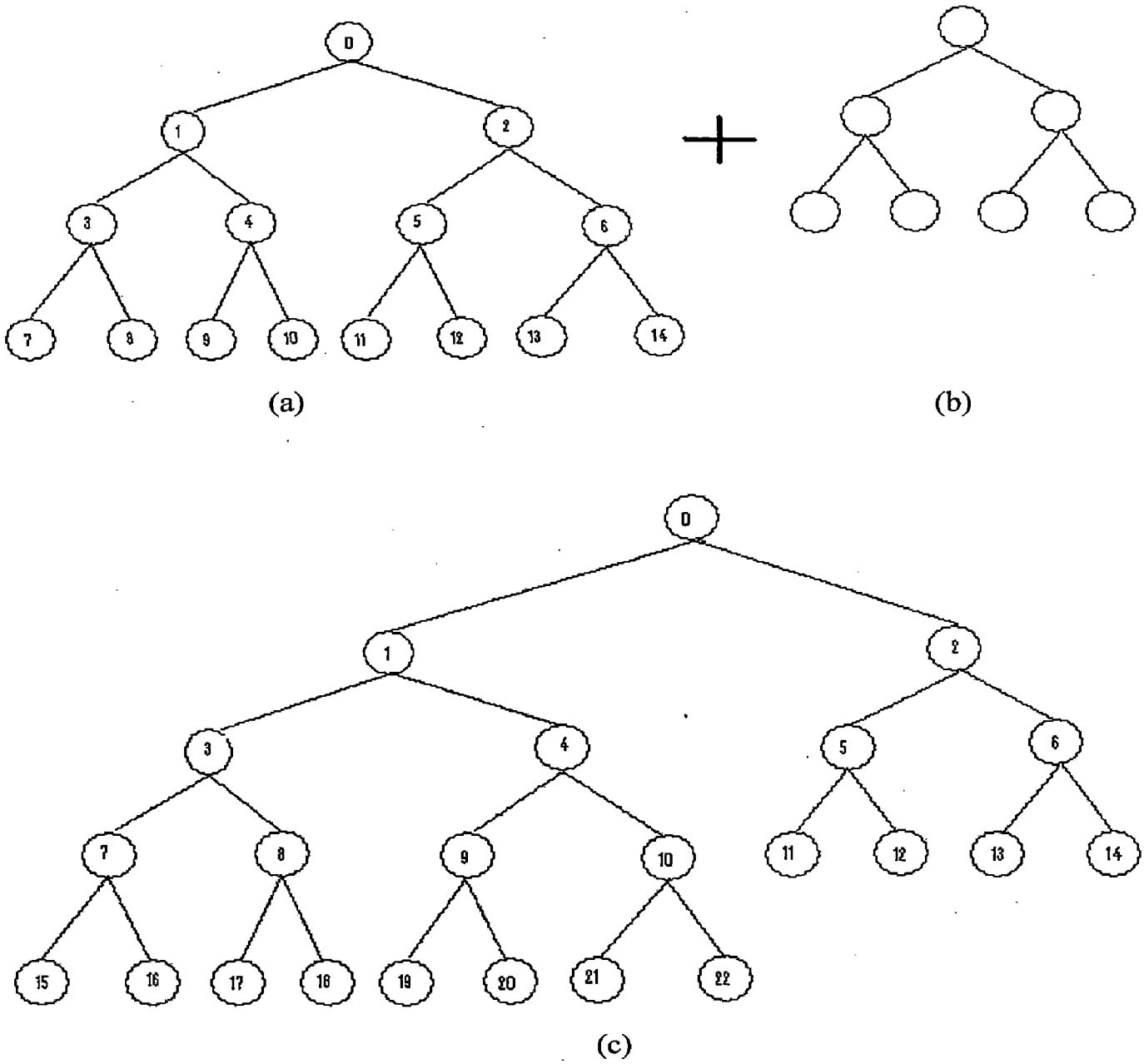


Figure 4.5 (a) ST_A key tree, (b) ST_B key tree and (c) resultant key tree.

Suppose that four members want to join in the group. First, form a new tree ST_B using these joining members as shown in Figure 4.5 (b). Now, according to the Method 2 ST_B will be added at 0^{th} level of ST_A. So, create a new node and make it as the new root of the key tree and insert ST_A and ST_B as its left and right child respectively. Figure 4.5 (c) shows the resultant key tree.

4.5 Batch Rekeying Algorithm

Here, we used a hybrid batch rekeying method, which depend on fixed interval period and fixed number of join and leave, to maintain group of join and leave. In this method, when a member wants to leave from the group and there exist some member, those want to join the group, the departed member is replace instantly by the waiting member. Use this technique we can improve the average waiting time of the joining members. Now we describe how the group of joins and leaves are handling in this method.

Case 1: When (J > L) /* where J and L used to represent the joining and leaving members respectively */

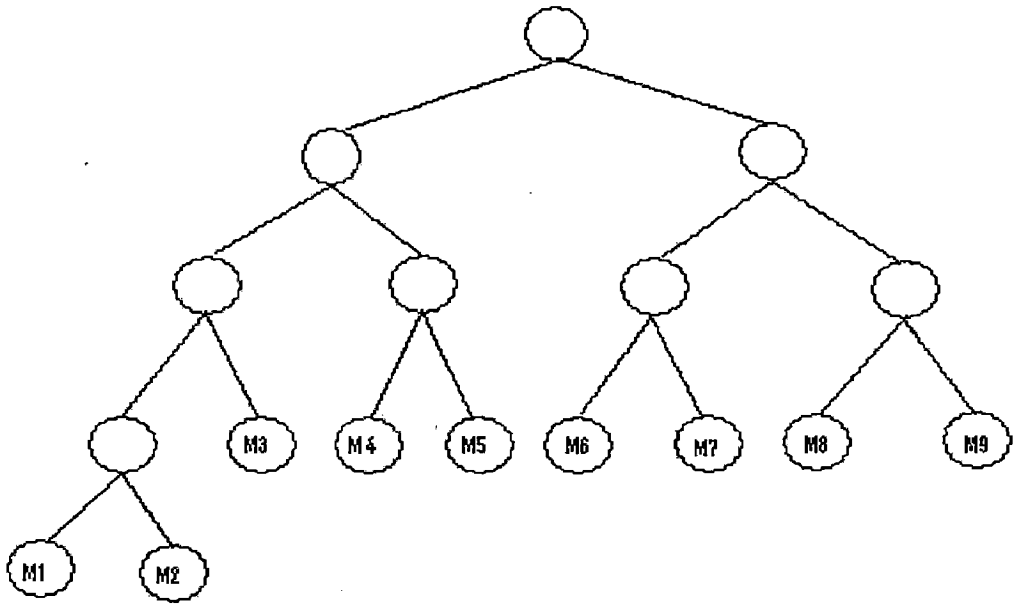
1. If (L = 0)
    a. Create a new tree T' using the new joining members.
    b. Called the merging algorithm to merge the original tree T with T'.
    c. All members are reassigned IDs.
2. If (L ≠ 0)
    a. Replace all leaving member by the newly joining members.
    b. Using remaining joining members create a new tree T'.
    c. Called the merging algorithm to merge the original tree T with T'.
    d. All members are reassigned IDs.

Case 2: When (J < L)

1. If (J = 0)
    a. Remove all departed member.
    b. Check the balance of the resulting tree.
        If the tree remains balanced then only reassign the IDs.
        Else maintain the balance of the tree using the balancing algorithm, which will be describe later in this section. And after that reassign the IDs.
2. If (J ≠ 0)
    a. All joining members replace same number of departed members.
    b. The remaining departed members are removed
    c. Later parts are same as 1 b.

When the group of members joins, the keys of the nodes along the path from the leaf nod corresponding to the new members to the root are changed as follows. Let N be the set of nodes such that the subtrees rooted at each node nd $\in$ N contains the new members M_i as one of its leaf nodes. Every member M_j ($j \neq i$) belongs to the subtrees rooted at k $\in$ N replaces key[nd] with its has function. 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 members securely.

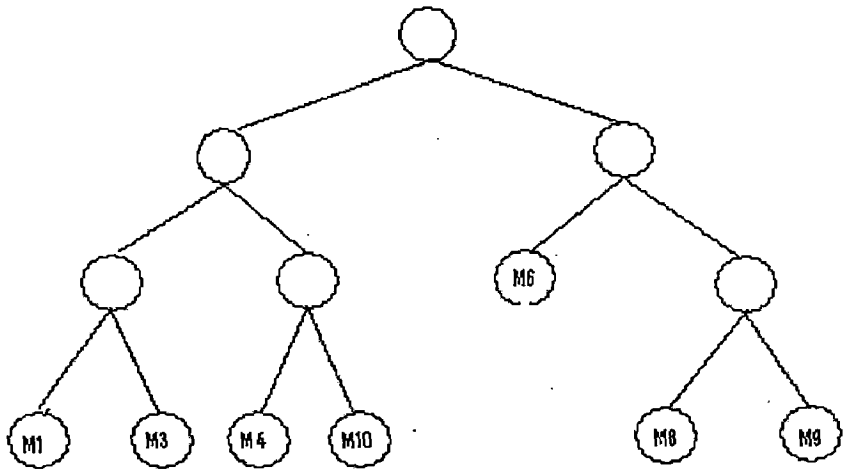
Now, we give some example to explain the above operations. Figure 4.6 explain what happened when L > J.



(a)

M_2, M_5 and M_7 leave

M_{10} Join



(b)

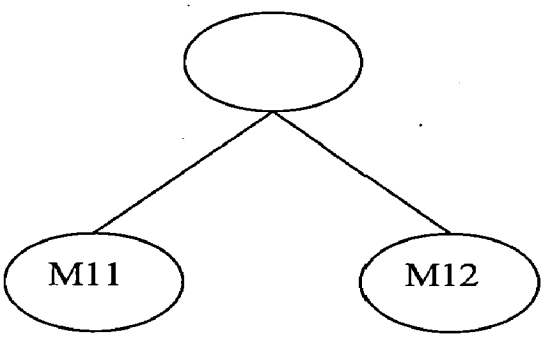
Figure 4.6 When M_2, M_5 and M_7 leave and M_{10} join the group

Example:

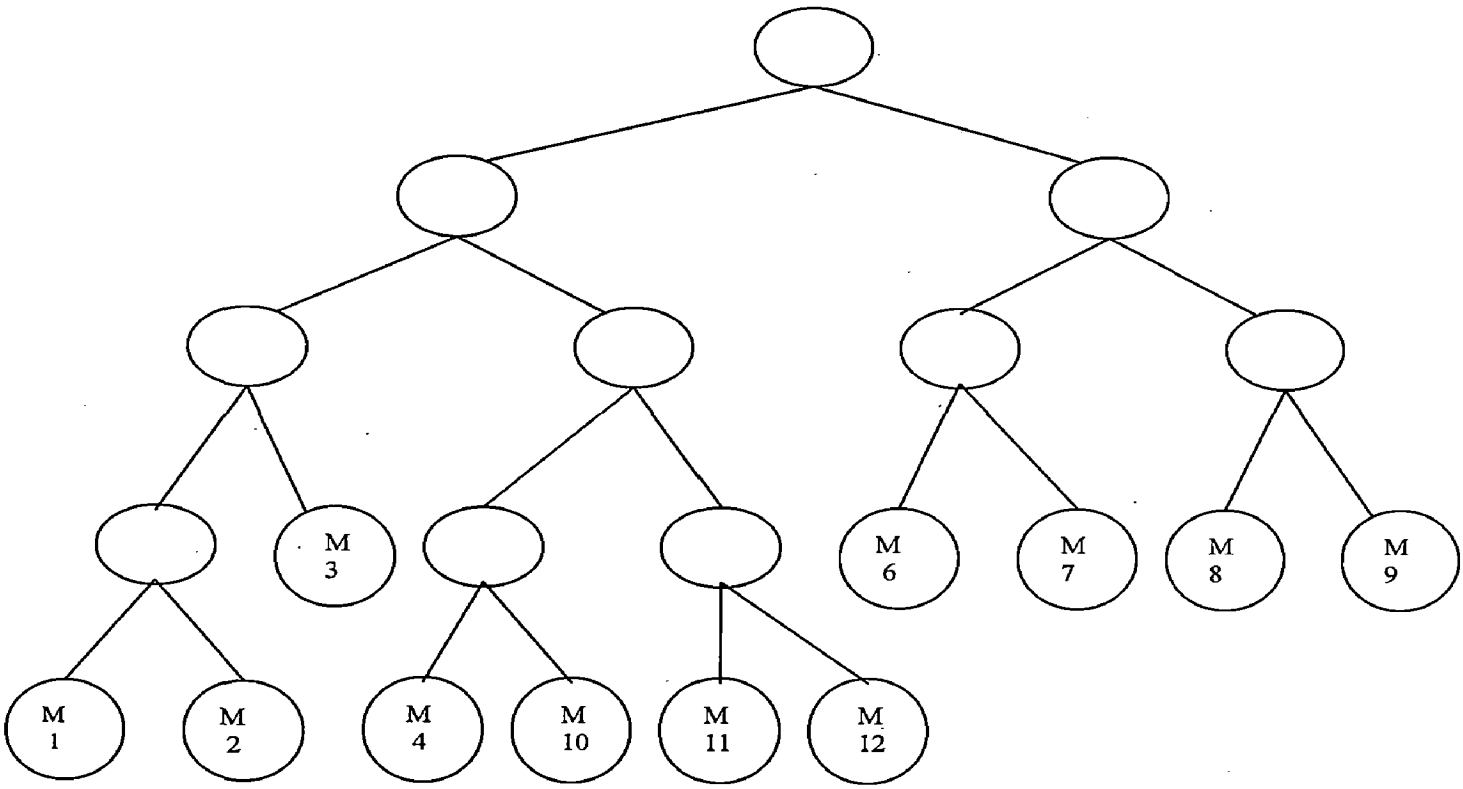
In this example the group consists of nine members as shown in Figure 4.6 (a). Let member M_2, M_5 and M_7 wants to leave the group and a new member M_{10} want to join the group. According to the above method M_5 will be immediately replace by the new joining member M_{10}, because M_5 has the lowest node ID among the leaving members. Joining members are added to the lowest node ID so that it can properly maintain the balance of the key tree. After removing the member M_2 and M_7 the resultant key tree will be as shown in Figure 4.6 (b). After time up of rekeying interval group members are reassign the node IDs and reform the broken chain by executing DH. We have no control over the leaving members. So if the tree becomes unbalanced after removing then find the node at maximum height and remove it from that place and add it to the node which is at lowest height as its child. Repeat this process until the tree become balance.

Figure 4.7 explain what happened if J > L. Let the group consists of nine members as shown in Figure 4.6 (a). Let member M_5 wants to leave the group and new members M_{10}, M_{11} and M_{12} wants to join the group. In that case according to the above algorithm M_{10} will replace member M_5 immediately within the rekeying interval. After that the remaining joining members here M_{11} and M_{12} form a new tree as shown in Figure 4.7 (b). Now using the merging algorithm as described in section 4.4 we join the two trees. Here

we use the method 1 one for joining the two trees. After time up of rekeying interval reassign the ID of the group members.



(b)



(c)

Figure 4.7 (b) tree of joining members and (c) Resultant key tree

The effectiveness of our scheme was evaluated by simulation using a JAVA based JiST (Java in Simulation Time) simulator.

5.1 JiST Simulator

Java in Simulation Time (JiST): JiST is a new Java-based discrete-event simulation engine, with a number of novel and unique design features [25]. It is a prototype of a new general-purpose approach to building discrete event simulators, called virtual machine-based simulation that unifies the traditional systems and language-based simulator designs. The resulting simulation platform is more efficient. It out-performs existing highly optimized simulators both in time and memory consumption.

The JiST system architecture, depicted in Figure 5.1, consists of four distinct components: a compiler, a byte code rewriter, a simulation kernel and a virtual machine. JiST simulation programs are written in plain, unmodified Java and compiled to byte code using a regular Java language compiler. These compiled classes are then modified, via a byte code-level rewriter, to run over a simulation kernel and to support the simulation time semantics described shortly. The simulation program, the rewriter and the JiST kernel are all written in pure Java. Thus, this entire process occurs within a standard, unmodified Java virtual machine (JVM). The benefits of this approach to simulator construction over traditional systems and languages approaches are numerous [25].

Embedding the simulation semantics within the Java language allows reuse of a large body of work, including the Java language itself, its standard libraries and existing compilers. JiST benefits from the automatic garbage collection, type-safety, reflection and many other properties of the Java language. This approach also lowers the learning curve for users and facilitates the reuse of code for building simulations. The use of a standard virtual machine provides an efficient, highly-optimized and portable execution platform and allows for important cross-layer optimization between the simulation kernel

and running simulation. Furthermore, since the kernel and the simulation are both running within the same process space it reduces serialization and context switching overheads. In summary, a key benefit of the JiST approach is that it allows for the efficient execution of simulation programs within the context of a modern and popular language. JiST combines simulation semantics, found in custom simulation languages and simulation libraries, with modern language capabilities. This design results in a system that is convenient to use, robust and efficient.

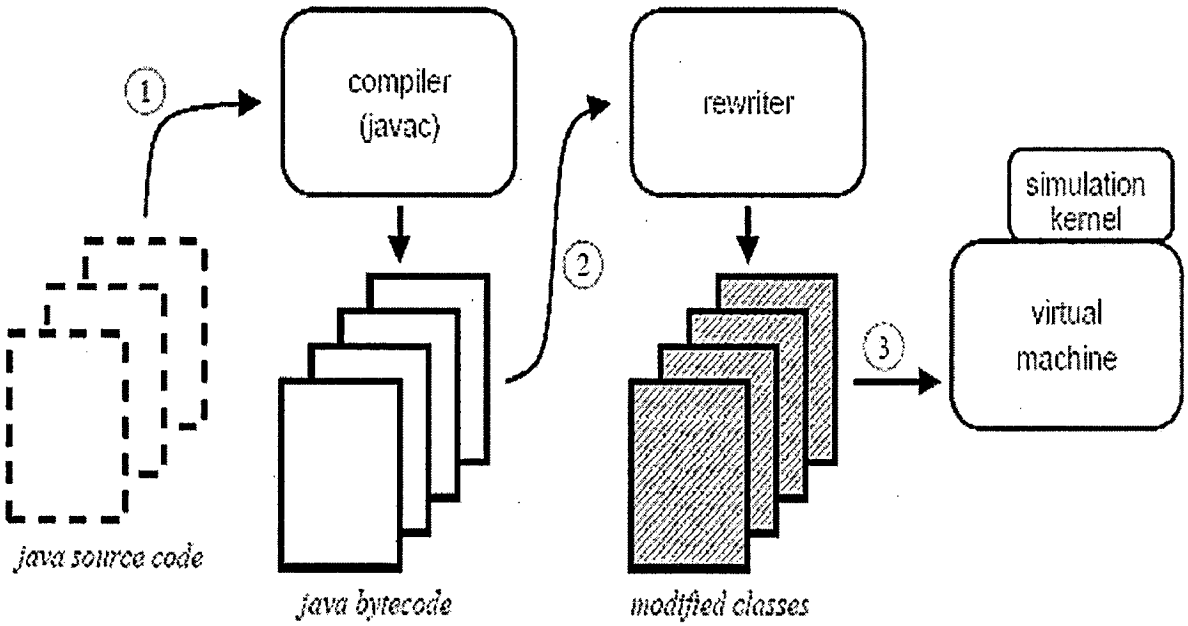


Figure 5.1 JiST system architecture

5.2 Simulation Parameter

In this section, we list the various simulation parameters we used in our simulation scenarios. These are given in Table 5.1.

Table 5.1
List of simulation parameters

Parameter	Value
Slot Interval	200 sec
Time Between Addition and Deletion	15 sec
Randomness	Equal number of addition and deletion, more addition, more deletion
Simulation Time	5000 sec
Initial Number of Nodes	90

5.3 Results

Our proposed 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 relies on the Diffie-Hellman protocol. 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 signed messages is absent. The 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.

In case of join, let a new member M wants to join the group. Then, consider 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 Diffie-

Hellman(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 neighbors (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 leftkey.

When our key change 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 this algorithm. There are three types of delays incurred proposed 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 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

Therefore, a member joins operation cause a maximum delay of 2d + 2*l* 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 (where D is the depth at which new member is join) serial exponentiations and broadcasts the modified key tree (with new values of blind keys for the nodes along the path from the joining member's node to the root). Therefore, for the TGDH protocol, the delay is d (2D + 3) + 2*l* and the messages passed include two unicasts and one broadcast.

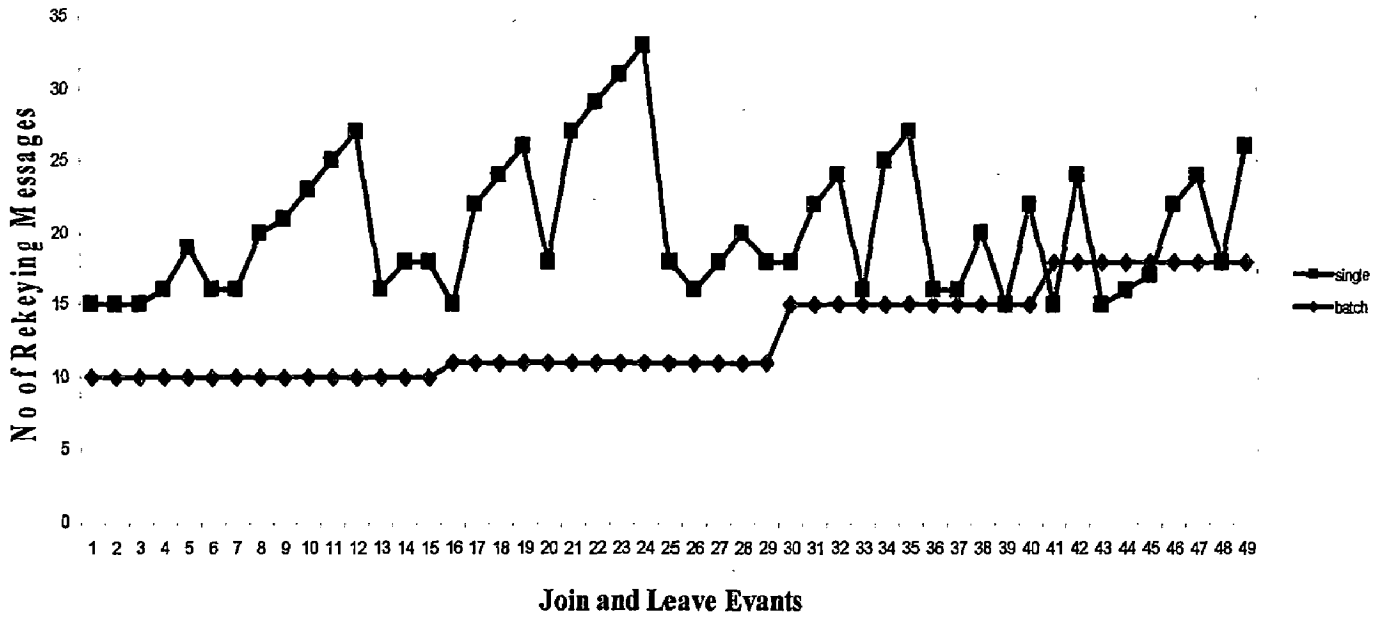


Figure 5.2 Effect of batching operation on rekeying

Figure 5.2 shows how the numbers of rekeying messages are reduced in case of batch operation with respect to the single join and leave operations. Therefore, using the batch operation, we can effectively minimize the computational cost. For example, consider two leaves that happen one after another. The keys need to be changed in each leave to preserve the forward secrecy. These two leave may happened so close to each other that the first set of new keys are actually not used and are immediately replaced by the second set of new keys. So, when request are frequent many new keys may be generated and distributed, but not used at all. This is the worst case of computational cost. In Figure 5.2 the single join and leave events case [6] require 5080 messages and in the case of batch join and leave events case require 2998 rekeying messages, which is substantially lower than the previous value.

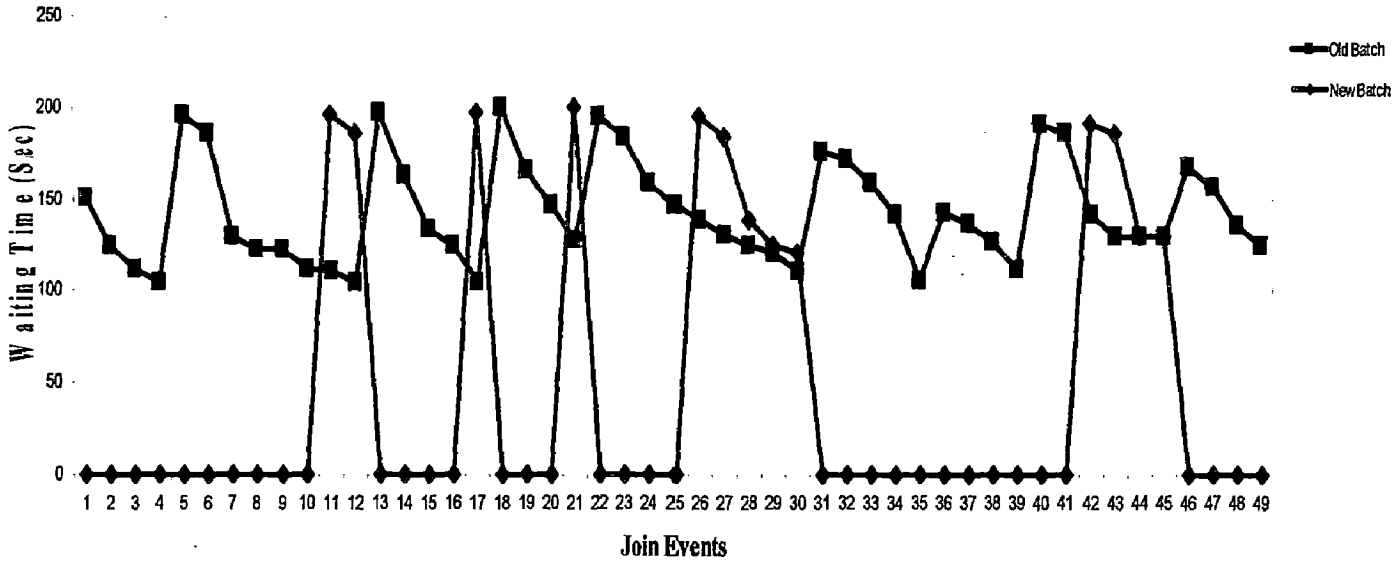


Figure 5.3 Effect of replacement of leave by join within rekeying interval

Figure 5.3 shows the effect on waiting time when a leaving member is immediately replaced by a joining member within the rekeying interval than replacing after rekeying interval. If we replace the leaving member by a joining member after the rekeying interval, then some form of forward secrecy is scarifies and each joining member need to wait for the rekeying interval time period. But in our case in which we replace the leaving member by a joining member within the rekeying interval time period, waiting time of each member will be less than previous work [4]. But, in our case some form of backward secrecy is scarifies instead forward secrecy. In previous work, the average waiting time of the joining members is 141 seconds where as, in our case, the average waiting time of joining members is 44 seconds which is reduced by 69 %.

6.1 Conclusion and Future Work

The motivation behind the dissertation work was to design a group communication system to reduce the computational overhead of each group members, storage requirement, message delivery delay and average waiting time of the joining members. In order to achieve these goals, we proposed a group key agreement algorithm which manages the group key with minimum computational overhead at each group members. The algorithm maintains a balanced distributed key tree at the group members using which group key can be changed efficiently whenever the group membership changes. This algorithm gives an easiest solution to the message authentication because each member receives messages encrypted with the key which it shares only with its adjacent neighbours. It also reduces the message delivery delay present in the existing TGDH protocol. We saw that our algorithm substantially minimizes the rekeying messages than the rekeying after individual join and leave. Our algorithm minimize the average waiting time of the joining members by allowing the joining members to replace the leaving members within the rekeying interval, in contrast in previous batch rekeying method [4] leaving members are replace by the joining members after time out of rekeying interval.

The proposed algorithm has a scope of extension. Future work may consider developing a more efficient tree balancing algorithm. In group communication, there is no control over leaving member and may cause the tree to be unbalanced. In our proposed algorithm, we have used two tree balancing methods: one for balancing the unbalanced tree due to leaving of members and other for merging two balanced trees where one is added in an appropriate position of other tree. So instead of two algorithms, one balancing algorithm may be developed to handle these two cases simultaneously.

REFERENCES

[1] Sandro Rafaeli, David Hutchison, "*A Survey of Key Management for Secure Group Communication*", ACM Computing Surveys, pp. 309–329, Published by ACM New York, NY, USA, 2003.

[2] Steiner, M., Tsudik, G., and Waidner, M., "*Diffie-Hellman Key Distribution Extended to group Communication*", 3rd ACM Conference on Computer and Communications Security, pp. 31-37, Published by ACM 1996.

[3] Yongdae Kim, Adnan Perrig, and Gene Tsudlk, "*Tree-Based Group Key Agreement*", ACM Transactions on Information and System Security, pp. 60-96, Published by ACM, 2002

[4] Patrick P. C. Lee, John C. S. Lui, David K. Y. Yau, "Distributed Collaborative Key Agreement Protocols for Dynamic Peer Groups", 10th IEEE International Conference on Network Protocols, pp. 322-333, Published by IEEE Computer Society, 2002.

[5] Mcdaniel, P., Prakash, A., and Honeyman, P., "*A Flexible Framework for Secure Group Communication*", In Proceedings of the 8th USENIX Security Symposium, pp. 99–114, 1999.

[6] S. Rahul, R. C. Hansdah, "*An Efficient Distributed Group Key Management Algorithm*", 10th International Conference, pp. 230, Published by IEEE Computer Society, 2004.

[7] Kim, Y., Perrig, A., and Tsudik,. "*Simple and Fault Tolerant Key Agreement for Dynamic Collaborative Groups*", In Proceedings of the 7th ACM Conference in Computer and Communication Security, pp. 235–241. 2000

[8] Harney, H. and Muckenhirn, C., "*Group Key Management Protocol (GKMP) Specification*", RFC 2093, 1997a.

[9] Harney, H. and Muckenhirn, C., "*Group KeyManagement Protocol (GKMP) Architecture*", RFC 2094, 1997b.

[10] Wong, C. K., Gouda, M.G., and Lam, S. S., "*Secure Group Communications Using Key Graphs*", IEEE/ACM Transactions on Networking (TON), pp. 16–30, Published by IEE, 2000.

[11] Mcgrew.D.A, and Sherman.A.T, "*Key Establishment in Large Dynamic Groups Using One Way Function Trees*", IEEE Transactions on Software Engineering , pp. 444 – 458, Published by IEEE, 2003.

[12] R. Canetti, J. Garay, G. Itkis, D. Micciancio, M. Naor, B. Pinkas, "*Multicast Security: A Taxonomy and Some Efficient Constructions*", Eighteenth Annual Joint Conference of the IEEE Computer and Communications Societies, pp. 708-716, Published by IEEE, 1999.

[13] Waldvogel. M, CARONNI. G., Sun. D., Weiler. N., and Plattner. B., "The Versa Key framework: Versatile group key management", Journal on Selected Areas in Communications: Special Issue on Middleware, pp. 1614–1631, Published by IEEE 1999.

[14] A. Ballardie, "*Scalable Multicast Key Distribution*", RFC 1949, 1996.

[15] A. Ballardie, "*Core Based Trees (CBT version 2) Multicast Routing Protocol Specification*", RFC 2189, 1997.

[16] T. Ballardie, I.P. Francis, and J. Crowcroft, "*Core Based Trees: An Architecture for Scalable Inter-domain Multicast Routing*", ACM SIGCOMM, pp. 85–95, 1993.

[17] T. Hardjono, B. Cain, and I. Monga, "*Intra-Domain Group Key Management for Multicast Security*", IETF Internet draft, pp. 324-332, 2000.

[18] S. Rafaeli and D. Hutchison., "Hydra: a decentralized group key management", 11th IEEE International WETICE: Enterprise Security Workshop, pp. 62-67, Published by IEEE Computer Society, 2002.

[19] G. Chaddoud, I. Chrisment, and A. Shaff, "*Dynamic Group Communication Security*", 6th IEEE Symposium on Computers and Communication, pp. 49, Published by IEEE Computer Society, 2001.

[20] B. Briscoe, "*MARKS: Multicast key management using arbitrarily revealed key sequences*", 1st International Workshop on Networked Group Communication, pp. 301-302, 1999.

[21] Burmester,M. and Desmedt, Y., "A Secure and Efficient Conference Key Distribution System (extended abstract)", In Advances in Cryptology EUROCRYPT , Lecture Notes in Computer Science, pp. 275–286, 1994.

[22] C. K. Wong, M. Gouda, and S. S. Lam, "Secure Group Communications using Key Graphs", IEEE Transactions on Networking, pp. 16-30, Published by IEEE press, 2006.

[23] Xiaozhou Steve Li, Yang Richard Yang, Mohamed G. Gouda and Simon S. Lam "*Batch Rekeying for Secure Group Communications*", 10th International Conference on World Wide Wave, pp. 525 – 534, Published by ACM New York, NY, USA, 2001

[24] Wee Hock Desmond Ng, Haitham Cruickshank, Zhili Sun, "*Scalable Balanced Batch Rekeying for Secure Group Communication*", Computers and Security, pp. 265-273, Published by Science Direct, 2006.

[25] R. Barr, Z. J. Haas, and R. van Renesse, "*Jist: An Efficient Approach to Simulation using Virtual Machines*", Software Practice & Experience, pp. 539–576, 2005.

APPENDIX

Algorithm for Batch Rekeying in Distributed Environment:

The following are some functions used in the algorithm.

- A sequence of number i… j can be divided into two groups as follows
  Low (i, j) = (a, b), where a = i, b = $\lfloor i + (j - 1) / 2 \rfloor$,

  High (i, j) = (a, b), where a = $\lceil i + (j - i) / 2 \rceil + 1$, b = j.
- The topple (first[nd], last[nd]) associated with a node nd is referred to by the notation 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(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
  (x_1, y_1), id(left[st]) ← low(i, j)
  (x_2, y_2), id(right[st]) ← high(i, j)
  CONSTRUCT_BT(x_1, y_1, left[st])
  CONSTRUCT_BT(x_2, y_2, right[st])
 end if

Group key Agreement

Let M_i ($1 \leq i \leq n$) be the i^{th} member in the group. Every pair of adjacent members sharing a secret key and every internal node $node_{a,b}$ representing a key shared by members Mi ($a \leq i \leq b$). There are three kinds of external events at each memberM_i

1. $SEND_{i,j}$(msg): Sending of a message msg from M_i to M_j
2. $MCAST_{i,(a,b)}$(msg): Sending of a message msg from M_i to all M_j ($a \leq j \leq b$)
3. $RECV_{j,i}$ (msg): Receipt of a message at M_i from M_j

The algorithm proceeds in two 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

- $M_i \rightarrow M_{i-1}$ ($1 \leq i < n$) $g^{\alpha i} \bmod p$
- $M_i \rightarrow M_{i-1}$ ($1 < i \leq n$) $g^{\alpha i} \bmod p$
- M_i ($1 \leq i < n$) rightkey $\leftarrow g^{\alpha i\, \alpha i+1} \bmod p$
- M_{i-1} ($1 < i \leq n$) leftkey $\leftarrow g^{\alpha i-1\, \alpha i} \bmod p$

Phase 2 Every member M_i executes the following algorithm

```
x ← par[Mi]
while x ≠ nil do {key at root node is not yet generated}
  l ← left[x]
  r ← right[x]
  if i = last[l] then {right most leaf node of l}
    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

x ← par[x]

end while

Group Key Change

The Group key management protocol has to change the group key whenever the group membership changes. It is initiated on the occurrence of any one of the following two events: First when new members want to join the group and second, when existing members has to be removed from the group. To minimize the average waiting time of the joining members here we used BATCH_KEY( ) function is defined below. In this function we used two queue *ql* for storing leaving members and *qj* for joining members. Within the rekeying interval when a members want to join the group, if there exit any member who want to leave the group immediately replace by the new member using REPLACE (M_l, M_j, st) function. After time up the rekeying interval, if there exist only joining members form a new tree using the remaining joining members and merge the new tree with the old one using the ADD ( ) function. Otherwise if there exist only leaving members then delete them form the group using DELETE_NODE (ql[frontl], T) function.

BATCH_KEY( )

while(timeslice)

if frontl ≠ rearl

if frontj ≠ rearj

while frontl ≠ rearl or frontj ≠ rearj

M_l = ql[frontl]

M_j = qj[frontj]

```
            REPLACE(Ml, Mj, st)
            frontl = frontl + 1
            frontj = frontj + 1
          end while
        end if
      end if
    if(frontl ≠ rearl)
      DELETE_NODE(ql[frontl], T)
    end if
    if frontj ≠ rearj
     ADD ( )
     SHIFT_RIGHT(hnode)
    end if
```

Here the leaving member M_l is replaced by the M_j joining member.

```
REPLACE(Ml, Mj, st)
  While st ≠ nil
    if last[left[st]] > Ml
      st = left[st]
    else
      st = right[st]

    new ←NEW_NODE( )
    par[new] ← par[st]
  end while
```

The ADD ( ) function is used to form the new tree using the remaining joining members and merge or insert it within the old key tree in order to properly maintain the balanced of the key tree.

ADD ( )

STA_B = CONSTRUCT_BT(frontj, rearj, st)

if ($H_{MAX\_ST\_A} - H_{MIN\_ST\_B}$) > 1 and ($HMAX_{ST\_A} - H_{MAX\_ST\_B}$) ≥ 1 then

$H_{INSERT} = H_{MIN\_ST\_A} - H_{MAX\_ST\_B}$

iter = H_{INSERT}

if iter = 0 then

iter = iter + 1

end if

NODE_COUNT (st, iter)

new ← NEW_NODE ( )

par[new] ← par[hnode]

left[new] ← hnode

right[new]← ST_B

par[ST_B] ← new

par[hnode] ← new

else if ($H_{MAX\_ST\_A} - H_{MIN\_ST\_B}$) ≤ 1 and ($HMAX_{ST\_A} - H_{MAX\_ST\_B}$) ≤ 1 then

new ← NEW_NODE ( )

par [ST_A] ← new

par [ST_B] ← new

left [new] ← ST_A

right [new] ← ST_B

end if

The NODE_COUNT (st, i) function is used to count the number of leaf nodes at the lowest level corresponding to the each node at the level H_{INSERT} where the new tree will be add. And find the node which has maximum number of leave nodes at lowest level.

NODE_COUNT (st, i)

max = 0

hnode = nil {used to find the inserting node }

if i ≠ 0

```
    NODE_COUNT (left[st], i-1)
    NODE_COUNT (right[st], i-1)
  else
    j = last [st] – first [st]
    temp = st
    count = 0
    while j ≠ 0
      if depth[list[j]] = H_MIN_ST_A
        count = count + 1
      end if
      j = j – 1
    end while
    if max < count
      max = count
      hnode = temp
    end if
  end if
```

The SHIFT_RIGHT(st) function is used to reassigned the member ID after the joining of members in the group.

```
SHIFT_RIGHT(st)
  first[par[st]] = first[st]
  last[par[st]] = last[right[par[st]]] – first[right[par[st]]] + 1 + last[st]
  l = last[st]
  temp =  right[par[st]]
  first[temp] = l + 1
  last[temp] = last[par[temp]]
  NO_NEW(temp)
  temp = par[temp]
  while par[temp] ≠ nil
```

```
    temp = par[temp]
    first[temp] = first[left[temp]]
    last[temp] = last[left[temp]] + last[right[temp]] – first[right[temp]] + 1
  end while
  no = last[right[temp]] – first[right[temp]]
  first[right[temp]] = last[left[temp]] + 1
  last[right[temp]] = first[right[temp]] + no
  NO_NEW(temp)
```

NO_NEW (st) function is used to assigning the ID to the new members that form the new tree which one was merging with the old one.

```
NO_NEW ( st)
  nnodes = last[left[st]] – first[left[st]]
  first[left[st]] = first[st]
  last[left[st]] = first[left[st]] + nnodes
  nnodes = last[right[st]] – first[right[st]]
  first[right[st]] = last[left[st]] + 1
  last[right[st]] = first[right[st]] + nnodes
  if first[st] ≠ last[st] then
    SHIFT_RIGHT(left[st])
    SHIFT_RIGHT(right[st])
  end if
```

The DELETE_NODE (M, T) function given below 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 in the function DELETE_NODE (M, T) itself. The DELETE_NODE (M, T) function makes use of the following functions which are also given below, to re-balance the tree

DELETE (M, T): This function deletes the node M from the tree T and returns the sibling node of M and a boolean value indicating whether M is a left child of its parent or not. After deletion, the first and last values of the nodes of T are readjusted.
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 used to maintain the balance of the tree.

```
DELETE_NODE (M, T)
  (sib, LN) ← DELETE (M, T) {sib is the sibling of M}
  INVALIDATE_KEYS (sib)
  if (height(T) = depth(sib) + 2) or (first[sib] = last[sib]) then {Tree has become
  unbalanced}
    balancer ← GET_BALANCER (sib)
    INVALIDATE_KEYS (balancer)
    DELETE (balancer, T)
    p ← NEW_NODE ( )
    if LN = TRUE then {sib was left node before deletion 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

DELETE (M, T)
  RT ← root[T]
  if M = left[par[M]] then
    left_node ← TRUE
    sib ← right[par[M]]
    SHIFT_LEFT(sib)
  else
    left_node ← FALSE
    sib ← left[par[M]]
  end if
  par[sib] ← par[par[sib]]
  x ← sib
  while x ≠ RT do
    last[par[x]] ← last[par[x]] - 1
    if x = left[par[x]] then
      SHIFT_LEFT(right[par[x]])
    end if
    x ← par[x]
```

```
    end while
    return(sib, left_node)

SHIFT-LEFT(x)
    first[x] ← first[x] - 1
    last[x] ← last[x] - 1
    if first[x] ≠ last[x] then
        SHIFT_LEFT(1eft[x])
        SHIFT_LEFT(right[x])
    end if

GET_BALANCER (x)
    nd ← par[x]
    while |(1ast [left[nd]] - first[left[nd]]) - (last[right[nd]] - first[right[nd]])| < 2 do
        nd ← par[nd]
    end while
    while first[nd] ≠ last[nd] do
        if (last[left[nd]] - first[left[nd]]) > (last[right[nd]] - first[right[nd]]) then
            nd ← left[nd]
        else
            nd ← right[nd]
        end if
    end while
    return (nd)

INVALIDATE_KEY(x)
    repeat
        key[x] ← nil
        x ← par[x]
    until x = root[T]
```

```
KEY_CHANGE_ON_LEAVE ( )
  x ← par[M1]
  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
           RECVlast[l], i ({key[x]}key[l])
           key[x] ← {{key[x]}key[l]}key[l] - 1
        else if i = first[r] then
          RECVlast[l], i ({key[x]}leftket)
          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
           RECVlast[r], i ({key[x]}key[r])
           key[x] ← {{key[x]}key[r]}key[r] - 1
        end if
     end if
     x ← par[x]
  end while
```