

**Distributed and Collaborative Key Agreement
Protocols with Authentication and Implementation for
Dynamic Peer Groups**

Lee, Pak-Ching

A Thesis Submitted in Partial Fulfillment
of the Requirements for the Degree of
Master of Philosophy
in
Computer Science and Engineering

©The Chinese University of Hong Kong

June, 2003

The Chinese University of Hong Kong holds the copyright of this thesis. Any person(s) intending to use a part or the whole of the materials in this thesis in a proposed publication must seek copyright release from the Dean of the Graduate School.

Abstract

We consider several distributed collaborative key agreement protocols for dynamic peer groups. There are several important characteristics which make this problem different from traditional secure group communication. They are (1) distributed nature in which there is *no centralized key server*, (2) collaborative nature in which the group key is *contributory* (i.e., each group member will collaboratively contribute its part to the global group key), and (3) dynamic nature in which existing members may leave the group while new members may join. Instead of performing individual rekeying operations, i.e., recomputing the group key after every join or leave request, we discuss an interval-based approach of rekeying. We consider three *interval-based distributed rekeying algorithms*, or *interval-based algorithms* for short, for updating the group key: (1) the *Rebuild algorithm*, (2) the *Batch algorithm*, and (3) the *Queue-batch algorithm*. Performance of these three interval-based algorithms under different stochastic settings, such as different join and leave probabilities, is analyzed. We show that the interval-based algorithms significantly outperform the individual rekeying approach, and that the Queue-batch algorithm performs the best among the three interval-based algorithms. More important, the Queue-batch algorithm has the intrinsic property of substantially reducing the computation and communication workload in a highly dynamic environment. To further enhance our algorithms, we focus on their extensions in two aspects: authentication and implementation. We incorporated a member authentication mechanism into the algorithms and hence strengthened their security. We

also implemented the *Secure Group Communication Library (SGCL)* to realize the algorithms and to offer a programming interface to software developers for building their secure group-oriented applications. Our work provides a fundamental understanding about establishing a group key via a distributed and collaborative approach for a dynamic peer group.

摘要

在本論文中，我們考慮數個為動態對等式群組而設的分散及合作式金鑰協議算法。我們研究的重點跟以往為安全群組通信提出的有數個不同的特徵，包括：(1) 分散性，意指在金鑰協議算法中並沒有中央金鑰伺服器的參與；(2) 合作性，意指群組金鑰是由所有群組成員參與計算而產生的；以及 (3) 流動性，意指群組成員能夠隨時加入或離開群組而不影響整個金鑰的製造過程。我們並不考慮採用單獨式更新金鑰的方法，即是指金鑰會在有一位成員加入或離開群組時更新，取而代之的是間歇式更新金鑰的方式。我們會提出三個“間歇式分散金鑰更新算法”(interval-based distributed rekeying algorithms)，分別是：(1) **重建算法** (the Rebuild algorithm)，(2) **批次算法** (the Batch algorithm)，及 (3) **佇列批次算法** (the Queue-batch algorithm)。我們會根據不同的設定，例如是加入或離開群組的或然率，以對這三種算法進行性能評估。從評估中，我們發現這三個算法的性能比單獨式更新金鑰方法的優勝，而佇列批次算法的性能在這三個算法中最為突出；更重要的，佇列批次算法能夠在高流動性的環境下大幅降低計算及通訊成本。為了更能夠改進我們的研究，我們考慮兩方面的延展：認證及系統製作。我們把認證機制合併於金鑰更新的算法中以增加它們的安全性；同時，我們亦製作了“**安全群組通信函式庫**”(Secure Group Communication Library，或簡稱 SGCL) 以了解算法的特點及讓程式開發員撰寫安全群組應用程式。我們希望透過這次研究帶出一個基本概念，就是如何以分散及合作的方法為動態對等式群組建立群組金鑰。

Acknowledgment

I am glad to take this opportunity to cordially acknowledge a number of people who provide me with great support in these two years.

First, I would like to thank my advisor Professor John C.S. Lui for his guidance throughout the research. He taught me not only how to do research (e.g., how to define research problems, how to devise new solutions, how to conduct meaningful experiments and how to elaborate findings in words), but also how to *enjoy* research. From him, I realized how to appreciate the fun part of research, and this attitude played the most crucial role in prompting me to pursue my master research. I feel so lucky that he can be my advisor.

Besides, I would like to thank Professor David K.Y. Yau, from Purdue University, for his advice in my research. His comments let me make much important progress.

Furthermore, I would like to thank my friends for their help in solving my research questions, among them are: Alix Chow, Kwok-Tai Law, Sam Lee, Richard Sia, Starsky Wong, and Siu-Fung Yeung. I am so pleased that they can always offer me a hand whenever I have troubles.

Last but not least, I would like to thank my parents and elder sister for their generous support and encouragement throughout my life. Their kindness makes my life meaningful.

Contents

1	Introduction	1
2	Related Work	5
3	Tree-Based Group Diffie-Hellman	9
4	Interval-Based Distributed Rekeying Algorithms	14
4.1	Rebuild Algorithm	15
4.2	Batch Algorithm	16
4.3	Queue-batch Algorithm	19
5	Performance Evaluation	22
5.1	Mathematical Analysis	22
5.1.1	Analysis of the Rebuild Algorithm	24
5.1.2	Analysis of the Batch Algorithm	25
5.1.3	Analysis of the Queue-batch Algorithm	30
5.2	Experiments	31
5.3	Discussion of the experimental results	35
6	Authenticated Tree-Based Group Diffie-Hellman	43
6.1	Description of A-TGDH	44
6.2	Security Analysis	47
7	Implementation and Applications	50

7.1	Leader and Sponsors	51
7.1.1	Leader	51
7.1.2	Sponsors	53
7.1.3	Rekeying Operation	56
7.2	System Architecture	57
7.2.1	System Preliminaries	57
7.2.2	System Components	58
7.2.3	Implementation Considerations	64
7.3	SGCL API	65
7.4	Experiments	67
7.5	Applications	72
7.6	Future Extensions	75
8	Conclusions and Future Directions	76
8.1	Conclusions	76
8.2	Future Directions	77
8.2.1	Construction of a Hybrid Key Tree with the Physical and Logical Properties	77
8.2.2	Extended Implementation	79
	Bibliography	80

List of Figures

3.1	A possible key tree used in the Tree-Based Group Diffie-Hellman protocol.	10
3.2	Illustration of the rekeying operation after a single leave.	11
3.3	Illustration of the rekeying operation after a single join.	12
4.1	Pseudo-code of the Rebuild algorithm.	16
4.2	Example of the Rebuild algorithm.	16
4.3	Pseudo-code of the Batch algorithm.	17
4.4	Example 1 of the Batch algorithm where $L > J > 0$	18
4.5	Example 2 of the Batch algorithm where $J > L > 0$	19
4.6	Pseudo-code of the Queue-subtree phase.	20
4.7	Pseudo-code of the Queue-merge phase.	20
4.8	Example of the Queue-merge phase.	21
5.1	Performance differences between individual rekeying and Rebuild.	36
5.2	Performance differences between individual rekeying and Batch.	36
5.3	Performance differences between individual rekeying and Queue-batch.	36
5.4	Performance results of Rebuild, Batch, and Queue-batch at different numbers of joins based on mathematical models.	37
5.5	Average performance results of Rebuild, Batch, and Queue-batch at different fixed join probabilities.	38

5.6	Instantaneous numbers of exponentiations of Batch and Queue-batch at different join and leave probabilities.	39
5.7	Instantaneous numbers of renewed nodes of Batch and Queue-batch at different join and leave probabilities.	40
5.8	Average performance results of Queue-batch at different reset intervals.	41
5.9	Average and instantaneous numbers of rounds of Rebuild, Batch, and Queue-batch at different join and leave probabilities.	42
6.1	Example of authenticated key agreement involving 4 members.	46
7.1	Pseudo-code of the sponsors coordination algorithm.	55
7.2	Example to illustrate the sponsor coordination algorithm in Fig. 7.1.	55
7.3	Illustration of the broadcast-efficient property of the sponsor coordination algorithm.	56
7.4	Formats of the regular packets.	60
7.5	Overview of general operations on received packets.	61
7.6	Overview of leader-specific components and their relationships with other components.	63
7.7	Flowchart of using the SGCL API.	67
7.8	Average analysis at different fixed T_{out} 's.	70
7.9	Average analysis at different levels of membership dynamics.	71
7.10	Illustration of Chatter in the graphical mode.	73
7.11	Illustration of Chatter in the text mode.	74
8.1	Approaches of updating the key tree.	78

List of Tables

7.1	Description of components used in SGCL.	59
7.2	Description of the SGCL API functions.	66

List of Publications

Part of this research work appeared in the following publications:

- Patrick P. C. Lee, John C. S. Lui, and David K. Y. Yau. Distributed Collaborative Key Agreement Protocols for Dynamic Peer Groups. In *Proceedings of the 10th IEEE International Conference on Network Protocols (ICNP)*, France, November, 2002.
- Patrick Pak Ching Lee, John C. S. Lui, David K. Y. Yau, Distributed Collaborative Key Agreement and Authentication Protocols for Dynamic Peer Groups. Technical Report CS-TR-2002-08, Dept of Computer Science and Engineering, The Chinese University of Hong Kong, August 2002.
- Patrick Pak Ching Lee, John C. S. Lui, David K. Y. Yau, Distributed Collaborative Key Agreement Protocols for Dynamic Peer Groups. Technical Report CS-TR-2002-04, Dept of Computer Science and Engineering, The Chinese University of Hong Kong, May 2002.

Chapter 1

Introduction

With the emergence of many group-oriented distributed applications such as tele/video-conferencing and multi-player games, there is a need for security services to provide group-oriented communication privacy and data integrity. To provide this form of group communication privacy, it is paramount that members of the group can establish a common secret key for encrypting group communication data. To illustrate the utility of this type of application, consider a group of people in a peer-to-peer or ad-hoc network having a closed and confidential meeting. Since they do not have a previously agreed upon common secret key, communication between group members is susceptible to eavesdropping. To solve the problem, we need a *secure distributed group key agreement protocol* so that people can establish and a common group key for secure and private communication. Note that this type of key agreement protocols is both distributed and contributory in nature: each member of the group contributes its part to the overall group key.

It is important to point out that the type of distributed group key agreement protocols we study is very different from more traditional centralized group key agreement protocols. Centralized protocols rely on a *centralized key server* to efficiently distribute the group key. An excellent body of work on centralized key distribution protocols exists in [14, 20, 26, 27]. In those approaches, group members are arranged in a logical key hierarchy known as

a *key tree*. Using the tree topology, it is easy to distribute the group key to members whenever there is any change in the group membership (e.g., a new member joins or an existing member leaves the group). For distributed key agreement protocols we consider, however, there is no centralized key server available. This arrangement is justified in many situations – e.g., in a peer-to-peer or an ad-hoc network where centralized resources are not readily available. 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.

For the special case of a communication group having only two members, these members can create a group key using the Diffie-Hellman key exchange protocol [5]. In the protocol, members X and Y use a cyclic group \mathcal{G} of order p and a generator α . They can generate their secret components e_X and e_Y , respectively. Member X (resp., Y) can compute its public key $\alpha^{e_X} \bmod p$ (resp., $\alpha^{e_Y} \bmod p$) and send it to Y (resp., X). Since both members know their own exponent, they can each raise the other party's public key to the exponent and produce a common group key, which is equal to $\alpha^{e_X e_Y} \bmod p$. Using this common group key, X and Y can encrypt their data to prevent eavesdropping by intruders.

In this dissertation, we consider a dynamic communication group in which members are located in a distributed fashion. We extend the Diffie-Hellman key exchange protocol to more than two members in the communication group. The membership of the communication group is dynamic so that members can leave and new members can join the group at any time. The contributions of our work are:

- The key agreement protocol is distributed and there is no centralized key server.
- The key agreement protocol is contributory – each member contributes

its part to the overall group key.

- Instead of performing individual rekeying operations, we propose to use an *interval-based* approach to significantly reduce the computation and communication costs of maintaining the group key. This interval-based approach preserves rekeying efficiency in dynamic peer groups.
- We propose three *interval-based distributed rekeying algorithms*, or *interval-based algorithms* for short, and conduct performance evaluation, including both analytical and simulation-based analysis, to illustrate their performance merits.
- We propose an authenticated group key agreement protocol and prove its security strengths.
- We implemented the *Secure Group Communication Library (SGCL)* to realize the interval-based algorithms. The library provides a set of API functions tailored for the development of secure group-oriented applications.

The balance of the dissertation is organized as follows. In Chapter 2, we first discuss related work about centralized group key distribution, distributed group key agreement, authenticated group key agreement, as well as the implementation experience concerning group key management. In Chapter 3, we provide the background of the Diffie-Hellman protocol. We then explain how it can be extended to the Tree-Based Group Diffie-Hellman protocol, the group key agreement protocol that accommodates more than two members in a dynamic peer group. In Chapter 4, we present three interval-based algorithms to reduce the computation and communication costs for maintaining the group key in a dynamic peer group. In Chapter 5, we conduct mathematical analysis to quantify the system performance according to the given performance metrics when the original Diffie-Hellman tree is completely balanced. We also report

several experiments that illustrate the system costs under dynamic joins and leaves using various system parameters (e.g., join and leave probabilities). In Chapter 6, we describe our proposed authenticated group key agreement protocol, known as the *Authenticated Tree-Based Group Diffie-Hellman* (A-TGDH) protocol, and provide arguments on whether it satisfies our security goals. In Chapter 7, we study the implementation details of SGCL and present experiments that evaluate the performance of the interval-based algorithms under real network settings. Finally, in Chapter 8, we conclude the dissertation and propose future directions that enrich the research.

Chapter 2

Related Work

In this chapter, we consider a number of group key management schemes previously developed to protect group communication. These schemes can be classified into two categories: *centralized group key distribution* and *decentralized group key agreement*. In centralized group key distribution, a centralized key server is set up to generate and distribute group keys to all group members. In decentralized group key agreement, however, all group members are involved in generating the group key and finally agree upon a common group key. To verify the identities of group members that participate in the key generation process, several decentralized group key agreement schemes are further extended to incorporate authentication and they are classified as *authenticated group key agreement* schemes. In the following, we review the research work about group key management in three areas: (1) centralized group key distribution, (2) decentralized group key agreement, and (3) authenticated key agreement, as well as the implementation experience regarding group key management.

Centralized group key distribution, as mentioned above, requires a single centralized key server to generate and distribute keys to group members. Intuitively, the key server can set up a secure unicast channel with each group member and distribute newly generated keys through these channels. This method, however, is not scalable when the member pool is very large. To

address this scalability issue, Wong *et al.* [27] and Wallner *et al.* [26] independently proposed the *key tree* approach to achieve secure group communication. They suggested to associate keys in a hierarchical tree and perform rekeying at every join or leave event. Later, the authors in [14, 20, 29] introduced *batch rekeying*, meaning that the group key is renewed at regular intervals. Therefore, the key renewal procedure is independent of membership dynamics and thus becomes more efficient. In this dissertation, we apply the key tree approach and the batch rekeying concept to our proposed algorithms.

Decentralized group key agreement requires the participation of all group members and therefore avoids the single-point-of-failure problem found in centralized key distribution. Its research is explored in [4, 24, 11, 12], in which the authors extended the Diffie-Hellman protocol [5] to support secure group communication in a peer-to-peer network. Burmester and Desmedt [4] proposed a computation-efficient protocol at the expense of high communication overhead. Steiner *et al.* [24] developed *Cliques*, in which every member introduces its key component into the result generated by its preceding member and passes the new result to its following member. Cliques is efficient in rekeying for leave or partition events, but imposes a high workload on the last member in the chain. Kim *et al.* [11] proposed the Tree-Based Group Diffie-Hellman (TGDH) to arrange keys in a tree structure. Every member only needs to hold the keys along its key path, implying that the rekeying workload is distributed to all members. The authors also suggested a variant of TGDH called STR which minimizes the communication overhead by trading off the computational complexity [12]. All the above schemes are *contributory*, meaning that all group members contribute their own private piece of information to generate the group key. While the key renewal in [4] is independent of membership change, the rest of the schemes [24, 11, 12] suggest to perform rekeying at single join, leave, merge or partition events. One of our research goals is to enhance TGDH to support rekeying involving a batch of join and leave events.

Rather than emphasizing the rekeying efficiency, authenticated group key agreement focuses on how to efficiently incorporate the certified key components of group members into a group key and hence attain a high degree of security. The authors in [10, 2, 17] developed authenticated group key agreement schemes based on the Burmester-Desmedt model, Cliques, and TGDH respectively. The one proposed in [17], called AGKA-G, is an extension of the two-party Günther scheme [8] to the TGDH protocol. However, the AGKA-G protocol has several drawbacks. First, the Günther scheme, and hence AGKA-G, does not provide perfect forward secrecy. Besides, AGKA-G is not role-symmetric since sponsors perform more operations in the key generation and distribution. Furthermore, it is not completely contributory as sponsors provide more contribution than non-sponsors in the resulting group key. As described in Chapter 6, we propose an authenticated group key agreement protocol that resolves these problems and meanwhile achieves desired security properties.

Up to now, there have not been many implementation projects that attempt to put group key management schemes in practice. The most famous one is called *Secure Spread* [25, 1], which implemented the centralized group key distribution protocol and a number of distributed group key agreement protocols including the Burmester-Desmedt model, Cliques, TGDH, and STR. The project reflects the features of the group key management schemes under join, leave, merge, and partition events, and provides a set of function calls suitable for secure application development. In our research, we implemented a programming library based on the interval-based approach and built applications with the library to demonstrate its strengths and effectiveness.

To summarize, group key management is divided into two categories: centralized group key distribution and decentralized group key agreement, while the latter is further extended to authenticated group key agreement. Through the investigation of batch rekeying under a key tree and the Tree-Based Group

Diffie-Hellman protocol, we make several contributions: proposing an interval-based approach to perform rekeying, designing an authentication mechanism to secure the algorithms, and implementing a programming library to realize the algorithms.

Chapter 3

Tree-Based Group

Diffie-Hellman

In this chapter, we introduce the working principle of the Tree-Based Group Diffie-Hellman (TGDH) protocol [11]¹, which substantiates our proposed protocols discussed in later chapters. In the following explanation, we also bring out several terminologies that will be used throughout this dissertation.

In TGDH, each member maintains a set of keys, which are arranged in a hierarchical *binary tree*. We assign a node ID v to every tree node. For a given node v , we associate a *secret* (or private) key K_v and a *blinded* (or public) key BK_v . All arithmetic operations are performed in a cyclic group of prime order p with the generator α . Therefore, the blinded key of node v can be generated by

$$BK_v = \alpha^{K_v \bmod p}. \quad (3.1)$$

Each leaf node in the tree represents the individual secret and blinded keys of a group member, denoted by M_i . Every member holds all the secret keys along its *key path* starting from its associated leaf node up to the root node. Therefore, the secret key held by the root node is shared by all the members and is regarded as the *group key*. Fig. 3.1 illustrates a possible key tree with

¹The journal version of this paper appeared in [13].

six members M_1 to M_6 , e.g., member M_1 holds the keys at nodes 7, 3, 1, and 0. The secret key at node 0 is the group key of this peer group.

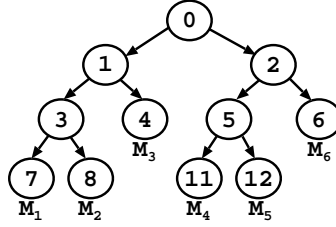


Figure 3.1: A possible key tree used in the Tree-Based Group Diffie-Hellman protocol.

The node ID of the root node is set to 0. Each *non-leaf* node v consists of two child nodes, whose node IDs are given by $2v + 1$ and $2v + 2$. Based on the Diffie-Hellman protocol [5], the secret key of a non-leaf node v can be generated by the secret key of one child node of v , and the blinded key of another child node of v . Mathematically, we have

$$\begin{aligned}
 K_v &= (BK_{2v+1})^{K_{2v+2}} \bmod p \\
 &= (BK_{2v+2})^{K_{2v+1}} \bmod p \\
 &= \alpha^{K_{2v+1}K_{2v+2}} \bmod p.
 \end{aligned} \tag{3.2}$$

Unlike the keys at non-leaf nodes, the secret key at a leaf node is selected by its corresponding group member. The key selection can be achieved through a secure pseudo random number generator [23].

Since the blinded keys are publicly known, every member can compute the keys along its key path to the root node based on its individual secret key. To illustrate, consider the group in Fig. 3.1. Every member M_i generates its own secret key and all the secret keys along the path to the root node. For example, member M_1 generates the secret key K_7 and it can request the blinded key BK_8 from M_2 , BK_4 from M_3 , and BK_2 from either M_4, M_5 or M_6 . Given M_1 's secret key K_7 and the blinded key BK_8 , M_1 can generate the secret key K_3 according to Eq. 3.2. Given the blinded key BK_4 and the newly generated

secret key K_3 , M_1 can generate the secret key K_1 based on Eq. 3.2. Given the secret key K_1 and the blinded key BK_2 , M_1 can generate the secret key K_0 at the root. From that point on, any communication in the group can be encrypted based on the secret key (or group key) K_0 .

To provide both backward confidentiality (i.e., joined members cannot access previous communication data) and forward confidentiality (i.e., left members cannot access future communication data), *rekeying*, which means renewing the keys associated with the nodes of the key tree, is performed whenever there is any group membership change, including any new member joining or any existing member leaving the group. Let us first consider individual rekeying, meaning that rekeying is conducted after every single join or leave event. Before the group membership is changed, a special member called the *sponsor* is elected, and the sponsor is responsible for updating the keys held by the new member (in the join case) or departed member (in the leave case). We use the convention that the rightmost member under the subtree rooted at the sibling of the join and leave nodes will take the sponsor role. Note that the existence of a sponsor does not violate the decentralized requirement of the group key generation since the sponsor does not add extra contribution to the group key.

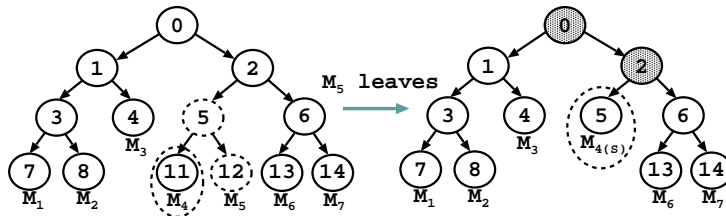


Figure 3.2: Illustration of the rekeying operation after a single leave.

Fig. 3.2 illustrates a member leave event. Suppose that member M_5 leaves the system. Node 11 is then *promoted* to node 5, and nodes 2 and 0 become *renewed nodes*, which are defined as the non-leaf nodes whose associated keys in the key tree are renewed. Also, member M_4 becomes the sponsor. It needs

to renew the secret keys K_2 and K_0 , and broadcasts the blinded keys BK_2 and BK_5 to all the members. Members M_1 , M_2 , and M_3 , upon receiving the blinded key BK_2 , can compute the new group key K_0 . Similarly, members M_6 and M_7 , upon receiving BK_5 , can compute K_2 and then the new group key K_0 .

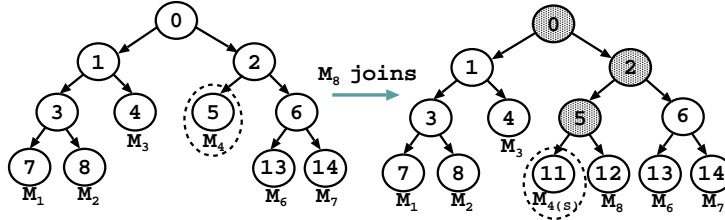


Figure 3.3: Illustration of the rekeying operation after a single join.

Fig. 3.3 illustrates a new member M_8 that wishes to join the group. M_8 has to first determine the *insertion node* under which M_8 can be inserted. To *add* a node, say v' (or tree, say T') to the insertion node, a new node, say n' , is first created. Then the subtree rooted at the insertion node becomes the left child of the node n' , and the node v' (or the root node of the tree T') becomes the right child of the node n' . The node n' will replace the original location of the insertion node. The insertion node is either the rightmost shallowest position such that the join does not increase the tree height, or the root node if the tree is initially well balanced (in this case, the height of the resulting tree will be increased by 1). Fig. 3.3 illustrates this concept. The insertion node is node 5 and the sponsor is M_4 . M_8 then broadcasts its blinded key BK_{12} upon insertion. Given BK_{12} , M_4 renews K_5 , K_2 , and K_0 , and then broadcasts the blinded keys BK_5 and BK_2 to all members in the group. After receiving the blinded keys from M_4 , all remaining members can rekey all the nodes along their key paths and obtain the new group key K_0 .

Based on the above leave and join events in Fig. 3.2 and 3.3, we find that we can *reduce* one rekeying operation if we can simply change the association of node 12 from M_5 to M_8 . *Interval-based rekeying* is thus proposed such that

rekeying is performed on a batch of join and leave requests so as to reduce the number of rekeying operations. Members carry out rekeying operations at regular *rekeying intervals*. In the following chapter, we describe the interval-based approach to manage rekeying operations.

Chapter 4

Interval-Based Distributed Rekeying Algorithms

In this chapter, we present three interval-based distributed rekeying algorithms, or interval-based algorithms for short. They are the *Rebuild algorithm*, the *Batch algorithm*, and the *Queue-batch algorithm*. The aim of interval-based rekeying is to maintain good rekeying performance which is independent of the dynamics of joins and leaves. The three interval-based algorithms are developed based on the following assumptions:

- The key tree of TGDH is used as a foundation of all the algorithms.
- Rekeying operations are carried out at the beginning of every rekeying interval. There exists a *virtual queue* holding all join and leave requests until the beginning of the next rekeying interval.
- When a new member sends a join request, it also includes its individual blinded key.
- For simplicity, all members know the existing key tree structure and they also know all the blinded keys within the tree.
- To obtain the blinded keys of the renewed nodes, the key paths of the sponsors should contain those renewed nodes. Since the interval-based

rekeying operations involve nodes lying on more than one key paths, more than one sponsors may be elected. Also, a renewed node may be rekeyed by more than one sponsor. Therefore, we assume that the sponsors can coordinate with one another such that the blinded keys of all the renewed nodes are broadcast only once.

In the next three sections, we present the interval-based algorithms. We adopt the following notations in our description. Let T denote the existing key tree. Assume that $L \geq 0$ existing members $\mathbf{M}^l = \langle M_1^l, \dots, M_L^l \rangle$ wish to leave and $J \geq 0$ new members $\mathbf{M}^j = \langle M_1^j, \dots, M_J^j \rangle$ wish to join the group within a rekeying interval.

4.1 Rebuild Algorithm

The motivation for the Rebuild algorithm is to *minimize* the resulting tree height so that the rekeying operations for each group member can be reduced. At the beginning of every rekeying interval, we reconstruct the whole key tree with all existing members that remain in the communication group, together with the newly joining members. The resulting tree is a *left-complete* tree, where its leaf nodes have depths differed by at most one and those deeper leaf nodes are located at the leftmost positions. The pseudo-code of the Rebuild algorithm to be performed by every member is shown in Fig. 4.1.

Fig. 4.2 shows the scenario where members M_2 , M_5 , and M_7 wish to leave and a new member M_8 wishes to join the communication group. Based on the algorithm, the resulting key tree consists of five members and has all non-leaf nodes renewed. Besides, the sponsors include all the five members.

Rebuild (T, M^j, J, M^l, L)

1. obtain all members from T and store them in M' ;
 2. remove the L leaving members in M^l from M' ;
 3. add the J new members in M^j to M' ;
 4. create a new binary tree T' based on members in M' and set $T = T'$;
 5. elect all members to be sponsors;
 6. rekey the key nodes and broadcast the new blinded keys in T ;
-

Figure 4.1: Pseudo-code of the Rebuild algorithm.

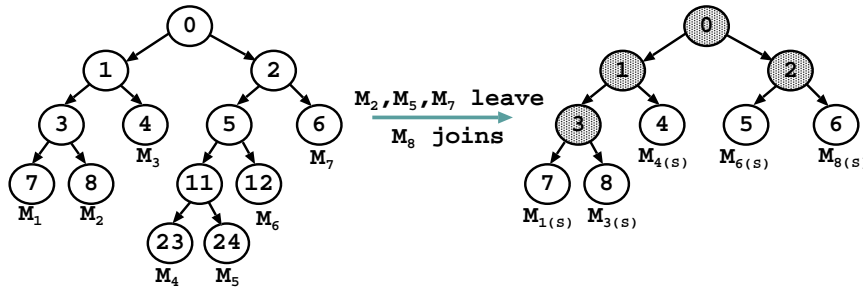


Figure 4.2: Example of the Rebuild algorithm.

4.2 Batch Algorithm

The Batch algorithm is based on the centralized approach in [14], which is now applied to a distributed system without a centralized key server and all members contribute to the composition of the group key. The pseudo-code of the Batch algorithm is given in Fig. 4.3. Notice that the sponsors may have to wait for the blinded keys on another key path in order to proceed upwards to rekey the nodes. Finally, all the members obtain the necessary blinded keys to compute the new group key K_0 .

The Batch algorithm is illustrated with two examples. In Fig. 4.4, we show the case where $L > J > 0$. Suppose M_2 , M_5 , and M_7 leave and a new member M_8 wishes to join. The following steps are carried out: (i) M_8 broadcasts its join request, including its individual blinded key. (ii) The leaf node 6

```

Batch ( $T, M^j, J, M^l, L$ )
1.  if ( $L == 0$ ) { /* pure join case */
2.      create a new tree  $T'$  based on new members in  $M^j$ ;
3.      either (a) add  $T'$  to the shallowest node of  $T$  (which need not be the leaf
         node) such that the merge will not increase the height of the result tree, or
         (b) add  $T'$  to the root node of  $T$  if the merge to any node of  $T$  will increase
         the tree height;
4.  } else { /* some existing members want to leave */
5.      sort  $M^l$  in an ascending order of the associated node IDs of the members
         and store the results in  $M^{l,s} = \langle M_1^{l,s}, \dots, M_L^{l,s} \rangle$ ;
6.      if ( $L \geq J$ ) { /* more members want to leave than join */
7.          if ( $J > 0$ )
8.              replace the departed nodes of  $\langle M_1^{l,s}, \dots, M_J^{l,s} \rangle$  with  $J$  joined nodes;
9.              if ( $L - J > 0$ ) {
10.                 remove remaining  $L - J$  leaving leaf nodes from the parent node;
11.                 promote the siblings of the leaving leaf nodes;
12.             }
13.         } else { /* more newly joining members than leaving members */
14.             divide  $M^j$  into  $L$  subgroups  $\mathbf{G} = \langle G_1, \dots, G_L \rangle$  such that the first  $J \bmod L$ 
                 subgroups  $\langle G_1, \dots, G_{J \bmod L} \rangle$  contain  $\lfloor \frac{J}{L} \rfloor + 1$  new members and the
                 rest contain  $\lfloor \frac{J}{L} \rfloor$  new members;
15.             create  $L$  subtrees  $\langle T'_1, \dots, T'_L \rangle$  for the subgroups  $\mathbf{G}$ ;
16.             replace the departed nodes of  $\langle M_1^{l,s}, \dots, M_{J \bmod L}^{l,s} \rangle$  with the roots of
                  $\langle T'_1, \dots, T'_{J \bmod L} \rangle$  and the remaining departed nodes with the roots of
                 remaining subtrees;
17.         }
18.     }
19.     elect the members to be sponsors if (1) they are new members, or (2) the right-
         most members of the subtrees rooted at the siblings of the departed nodes or
         replaced nodes in  $T$ ;
20.     if (sponsor) /* responsibility of the sponsor */
21.         rekey the key nodes and broadcast the new blinded keys;

```

Figure 4.3: Pseudo-code of the Batch algorithm.

associated with M_7 is replaced by the node of M_8 , and the leaf nodes 8 and 24 are removed. Nodes 7 and 23 are promoted to nodes 3 and 11, respectively.

(iii) M_1 , M_4 , M_6 , and M_8 are elected to be the sponsors. M_1 renews secret keys K_1 and K_0 , and M_4 renews K_5 , K_2 , and K_0 . M_1 then broadcasts BK_1 , and M_4 broadcasts BK_5 and BK_2 . M_6 and M_8 , though having the sponsor role, do not need to broadcast any blinded keys as M_4 has already broadcast this information. (iv) Finally, every member can compute the new group key based on the received blinded keys.

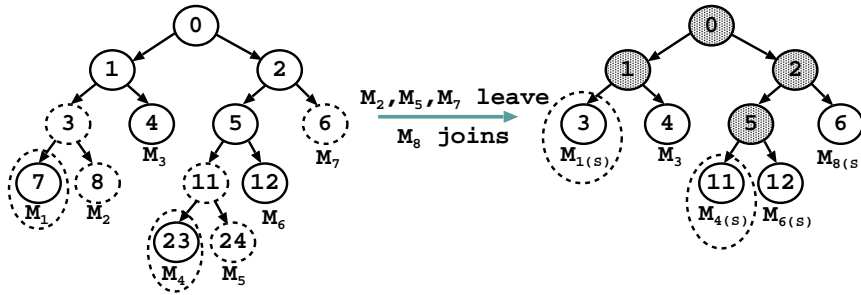
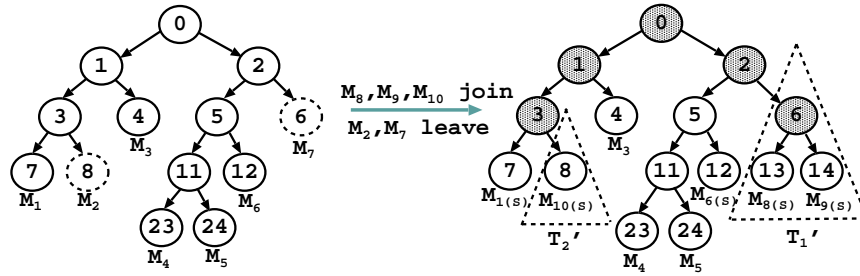


Figure 4.4: Example 1 of the Batch algorithm where $L > J > 0$.

Fig. 4.5 illustrates the case where $J > L > 0$. Suppose M_8 , M_9 , and M_{10} join, and M_2 and M_7 leave. The rekeying process is: (i) M_8 , M_9 , and M_{10} broadcast their join requests together with their own individual blinded key. (ii) M_8 and M_9 form the subtree T_1' and M_{10} is the only member of T_2' . The root of T_1' replaces node 6 and the root of T_2' replaces node 8. (iii) The sponsors are M_1 , M_6 , M_8 , M_9 , and M_{10} . M_8 and M_9 first need to compute the secret key K_6 , and either one of them computes and broadcasts the new blinded key BK_6 . (iv) M_1 (or M_{10}) renews K_3 and K_1 and broadcasts BK_3 and BK_1 . M_6 renews K_2 and broadcasts BK_2 . (v) Finally, all the members can compute the new group key K_0 .

Figure 4.5: Example 2 of the Batch algorithm where $J > L > 0$.

4.3 Queue-batch Algorithm

We find that the previous approaches perform all rekeying steps at the beginning of every rekeying interval. This results in a high processing load during the update instance and thereby delays the start of the secure group communication. Thus, we propose a more effective algorithm which we call the *Queue-batch* algorithm. Its intuition is to reduce the rekeying load by pre-processing the joining members in the virtual queue during the idle rekeying interval.

The Queue-batch algorithm is divided into two phases, namely the *Queue-subtree* phase and the *Queue-merge* phase. The first phase occurs whenever a new member joins the communication group during the rekeying interval. In this case, we append this new member in a temporary key tree T' . The second phase occurs at the beginning of every rekeying interval and we merge the temporary tree T' (which contains all newly joining members) to the existing key tree T . The pseudo-codes of the Queue-subtree phase and the Queue-merge phase are illustrated in Figs. 4.6 and 4.7.

The Queue-batch algorithm is illustrated in Fig. 4.8, where members M_8 , M_9 , and M_{10} wish to join the communication group, while M_2 and M_7 wish to leave. Then the rekeying process is as follows: (i) In the *Queue-subtree* phase, the three new members M_8 , M_9 , and M_{10} first form a tree T' . M_{10} , in this case, is elected to be the sponsor. (ii) In the *Queue-merge* phase, the tree T' is added at the highest departed position, which is at node 6. Also, the

Queue-subtree (T')

1. **if** (a new member joins) {
 2. **if** ($T' == \text{NULL}$) /* no new members in T' */
 3. create a new tree T' with the only one new member;
 4. **else** { /* there are new members in T' */
 5. find the insertion node;
 6. add the new member to T' ;
 7. elect the rightmost member under the subtree rooted at the sibling of the joining node to be the sponsor;
 8. **if** (sponsor) /* responsibility of the sponsor */
 9. rekey the key nodes and broadcast the new blinded keys to the group;
 10. }
 11. }
-

Figure 4.6: Pseudo-code of the Queue-subtree phase.

Queue-merge (T, T', M^l, L)

1. **if** ($L == 0$) { /* there is no leave */
 2. add T' to either (a) the shallowest node (which need not be the leaf node) of T such that the merge will not increase the resulting tree height, or (b) the root node of T if the merge to any locations will increase the resulting tree height;
 3. } **else** { /* there are leaves */
 4. add T' to the highest leave position of the key tree T ;
 5. }
 6. elect members to be sponsors if they are (a) the rightmost members of the subtree rooted at the sibling nodes of the departed leaf nodes in T , or (b) they are the rightmost member of T' ;
 7. **if** (sponsor) /* responsibility of the sponsor */
 8. rekey the key nodes and broadcast the new blinded keys to the group;
-

Figure 4.7: Pseudo-code of the Queue-merge phase.

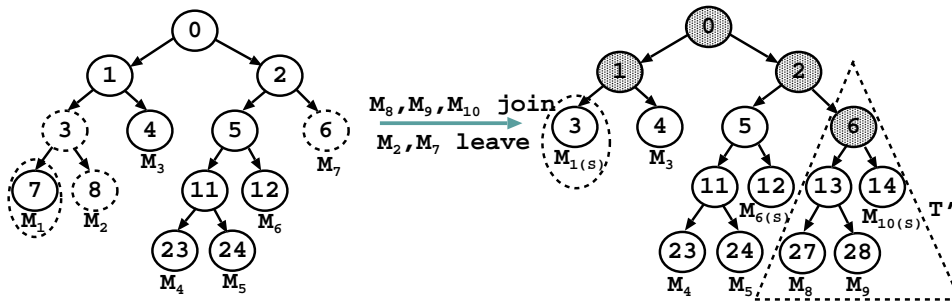


Figure 4.8: Example of the Queue-merge phase.

blinded key of the root node of T' , which is BK_6 , is broadcast by M_{10} . (iii) The sponsors M_1 , M_6 , and M_{10} are elected. M_1 renews the secret key K_1 and broadcasts the blinded key BK_1 , M_6 renews the secret key K_2 and broadcasts the blinded key BK_2 . (iv) Finally, all members can compute the group key.

Chapter 5

Performance Evaluation

This chapter covers the performance evaluation of the interval-based algorithms, consisting of Rebuild, Batch, and Queue-batch, in two aspects: mathematical analysis and simulations. It is important to point out that we only measure the rekeying performance at the update instance occurring at the beginning of each rekeying interval. Hence, for Queue-batch, we only consider the Queue-merge phase but not the Queue-subtree phase. The pre-processing steps in the latter do not influence the underlying communication which is protected with the current group key except that it introduces slight overhead of extra key exchange traffic. The measurement reflects the latency of generating the latest group key for data encryption in order to provide backward and forward confidentiality.

In the following text, we describe a number of mathematical models and simulation-based experiments. We also study the performance of the algorithms in terms of their computation and communication costs. At the end of this chapter, we discuss the implication brought by our evaluation findings.

5.1 Mathematical Analysis

In this section, we present the mathematical analysis of the three proposed algorithms. We consider two performance measures, namely:

1. *Number of exponentiation operations:* This metric gives a measure of the computation load of all members in the communication group.
2. *Number of renewed nodes:* A node is said to be *renewed* if it is a non-leaf node and its associated keys are renewed. This metric provides a measure on the communication cost since new blinded keys of the renewed nodes have to be broadcast to the whole group.

For simplicity, we assume the following in the analysis:

- The existing key tree is completely balanced prior to the interval-based rekeying event.
- Each member has a homogeneous leave probability.
- The computation of the blinded group key of the root node is counted in the blinded key computations. With this assumption, the number of blinded key computations simply equals the number of renewed nodes, provided that the blinded key of each renewed node is broadcast only once.

For the mathematical analysis, let N be the number of members originally in the system, L (where $0 \leq L \leq N$) be the number of members that wish to leave the system, and $J \geq 0$ be the number of new members that want to join the communication group. Let T denote the existing tree which contains N members. The level of a node v is $l = \lfloor \log_2(v + 1) \rfloor$, where v is the node ID, and the maximum level of T is h . Based on the first assumption, i.e., the key tree is initially balanced, we know that $N = 2^h$. Also, let \mathcal{R}_{alg} be the number of renewed nodes and \mathcal{E}_{alg} be the number of exponentiations for the particular algorithm alg . The performance measure \mathcal{E}_{alg} is composed of two parts: \mathcal{E}_{alg}^s and \mathcal{E}_{alg}^b , which respectively represent the number of exponentiations of calculating the secret keys (which is done by all members) and that of calculating

the blinded keys (which is done by sponsors only). We have

$$\mathcal{E}_{alg} = \mathcal{E}_{alg}^s + \mathcal{E}_{alg}^b. \quad (5.1)$$

Also, we know the number of blinded key computations is

$$\mathcal{E}_{alg}^b = \mathcal{R}_{alg}. \quad (5.2)$$

which is simply the mathematical interpretation of the last assumption.

In the following analysis, we only focus on the number of secret key computations \mathcal{E}_{alg}^s .

5.1.1 Analysis of the Rebuild Algorithm

Given N , L , and J , we can obtain the *exact* expressions for the two performance measures $\mathcal{R}_{Rebuild}$ and $\mathcal{E}_{Rebuild}$. It is important to note that the derived expressions below are valid even if the existing key tree T is not completely balanced originally.

The resulting number of members is $N^* = N - L + J \geq 0$. Thus, the number of renewed nodes (i.e. the number of non-leaf nodes) is

$$\mathcal{R}_{Rebuild}(N^*) = \begin{cases} 0 & \text{if } N^* = 0, \\ N^* - 1 & \text{otherwise.} \end{cases} \quad (5.3)$$

For $\mathcal{E}_{Rebuild}(N^*)$, we find that when $N^* \leq 1$, $\mathcal{E}_{Rebuild}(N^*) = 0$. If $N^* \in (2^{h'-1}, 2^{h'}]$ for $h' \geq 1$ where $h' = \lfloor \log_2(N^* - 1) \rfloor + 1$, we have

$$\begin{aligned} \mathcal{E}_{Rebuild}^s(N^*) &= (\text{number of members at level } h') \times h' + \\ &\quad (\text{number of members at level } h' - 1) \times (h' - 1) \\ &= 2(N^* - 2^{\lfloor \log_2(N^* - 1) \rfloor}) (\lfloor \log_2(N^* - 1) \rfloor + 1) + \\ &\quad (N^* - 2(N^* - 2^{\lfloor \log_2(N^* - 1) \rfloor})) \lfloor \log_2(N^* - 1) \rfloor \\ &= N^* \lfloor \log_2(N^* - 1) \rfloor + 2N^* - 2^{(\lfloor \log_2(N^* - 1) \rfloor + 1)}. \end{aligned} \quad (5.4)$$

5.1.2 Analysis of the Batch Algorithm

In analyzing the performance of the Batch algorithm, we consider the following five cases. Note that when $L > 0$, the performance metrics will depend on the membership leave positions and exact metrics cannot be obtained. Therefore, whenever $L > 0$ (e.g., cases 2 to 5 below), we derive the *expected* performance measures. We also define $\mathcal{R}_{alg,c}$ and $\mathcal{E}_{alg,c}$ be the two performance measures under condition c . We adopt the convention that the combination $\binom{n}{r}$ equals 0 if $n < 0$, $r < 0$ or $n < r$. (The following analysis is the extension of the centralized case in [14] to the distributed case.)

Case 1: $J > L = 0$ (pure join). Since the original key tree T is completely balanced before the rekeying operations, the subtree T' of the newly joined members will be inserted at the root of the existing tree T . Thus, the number of renewed nodes is

$$\mathcal{R}_{Batch, J>L=0} = \mathcal{R}_{Rebuild}(J) + 1 = (J - 1) + 1 = J. \quad (5.5)$$

The first term corresponds to the number of renewed nodes for all new members and the last term is to account for the node renewal cost to the root node in the resulting tree T .

The number of secret key exponentiations for the Batch algorithm is

$$\mathcal{E}_{Batch, J>L=0}^s = \mathcal{E}_{Rebuild}^s(J) + (N + J). \quad (5.6)$$

The first term corresponds to the exponentiation cost of creating a tree for the J new members. The term $(N + J)$ is the secret key computations of the new root node in the resulting tree performed by the $N + J$ members.

Case 2: $L > J = 0$ (pure leave). Consider a node v at level l . In a completely balanced tree, the node v has $N/2^l$ descendants. When $L > 0$, the node v can be in one of the three different states at the rekeying instances: *no-change*, *pruned*, and *renewed*. The node v can be in the “no-change” state

if none of its $N/2^l$ descendants wish to leave. The probability of being in the no-change state is

$$P[\text{node } v \text{ is no-change}] = \frac{\binom{N-N/2^l}{L}}{\binom{N}{L}}. \quad (5.7)$$

The node v is pruned if (1) all descendants of the node v leave, and (2) all descendants of either its left or right subtree leave. In the latter case, v is pruned due to node promotion (please refer to step 11 of the pseudo code of the Batch algorithm). The number of non-leaf nodes that are pruned due to the node promotion is L (or $L - 1$ if all members leave). Thus, the expected number of renewed nodes can be expressed as

$$E[\mathcal{R}_{Batch, L>J=0}] = \begin{cases} \sum_{l=0}^{h-1} 2^l \left[1 - \frac{\binom{N-N/2^l}{L}}{\binom{N}{L}} \right] - L & \text{if } L < N \\ 0 & \text{if } L = N. \end{cases} \quad (5.8)$$

To calculate the expected number of secret key computations, we first derive the probability of renewing a node in terms of the number of departed descendants. When there is no node promotion, the node v is renewed if at least one but not all descendants of v leave the communication group. With node promotion, we have to exclude the counting of the renewed nodes that are pruned due to the departure of all descendants of their left or right subtree. The probability is thus given by

$$\begin{aligned} P[\text{node } v \text{ is renewed}] &= \sum_{i=1}^{N/2^l-1} \frac{\binom{N/2^l}{i} \binom{N-N/2^l}{L-i}}{\binom{N}{L}} - 2 \sum_{i=0}^{\frac{N}{2^{l+1}}-1} \frac{\binom{N/2^{l+1}}{i} \binom{N-N/2^l}{L-i-N/2^{l+1}}}{\binom{N}{L}} \\ &= \sum_{i=1}^{N/2^l-1} p_1(i) - 2 \sum_{i=0}^{\frac{N}{2^{l+1}}-1} p_2(i), \end{aligned} \quad (5.9)$$

where $p_1(i)$ is the probability that i members under the node v leave and $p_2(i)$ is the probability that all descendants under the left (or right) subtree of the node v leave and i members under the right (or left) subtree of the node v leave.

Let $M_v(l)$ be the expected number of members involved in the secret key computations of the node v . By considering how many members remain under the node v , the expected number of secret key computations is thus equal to

$$E[\mathcal{E}_{Batch, L>J=0}^s] = \sum_{l=0}^{h-1} 2^l M_v(l), \quad (5.10)$$

where $M_v(l)$ is given by

$$M_v(l) = \sum_{i=1}^{N/2^l-1} \left[\frac{N}{2^l} - i \right] p_1(i) - 2 \sum_{i=0}^{\frac{N}{2^{l+1}}-1} \left[\frac{N}{2^{l+1}} - i \right] p_2(i). \quad (5.11)$$

Case 3: $J = L > 0$. Consider again a node v at level l . The probability that the node v will be renewed is given by

$$\begin{aligned} P[\text{node } v \text{ is renewed}] &= 1 - P[\text{no member under node } v \text{ leaves}] \\ &= 1 - \frac{\binom{N-N/2^l}{L}}{\binom{N}{L}}. \end{aligned} \quad (5.12)$$

Thus, the expected number of renewed nodes is

$$E[\mathcal{R}_{Batch, J=L>0}] = \sum_{l=0}^{h-1} 2^l \left[1 - \frac{\binom{N-N/2^l}{L}}{\binom{N}{L}} \right]. \quad (5.13)$$

Similar to case 2 above, let us consider the expected number of members that compute the secret key of node v at level l , which is

$$\begin{aligned} M_v(l) &= \sum_{i=1}^{N/2^l} P[i \text{ members under node } v \text{ leave}] [N/2^l] \\ &= \frac{N}{2^l} \left[1 - \frac{\binom{N-N/2^l}{L}}{\binom{N}{L}} \right]. \end{aligned} \quad (5.14)$$

The expected total number of secret key computations is given by

$$E[\mathcal{E}_{Batch, J=L>0}^s] = \sum_{l=0}^{h-1} 2^l M_v(l) = N \sum_{l=0}^{h-1} \left[1 - \frac{\binom{N-N/2^l}{L}}{\binom{N}{L}} \right]. \quad (5.15)$$

Case 4: $J > L > 0$. In this case, the L leaving leaf nodes are replaced by the roots of the subtrees T_i^l 's consisting of J new members, where $1 \leq i \leq L$.

Also, these subtrees will introduce an extra $J - L$ renewed nodes. Using the result in case 3, the expected number of renewed nodes is

$$E[\mathcal{R}_{Batch, J>L>0}] = \sum_{l=0}^{h-1} 2^l \left[1 - \frac{\binom{N-N/2^l}{L}}{\binom{N}{L}} \right] + (J - L). \quad (5.16)$$

Among the L subtrees, the first $J \bmod L$ subtrees consist of $\lfloor \frac{J}{L} \rfloor + 1$ new members and require $\mathcal{E}_{Rebuild}^s(\lfloor \frac{J}{L} \rfloor + 1)$ secret key computations, and the rest require $\mathcal{E}_{Rebuild}^s(\lfloor \frac{J}{L} \rfloor)$ secret key computations. Let $J' = L$. The expected number of secret key computations is

$$\begin{aligned} E[\mathcal{E}_{Batch, J>L>0}^s] &= E[\mathcal{E}_{Batch, J'=L>0}^s] + (J \bmod L) \mathcal{E}_{Rebuild}^s(\lfloor \frac{J}{L} \rfloor + 1) \\ &\quad + (L - J \bmod L) \mathcal{E}_{Rebuild}^s(\lfloor \frac{J}{L} \rfloor) - Lh + Jh. \end{aligned} \quad (5.17)$$

Note that the second to the last term is to subtract the secret key computations of the leaf node which is now replaced by the root node of the L subtrees. The last term refers to the extra computations required by new members to obtain the keys along the key path of the original tree T .

Case 5: $L > J > 0$. In this case, we assume that the J newly joining members will randomly select L leaving leaf nodes for replacement as those leave positions are at the same level h . Using similar arguments as in case 2, since the actual number of pruned nodes is $L - J$, the expected number of renewed nodes is

$$E[\mathcal{R}_{Batch, L>J>0}] = \sum_{l=0}^{h-1} 2^l \left[1 - \frac{\binom{N-N/2^l}{L}}{\binom{N}{L}} \right] - (L - J). \quad (5.18)$$

Similar to the analysis in case 2, the probability of a node v being renewed at level l is equal to the probability of the node v considered to be renewed when no node promotion is performed, subtracting the probability of the node v considered to be renewed without node promotion but pruned with node

promotion. These two probabilities, denoted respectively p_1 and p_2 , are

$$\begin{aligned}
p_1 &= P[\text{node } v \text{ is renewed when no node promotion}] \\
&= \sum_{i=0}^{N/2^l-1} \sum_{k=i}^{N/2^l} P \left[\begin{array}{l} k \text{ members under} \\ \text{node } v \text{ leave} \end{array} \right] \times P \left[\begin{array}{l} k-i \text{ members join} \\ \text{under node } v \end{array} \right] \\
&\quad - P \left[\begin{array}{l} \text{no member under} \\ \text{node } v \text{ leave} \end{array} \right] \times P \left[\begin{array}{l} \text{no member joins} \\ \text{under node } v \end{array} \right] \\
&= \sum_{i=0}^{N/2^l-1} \sum_{k=i}^{N/2^l} \left(\frac{\binom{N/2^l}{k} \binom{N-N/2^l}{L-k}}{\binom{N}{L}} \cdot \frac{\binom{k}{k-i} \binom{L-k}{J-k+i}}{\binom{L}{J}} \right) - \frac{\binom{N-N/2^l}{L}}{\binom{N}{L}} \\
&= \sum_{i=0}^{N/2^l-1} p'_1(i) - \frac{\binom{N-N/2^l}{L}}{\binom{N}{L}}, \\
&\quad \text{where } p'_1(i) = \sum_{k=i}^{N/2^l} \frac{\binom{N-L}{N/2^l-k} \binom{J}{k-i} \binom{L-J}{i}}{\binom{N}{N/2^l}}. \tag{5.19}
\end{aligned}$$

$$\begin{aligned}
p_2 &= P[\text{a renewed node } v \text{ is pruned due to node promotion}] \\
&= 2 \sum_{i=0}^{\frac{N}{2^{l+1}}-1} \sum_{k=i}^{\frac{N}{2^{l+1}}} P \left[\begin{array}{l} \text{all members under the left} \\ \text{(or right) subtree leave and } k \\ \text{members under the right (or} \\ \text{left) subtree leave} \end{array} \right] \cdot P \left[\begin{array}{l} k-i \text{ members} \\ \text{join under the} \\ \text{right (or left)} \\ \text{subtree} \end{array} \right] \\
&= 2 \sum_{i=0}^{\frac{N}{2^{l+1}}-1} \sum_{k=i}^{\frac{N}{2^{l+1}}} \frac{\binom{\frac{N}{2^{l+1}}}{k} \binom{N-N/2^l}{L-k-\frac{N}{2^{l+1}}}}{\binom{N}{L}} \cdot \frac{\binom{k}{k-i} \binom{L-k-\frac{N}{2^{l+1}}}{J-k+i}}{\binom{L}{J}} \\
&= 2 \sum_{i=0}^{\frac{N}{2^{l+1}}-1} p'_2(i), \\
&\quad \text{where } p'_2(i) = \sum_{k=i}^{\frac{N}{2^{l+1}}} \frac{\binom{\frac{N}{2^{l+1}}}{k} \binom{N-N/2^l}{L-k-\frac{N}{2^{l+1}}}}{\binom{N}{L}} \cdot \frac{\binom{k}{k-i} \binom{L-k-\frac{N}{2^{l+1}}}{J-k+i}}{\binom{L}{J}}. \tag{5.20}
\end{aligned}$$

Thus, the probability that the node v is renewed is

$$P[\text{node } v \text{ is renewed}] = p_1 - p_2. \tag{5.21}$$

Hence, the expected number of secret key computations is given by

$$E[\mathcal{E}_{Batch, L>J>0}^s] = \sum_{l=0}^{h-1} 2^l M_v(l), \tag{5.22}$$

where $M_v(l)$ is given by

$$M_v(l) = \sum_{i=0}^{\frac{N}{2^l}-1} \binom{N}{2^l-i} p_1'(i) - \frac{N}{2^l} \cdot \frac{\binom{N/2^l}{L}}{\binom{N}{L}} - 2 \sum_{i=0}^{\frac{N}{2^{l+1}}-1} \binom{N}{2^{l+1}-i} p_2'(i). \quad (5.23)$$

5.1.3 Analysis of the Queue-batch Algorithm

The main idea of the Queue-batch algorithm exploits the idle rekeying interval to pre-process some rekeying operations. When we compare its performance with the Rebuild or Batch algorithms, we only need to consider the rekeying operations occurring at the beginning of every rekeying interval.

When $J = 0$, Queue-batch is equivalent to Batch in the pure leave scenario. For $J > 0$, the number of renewed nodes in Queue-batch during the Queue-merge phase is equivalent to that of Batch when $J = 1$. Thus, the expected number of renewed nodes is

$$E[\mathcal{R}_{Queue-batch}] = \begin{cases} 1 & \text{if } J > 0, L = 0 \\ \sum_{l=0}^{h-1} 2^l \left[1 - \frac{\binom{N-N/2^l}{L}}{\binom{N}{L}} \right] - L & \text{if } J = 0, L > 0 \\ \sum_{l=0}^{h-1} 2^l \left[1 - \frac{\binom{N-N/2^l}{L}}{\binom{N}{L}} \right] - (L-1) & \text{if } J > 0, L > 0. \end{cases} \quad (5.24)$$

Also, the expected number of exponentiations when $J > 0$ for Queue-batch is given by

$$E[\mathcal{E}_{Queue-batch}] = \begin{cases} N + J & \text{if } J > 0, L = 0 \\ E[\mathcal{E}_{Batch, L > J=0}] & \text{if } J = 0, L > 0 \\ E[\mathcal{E}_{Batch, J=1 \text{ and } L > 0}] - d + dJ & \text{if } J > 0, L > 0. \end{cases} \quad (5.25)$$

For $J > 0$ and $L > 0$, assume the new subtree is attached to a node at some level d . We first decrement d from $E[\mathcal{E}_{Batch, J=1 \text{ and } L > 0}]$ to exclude the secret key computations of the leaf node which is now replaced by the root node of the new subtree. We then add dJ to account for the secret key computations done by these new J members.

The value d is the level of the highest node that has all its descendants departed. Instead of computing the expected value of d , we can find an upper bound value for d , which occurs when the leaving leaf nodes are evenly distributed in the key tree. Thus, d is given by

$$d = \begin{cases} \lfloor \log_2(N - L) \rfloor + 1 & \text{if } N > L \\ 0 & \text{if } N = L. \end{cases} \quad (5.26)$$

5.2 Experiments

In the previous section, we quantified the performance measures by assuming that the existing tree is completely balanced. In this section, we perform a more elaborate performance study by investigating the costs of exponentiations and renewed nodes of the three proposed algorithms under different experimental settings. Besides, we also consider how many rounds the members take to generate the group key using different algorithms.

In the experiments, we assume a finite population of size 1024, among which 512 are originally in the communication group at the beginning of each experiment. We also assume that potential members outside the group have a tendency to join the group with the same join probability. Similarly, members within the group have a fixed leave probability of leaving the group. We let p_J and p_L denote the join and leave probabilities, respectively.

Experiment 1: (Comparison between individual rekeying and interval-based rekeying algorithms) We first demonstrate through simulations that interval-based rekeying outperforms individual rekeying. Given a number of join and leave requests, the individual rekeying approach first processes one by one the join requests followed by the leave requests. We ran the simulations over 300 rekeying intervals. Then we discarded the results of the first 50 rekeying intervals to avoid transient discrepancies. Finally we computed the average results of the remaining intervals.

Figs. 5.1, 5.2, and 5.3¹ illustrate the performance measures under different join and leave probabilities. These figures show that the three interval-based rekeying algorithms perform much better than the individual rekeying method. The advantage is even more prominent under high join and high leave probabilities. This implies that the interval-based rekeying algorithms can reduce the computation and communication costs of the a group is highly dynamic.

Experiment 2: (Evaluation based on mathematical models) This experiment evaluates the metrics of the three interval-based algorithms based on the mathematical models presented in the previous section. We started with a well-balanced key tree involving 512 members and then obtained the metrics under different values of joins and leaves (i.e., J and L).

Fig. 5.4 illustrates the average number of exponentiations and the average number of renewed nodes under different numbers of joining and leaving members. From these figures, we observe that Queue-batch outperforms the other two interval-based algorithms in all cases and there is a significant computation/communication reduction when the peer group is very dynamic (i.e., high number of members that wish to join or leave the communication group).

Besides, we observe from Fig. 5.4 that the metrics of Batch and Queue-batch exhibit a left-skewed bell-shaped pattern. To explain this behavior, we notice that at the beginning, the number of renewed nodes increases with the number of leaves and hence members have to rekey more nodes. However, as the number of leaves keeps increasing, the tree depth diminishes and members can take fewer rekeying steps to compute the group key. It is shown that the reduction of the tree depth begins to dominate the effect of the increase in the number of leaves when the number of leaves is around 100 to 200.

Experiment 3: (Average analysis at different fixed join probabilities) The previous experiment studies the case where the original tree is a balanced key tree. In this experiment, we further examine the case when the

¹Because of the large size of the figures, we present them at the end of this chapter.

key tree becomes unbalanced after many intervals of join and leave events. We varied the join probability p_J to be 0.25, 0.5, and 0.75, and then evaluated the average performance measures of the three algorithms under various leave probabilities.

The results are illustrated in Fig. 5.5. We observe that Queue-batch outperforms the other two algorithms in terms of the costs of exponentiation and renewed nodes in most cases. The exception is that Queue-batch needs more exponentiations than Batch when the leave probability is low (smaller than 0.2). The reason is that attaching the subtree of new members to an existing tree with few leaves may make the key tree unbalanced, leading to more computations in subsequent rekeying intervals. Moreover, the performance of Rebuild is the worst when p_L is low, but approaches that of Batch when p_L is high (e.g., both algorithms have similar average numbers of exponentiations and renewed nodes when p_L is higher than 0.6 and 0.8, respectively). In most situations, Queue-batch outperforms the other two algorithms at different join and leave probabilities. This shows that the pre-processing of the join requests in Queue-batch can significantly reduce the computation and communication loads at the rekeying intervals.

Experiment 4: (Instantaneous analysis at different join and leave probabilities) This experiment compares the instantaneous performance measures of Batch and Queue-batch over 300 rekeying intervals (we ignore Rebuild because it performs the worst among the three algorithms). We consider the cases with different values of p_J and p_L to represent different mobility characteristics of the peer group. In this experiment, we recorded the metrics at each rekeying interval.

Fig. 5.6 illustrates the instantaneous number of exponentiations at different values of p_J and p_L . It is interesting to note that when the group has a moderate to high leave probability, then Queue-batch significantly outperforms the Batch algorithm. Fig. 5.7 illustrates the instantaneous number of

renewed nodes. Queue-batch has a much lower cost in renewing nodes, as compared to the Batch algorithm. This implies that Queue-batch can reduce the communication cost significantly.

Experiment 5: (Performance analysis of Queue-batch with different reset intervals) Queue-batch does not reconstruct the whole key tree as Rebuild during the rekeying. Thus the key tree may become unbalanced after some rekeying intervals. In this experiment, we consider how Queue-batch performs if we reconstruct the key tree using the Rebuild algorithm every T_R rekeying intervals, where T_R is called the *reset interval*. This approach keeps the tree balanced at the cost of executing the Rebuild algorithm. We fixed $p_J = 0.5$ and $p_L = 0.25, 0.5,$ and 0.75 , and ran the simulations over 1000 rekeying intervals. Fig. 5.8 illustrates that the performance of Queue-batch remains approximately constant even at high reset intervals, meaning that Queue-batch can still preserve its performance without reconstructing the key tree after a long period of rekeying. This shows the robustness of the Queue-batch algorithm in maintaining a relatively balanced tree. This property is important because it can reduce the average costs of exponentiations and renewed nodes in the system.

Experiment 6: (Analysis in terms of number of rounds) In this experiment, we investigate the number of “rounds” required for the members to obtain the group key using different rekeying algorithms. We define one *round* as the period during which the group members can compute the secret keys as far up the key tree as they can. At the end of each round, all sponsors have to broadcast the blinded keys of the renewed nodes that have their secret keys computed so that other members can proceed with the secret key computations. In the analysis, we assume that rekeying is performed in *lock-step*, meaning that the two steps of secret key computations and blinded key broadcasts do not occur coincidentally.

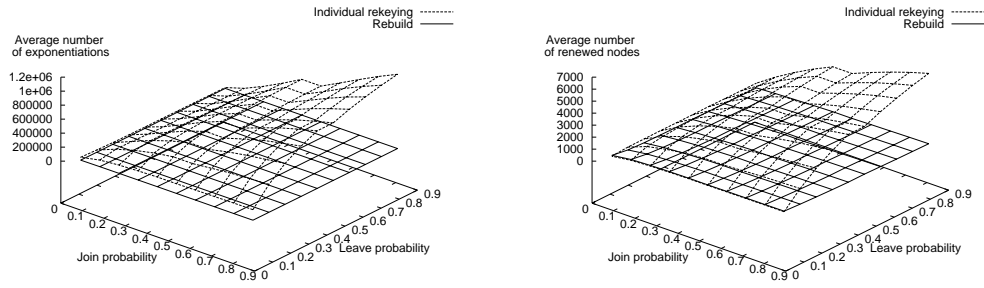
Fig. 5.9 illustrates the results in both the average and instantaneous cases.

At high leave probabilities, Queue-batch saves 3 to 4 rounds as compared to Rebuild and Batch. The savings are due to the preprocessing of join requests at the Queue-subtree stage. A fewer number of rounds is preferred as less message overhead is involved in processing rekeying messages and storing message headers.

5.3 Discussion of the experimental results

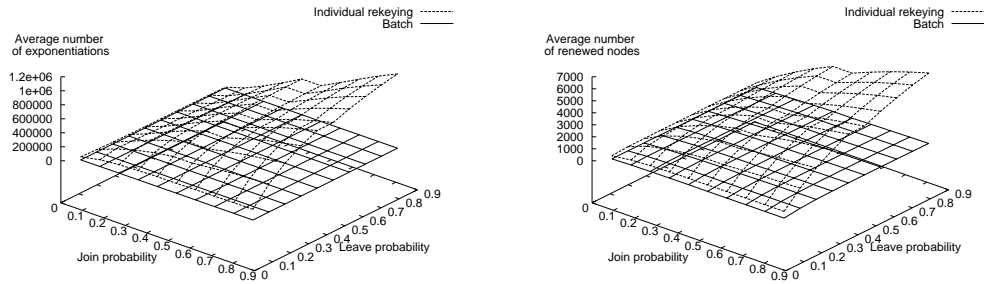
The above experiments show that under the stochastic settings, the interval-based algorithms offer better computation and communication performance than the individual rekeying approach and Queue-batch is the best among the three interval-based algorithms. The superior performance of Queue-batch is more obvious when the occurrences of join and leave events are highly frequent, and the reason is explained below. Moreover, Queue-batch demonstrates its robustness in keeping the key tree balanced and its capability in minimizing the number of rounds required.

To understand why Queue-batch outperforms more than the other two algorithms when the group is highly dynamic, we consider two cases: frequent joins and frequent leaves. When the number of join events is high, Queue-batch gains substantial performance advantages via the pre-processing of the join events in the Queue-subtree phase. Besides, when the number of leave events is high, Queue-batch reduces the depths of the existing tree nodes through node pruning. Batch, however, replaces the leaving leaf nodes with the joining ones and preserves the depths of the tree nodes. It implies Queue-batch requires fewer rekeying steps for the members whose associated leaf nodes are promoted to shallow positions. In combining two cases, Queue-batch can receive higher performance gains benefited from the frequent membership events.



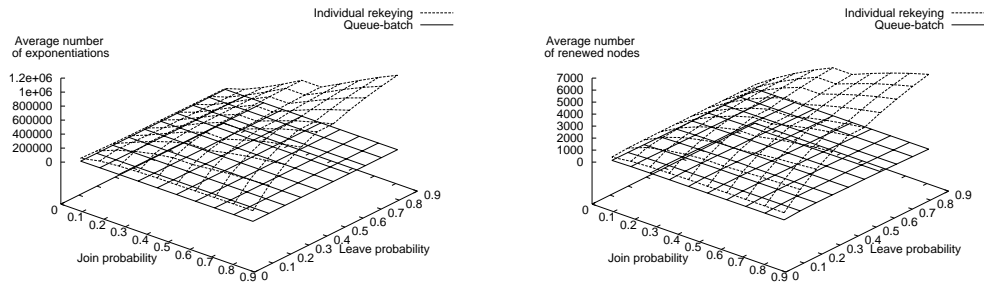
(a) Average number of exponentiations (b) Average number of renewed nodes

Figure 5.1: Performance differences between individual rekeying and Rebuild.



(a) Average number of exponentiations (b) Average number of renewed nodes

Figure 5.2: Performance differences between individual rekeying and Batch.



(a) Average number of exponentiations (b) Average number of renewed nodes

Figure 5.3: Performance differences between individual rekeying and Queue-batch.

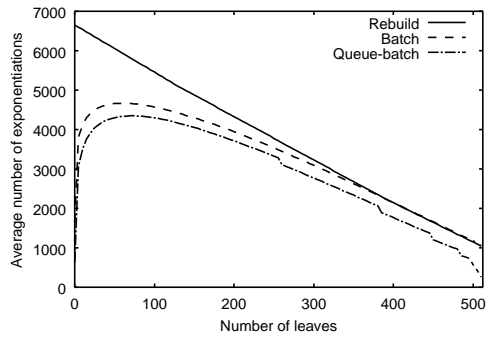
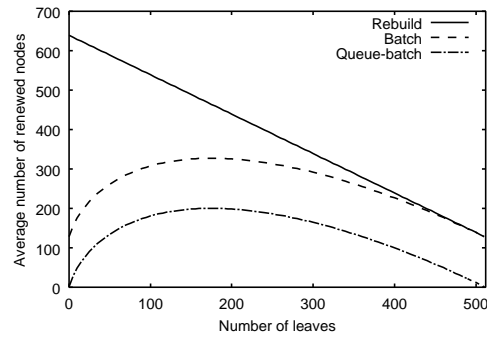
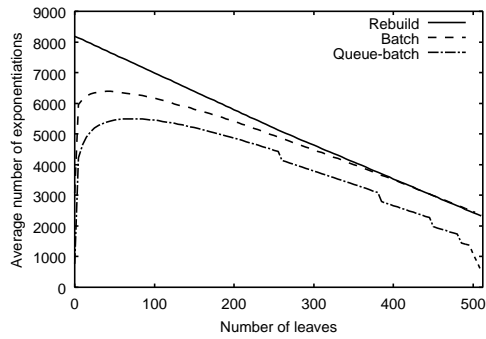
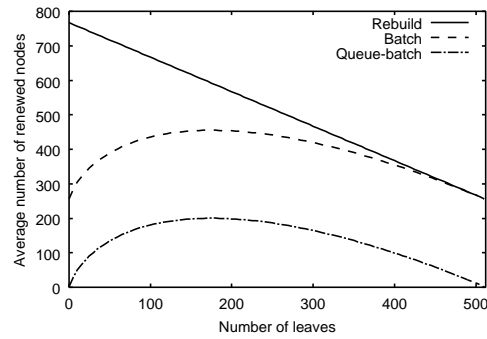
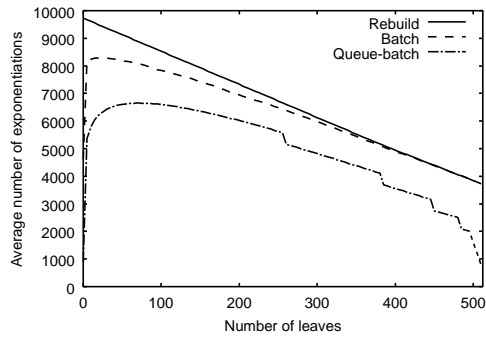
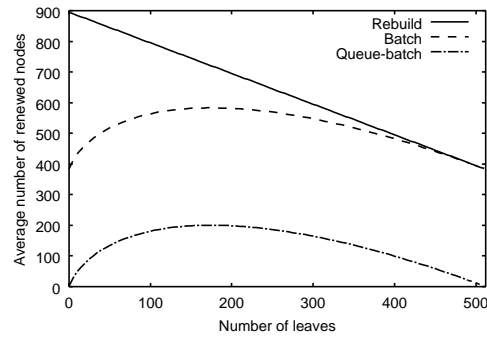
(a) Average number of exponentiations at $J = 128$ (b) Average number of renewed nodes at $J = 128$ (c) Average number of exponentiations at $J = 256$ (d) Average number of renewed nodes at $J = 256$ (e) Average number of exponentiations at $J = 384$ (f) Average number of renewed nodes at $J = 384$

Figure 5.4: Performance results of Rebuild, Batch, and Queue-batch at different numbers of joins based on mathematical models.

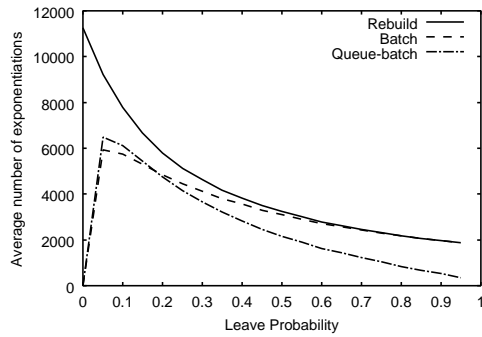
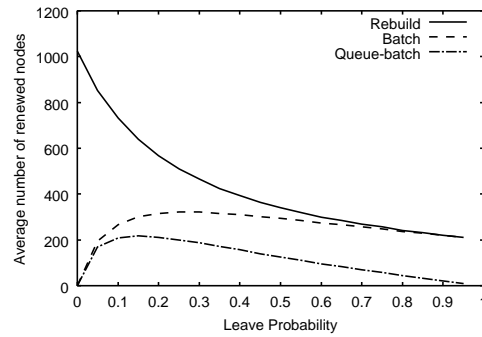
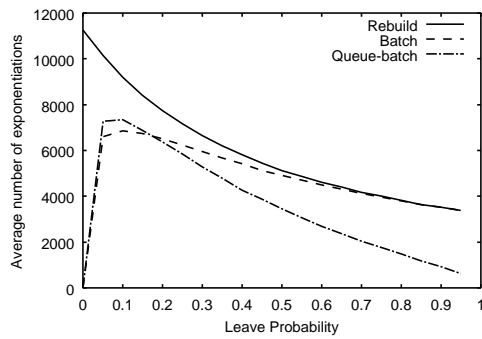
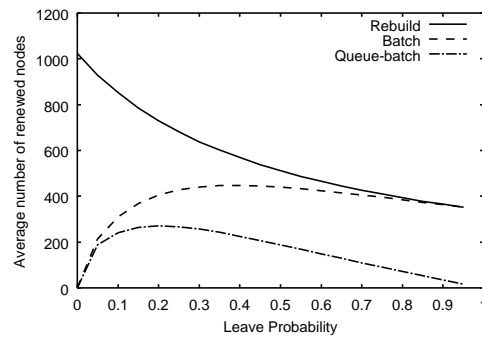
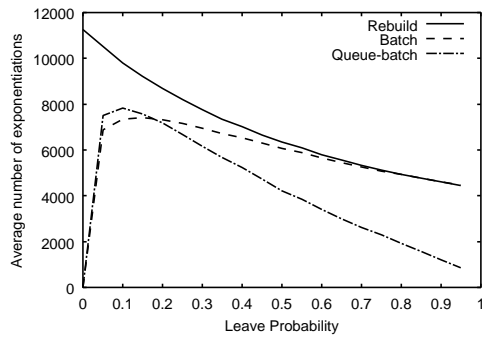
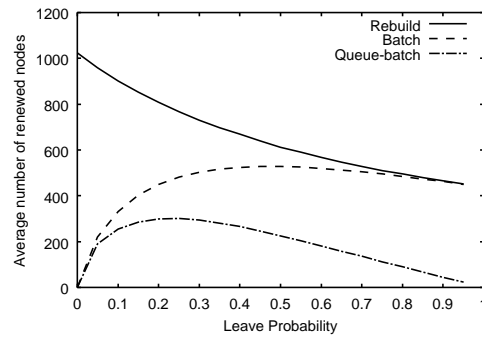
(a) Average number of exponentiations at $p_J = 0.25$ (b) Average number of renewed nodes at $p_J = 0.25$ (c) Average number of exponentiations at $p_J = 0.5$ (d) Average number of renewed nodes at $p_J = 0.5$ (e) Average number of exponentiations at $p_J = 0.75$ (f) Average number of renewed nodes at $p_J = 0.75$

Figure 5.5: Average performance results of Rebuild, Batch, and Queue-batch at different fixed join probabilities.

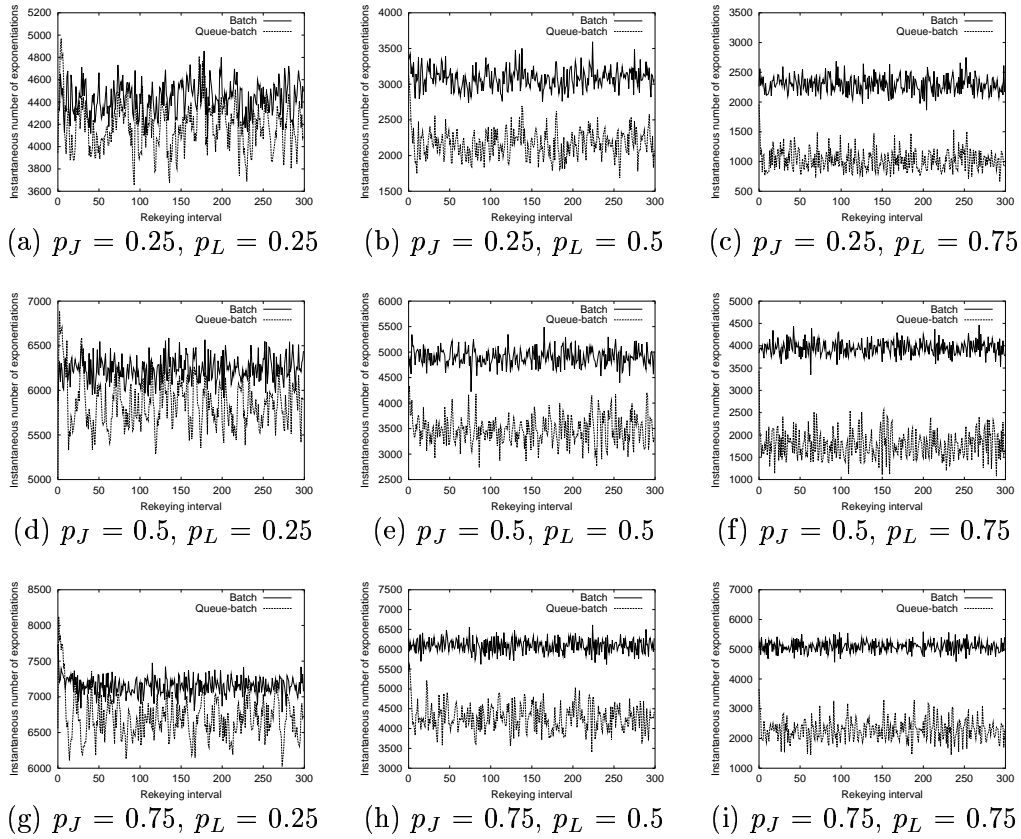


Figure 5.6: Instantaneous numbers of exponentiations of Batch and Queue-batch at different join and leave probabilities.

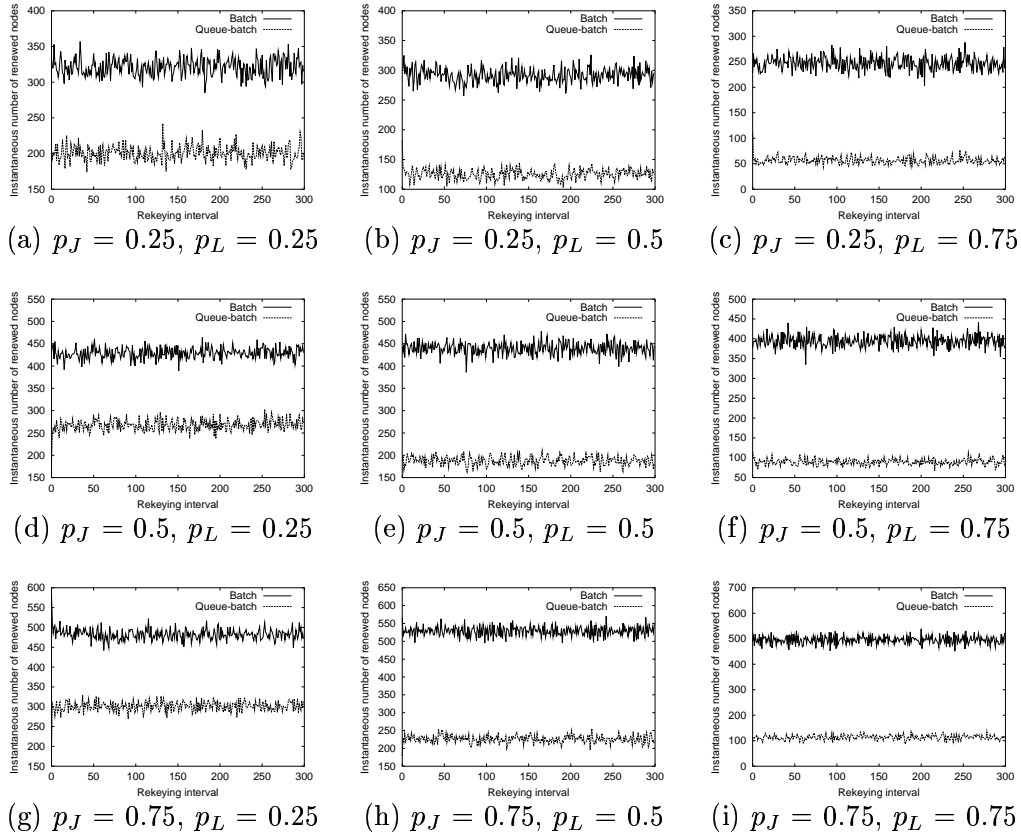


Figure 5.7: Instantaneous numbers of renewed nodes of Batch and Queue-batch at different join and leave probabilities.

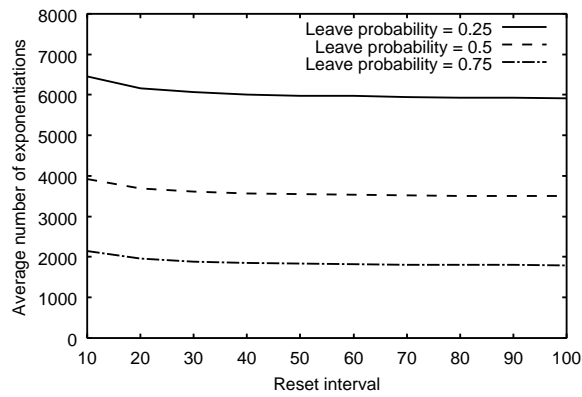
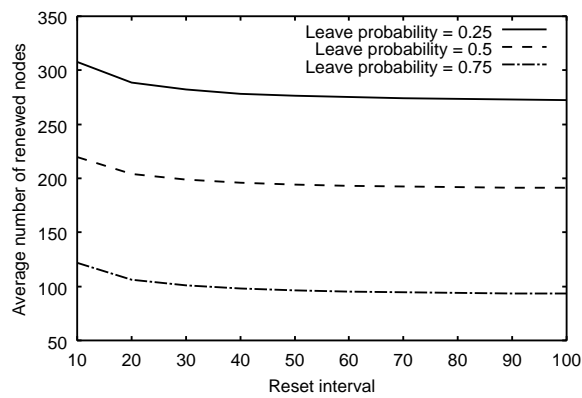
(a) Average number of exponentiations at $p_J = 0.5$ (b) Average number of renewed nodes at $p_J = 0.5$

Figure 5.8: Average performance results of Queue-batch at different reset intervals.

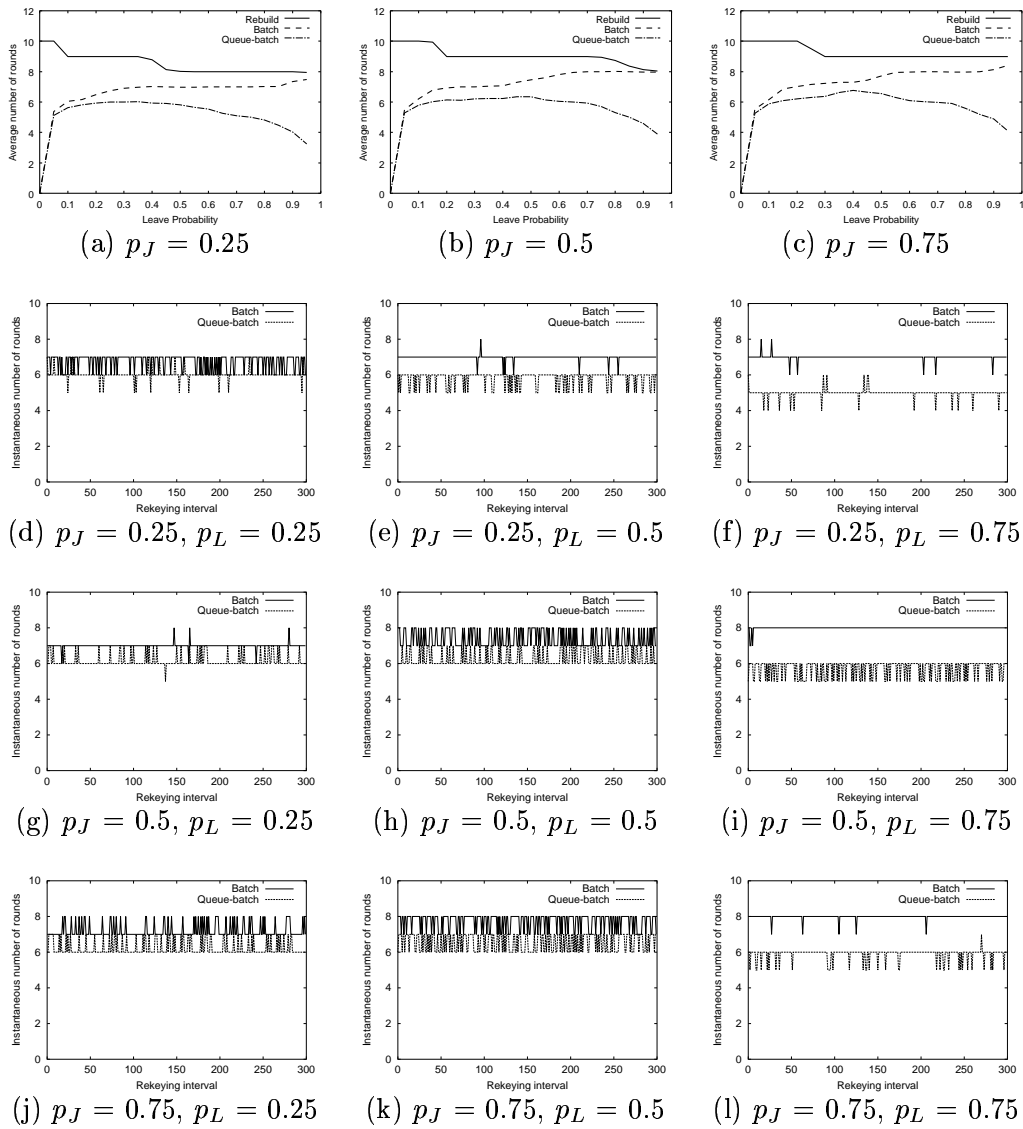


Figure 5.9: Average and instantaneous numbers of rounds of Rebuild, Batch, and Queue-batch at different join and leave probabilities.

Chapter 6

Authenticated Tree-Based Group Diffie-Hellman

The Diffie-Hellman protocol [5], which is the basic protocol for constructing TGDH, is vulnerable to the man-in-the-middle attack. To resolve the problem, we incorporate an authentication step into the Diffie-Hellman protocol. One simple approach is to require both parties to sign and certify their blinded keys; i.e., members exchange with each other a key message containing $\{BK, \text{sign}(BK)\}$, where $\text{sign}(\cdot)$ is the signature function to be applied to the blinded key BK . We may assume that both parties have already acquired the certificate of the other member from a trusted certificate authority (CA) to verify the signatures. The simple approach, however, incurs high computation cost in verifying signatures. More important, it is vulnerable to the substitution attack [6], meaning that an intruder may substitute its own signature for the signature of the other member.

To provide authentication in group key agreement, we devise an *Authenticated Tree-Based Group Diffie-Hellman* (A-TGDH) protocol. Our protocol extends the two-party authenticated Diffie-Hellman protocol proposed in [21], and our authentication protocol has the following security properties: (i) *key authentication* (i.e., all group members are assured that no outsiders can access the group key), (ii) *key confirmation* (i.e. all group members are assured

that they all possess the same group key), (iii) *known-key security* (i.e., the compromise of past short-term keys does not undermine the secrecy of future short-term keys), and (iv) *perfect forward secrecy* (i.e., the compromise of long-term keys does not undermine the secrecy of past short-term keys). Each member holds two types of keys: *short-term secret* and *blinded* keys, as well as *long-term private* and *public* keys.¹ Short-term keys (or *session* keys) are randomly generated when a member joins the group and becomes expired when the member leaves, while long-term keys (or *permanent* keys) remain static across many sessions and are certified by a trusted CA. Property (i) by itself provides *implicit* key authentication. If both properties (i) and (ii) are satisfied, the group key agreement scheme achieves *explicit* key authentication.

6.1 Description of A-TGDH

In the following description, we adopt several notations. As stated in Chapter 6, every node v in the key tree is associated with a secret key K_v and a blinded key BK_v . We then construct the *blinded key set* BK'_v , which, in general, refers to a number of copies BK_v 's respectively encrypted by the long-term private component of every descendant group member of node v (the mathematical formulation of BK'_v is presented below). The set of the descendant members of node v is given by \mathbf{M}_v . The i th member, M_i , holds a short-term secret key r_{M_i} and the corresponding blinded key $\alpha^{r_{M_i}} \bmod p$, as well as a long-term private key x_{M_i} and the corresponding public key $\alpha^{x_{M_i}} \bmod p$.

For simplicity, we assume that all group members acquire each other's certificates and hence long-term public keys from a trusted CA *before* the key agreement process starts. Otherwise, the process should include the steps of exchanging the certificates.

¹To distinguish between short-term and long-term key pairs, we use “secret key” and “blinded key” to represent short-term keys and the keys associated with the tree nodes, as well as “private key” and “public key” to refer to long-term keys.

We first review the two-party AK protocol given in [21]. Our presentation is based on the cyclic group of prime order p with generator α . Given two parties, say M_1 and M_2 , the AK protocol works as follows (all arithmetic operations are to be performed mod p , although the convention is omitted for brevity): M_1 sends $\alpha^{r_{M_1}}$ to M_2 and M_2 sends $\alpha^{r_{M_2}}$ to M_1 . M_1 computes $(\alpha^{x_{M_2}})^{r_{M_1}} \cdot (\alpha^{r_{M_2}})^{r_{M_1} + x_{M_1}} = \alpha^{r_{M_1}r_{M_2} + r_{M_1}x_{M_2} + r_{M_2}x_{M_1}}$. Analogous operations are performed by M_2 . The agreed session key is then given by $K = \alpha^{r_{M_1}r_{M_2} + r_{M_1}x_{M_2} + r_{M_2}x_{M_1}}$.

The AK protocol offers a number of advantages. It involves only two passes and thus saves communication cost. It achieves key authentication and known-key security [21]. If it is incorporated with key confirmation, it gives perfect forward secrecy as well [3].

We next extend the two-party AK protocol to our proposed A-TGDH protocol. In A-TGDH, we associate a node v with K_v and BK'_v as follows:

- If node v is a non-leaf node (with child nodes $2v + 1$ and $2v + 2$):

$$K_v = \alpha^k \text{ mod } p,$$

$$\text{where } k = K_{2v+1}K_{2v+2} + K_{2v+1} \sum_{M_i \in \mathbf{M}_{2v+2}} x_{M_i} + K_{2v+2} \sum_{M_i \in \mathbf{M}_{2v+1}} x_{M_i} \quad (6.1)$$

$$BK'_v = \begin{cases} \{\alpha^{K_v x_{M_i}} \text{ mod } p : M_i \in \mathbf{M}_{v+1}\} & \text{if node } v \text{ is the left child} \\ & \text{of its parent} \\ \{\alpha^{K_v x_{M_i}} \text{ mod } p : M_i \in \mathbf{M}_{v-1}\} & \text{if node } v \text{ is the right child} \\ & \text{of its parent} \\ \text{undefined} & \text{if node } v \text{ is the root node} \\ & \text{(i.e. } v = 0) \end{cases} \quad (6.2)$$

- If node v is a leaf node (associated with member M_i):

$$K_v = r_{M_i} \quad (6.3)$$

$$BK'_v = \alpha^{r_{M_i}} \text{ mod } p. \quad (6.4)$$

Thus, if a given node v needs to be renewed, a sponsor can simply broadcast BK'_v according to one of our interval-based rekeying algorithms. Also, any

member can still include its short-term blinded key (i.e., the blinded key of its corresponding leaf node) in its join request.

To achieve key confirmation, each member can broadcast the one-way function result of the group key after it is generated. However, this involves $O(N)$ broadcasts, where N is the number of members in the group, and this may be impractical. In an alternative approach given in [10], each member only needs to demonstrate its knowledge of the group key to its neighbors, provided that all the members are arranged in a linear chain. However, such an approach is vulnerable to the collusion attack [10]. To avoid the collusion attack problem, we propose the following. We divide a group into subgroups, such that members only confirm (via broadcasts) the group key with others *within the same subgroup*. The subgroups can be disjoint or intersected. The subgroup size and the number of subgroups are chosen depending on the desired level of security.

To illustrate how A-TGDH works, we consider a possible key tree formed after the rekeying process as shown in Fig. 6.1. Nodes 0, 1, and 2 are renewed nodes. Also, M_1 and M_3 are chosen to be the sponsors. Hence, the members perform the following steps (key confirmation is ignored):

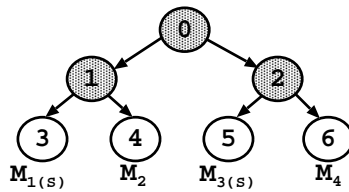


Figure 6.1: Example of authenticated key agreement involving 4 members.

- Since the blinded keys of leaf nodes are α^{rM_i} , for $i = 1, 2, 3$, and 4, the secret keys of nodes 1 and 2 are computed as

$$K_1 = \alpha^{rM_1rM_2+rM_1xM_2+rM_2xM_1}$$

$$K_2 = \alpha^{rM_3rM_4+rM_3xM_4+rM_4xM_3}.$$

- The sponsor M_1 broadcasts $\alpha^{K_1 x_{M_3}}$ and $\alpha^{K_1 x_{M_4}}$, and the sponsor M_3 broadcasts $\alpha^{K_2 x_{M_1}}$ and $\alpha^{K_2 x_{M_2}}$.
- M_1 and M_2 can retrieve α^{K_2} from $\alpha^{K_2 x_{M_1}}$ and $\alpha^{K_2 x_{M_2}}$, respectively. Similarly, M_3 and M_4 can retrieve α^{K_1} . Therefore, the members can compute the resulting group key, K_0 , as

$$K_0 = \alpha^{K_1 K_2 + K_1(x_{M_3} + x_{M_4}) + K_2(x_{M_1} + x_{M_2})}.$$

A-TGDH acquires a higher degree of security at the expense of involving more key exchanges and key computations. The trade-off study between security and performance is a classic problem and the right answer varies from applications. However, we point out that secure applications should include authentication in all situations since the man-in-the-middle attack can bring catastrophic consequences. Authentication can be achieved through either authenticated key agreement protocols such as A-TGDH or non-authenticated key agreement protocols that are protected with digital signatures.

6.2 Security Analysis

We argue that A-TGDH satisfies our stated security goals. We assume the existence of an active adversary E , which can inject, modify, and delete messages transferred between group members. The following arguments are mainly based on the *Diffie-Hellman problem* [5], i.e., given the elements α , p , $\alpha^x \bmod p$, and $\alpha^y \bmod p$, it is computationally infeasible to obtain $\alpha^{xy} \bmod p$ without knowing both x and y .

Theorem: *A-TGDH satisfies key authentication, key confirmation, known-key security and perfect forward secrecy.*

Proof: We show that the protocol satisfies the following security properties:

1) *Key authentication.* Suppose E replaces $\alpha^{K_v x_{M_i}}$ with $\alpha^{e(v,i)}$, for all possible public components involving node v and member M_i . In this case, a legitimate member M_i will compute the secret key of some node v_o as $K_{v_o} = E_c \cdot \alpha^{e(v_o,i)} x_{M_i}^{-1} \cdot K_{2v_o+1}$ (assuming that M_i holds the secret component K_{2v_o+1}), where E_c is the product of the public components obtainable by E . However, it is known to be computationally infeasible for E to obtain K_{v_o} without knowing K_{2v_o+1} and x_{M_i} (this is the Diffie-Hellman problem). Hence, A-TGDH provides the key authentication property.

2) *Key confirmation.* As stated in the previous sub-section, a group can achieve different security levels of key confirmation determined by the subgroup size and the number of subgroups. The larger the subgroup size, the higher the degree of key confirmation that can be achieved. Also, if there are many subgroups and all of them intersect, we can obtain a higher level of key confirmation.

3) *Known-key security.* It should be noted that the authenticated group key K_0 consists of a secret random component equivalent to the group key of the non-authenticated TGDH. If E compromises this authenticated group key K_0 , it cannot compute the past group keys as their corresponding secret random components are composed of the short-term secrets r_{M_i} 's offered by different combinations of members, and doing so will require E to solve the Diffie-Hellman problem. If any two past group keys refer to the same set of members, they are still different since each member M_i renews r_{M_i} when it re-joins the group.

4) *Perfect forward secrecy.* We want to prove that the secret keys of all non-leaf nodes provide perfect forward secrecy. We prove this property by induction on the levels of the tree which has the lowest level h .

- **Basis.** Consider a node v_o at level $h - 1$ whose children are both leaf nodes associated with members M_{i1} and M_{i2} . Given the long-term private keys $x_{M_{i1}}$ and $x_{M_{i2}}$, the adversary E cannot compute $K_{v_o} =$

$\alpha^{r_{M_{i1}}r_{M_{i2}}+r_{M_{i1}}x_{M_{i2}}+r_{M_{i2}}x_{M_{i1}}}$, since computing $\alpha^{r_{M_{i1}}r_{M_{i2}}}$ without knowing $r_{M_{i1}}$ and $r_{M_{i2}}$ is the Diffie-Hellman problem.

- **Induction hypothesis.** Suppose that the keys of nodes $2v + 1$ and $2v + 2$ at some level l , where $0 < l \leq h - 1$, give perfect forward secrecy.
- **Induction step.** Consider the node v at level $l - 1$. Given only the long-term private keys, we cannot deduce K_{2v+1} and K_{2v+2} (by hypothesis). This implies K_v cannot be computed as it contains the component $\alpha^{K_{2v+1}K_{2v+2}}$.

Thus, by induction, E cannot compute the secret keys of all non-leaf nodes given only the long-term private keys, i.e., those keys satisfy perfect forward secrecy.

Chapter 7

Implementation and Applications

We implemented a Linux-based C language application programming interface (API) library based on our interval-based algorithms. The API library, called the *Secure Group Communication Library (SGCL)*¹, provides necessary software components for developers to write secure group-oriented applications. To realize how to use the library, we built two demo applications: *Chatter* and *Gauger*. *Chatter* is a secure chat-room application which allows group members to communicate in plain messages that can be encrypted in real-time. It supports both graphical and text modes. *Gauger*, on the other hand, aims to analyze the performance of the interval-based algorithms under real network settings. It can measure various performance metrics, such as the rekeying time, the number of exponentiations in the group key generation and the number of blinded key broadcasts, during a rekeying operation. Both applications reveal the strengths of using SGCL in the development of secure group-oriented applications for a peer-to-peer or mobile ad-hoc environment.

This chapter covers the implementation issues regarding SGCL. In Section 7.1, we first discuss two special member roles, *leader* and *sponsor*, and

¹For details about SGCL as well as its source codes, please refer to: <http://www.cse.cuhk.edu.hk/~cslui/ANSRlab/software/SGCL/index.html>.

explain their functions in facilitating the implementation. In Section 7.2, we present the system architecture of the library. In Section 7.3, we introduce the API functions and describe their properties. In Section 7.4, we give experimental results obtained from Gauger and study the performance of the interval-based algorithms under a real network environment. In Section 7.5, we introduce Chatter, a real-life application developed under SGCL, and suggest other potential applications where SGCL fits their development needs. Finally, in Section 7.6, we discuss possible future extensions that enhance the security of the implementation.

7.1 Leader and Sponsors

The interval-based algorithms mentioned in Chapter 4 are built upon two important assumptions. First, all group members are synchronized to conduct rekeying operations periodically. Second, the sponsors know how to coordinate with each other so that they refrain from broadcasting the same blinded key more than once. In this section, we consider how to implement these assumptions. We begin with the introduction of a new role, called the *leader*, whose responsibility is to notify members to start a rekeying operation synchronously. We describe how a leader is elected to carry out its designated duties. Also, we discuss how sponsors are elected and coordinate with each other to minimize the number of communication rounds required for broadcasting renewed blinded keys. At the end of this section, we summarize the working idea of a rekeying operation in the presence of the leader and sponsors.

7.1.1 Leader

The leader is the single member that is responsible for periodically notifying all group members to start a rekeying operation at regular rekeying intervals, for instance, via the broadcast of a rekeying message to all group members.

Such a role is necessary because of two reasons. First, the members may not share a global clock to synchronize on performing rekeying operations. Second, new members do not know the rekeying information including the present join and leave events as well as the existing key tree when they join the group. Although the leader can provide such rekeying information specifically for each new joining member, the processing load of the leader will become significant when the number of joining members is very high. To address both issues simultaneously, not only should the leader periodically broadcast a rekeying message to notify others to start a rekeying operation, but the leader also needs to include in the rekeying message the join and leave events as well as the structure of the current key tree that are to be manipulated in the rekeying operation. In order to achieve both purposes, we elect the member that stays in the communication group for the longest time to be the leader.

A group member carries out leader election in two scenarios: (1) when the group member newly joins the group and (2) when the leader leaves the group. In either scenario, the group member first decides if it is the leader by checking if it stays the longest in the group. Assume that a group member recognizes the current membership when it joins the group. Then it can make the decision by checking if it is the first member in the group (for scenario 1) or if all other members that are initially in the group have departed (for scenario 2). If it is the leader, it starts periodically broadcasting rekeying messages to notify others to start a rekeying operation. Otherwise, it waits for the first incoming rekeying message and concludes that the sender of the received rekeying message to be the leader.

It is possible that a newly elected leader does not know the current key tree structure. This occurs when it has joined the group for some time and has not started any rekeying operation. In this case, the leader should include only an empty tree and the join events in the rekeying notification. The leave events, however, are not required as they do not take effect in an empty tree.

7.1.2 Sponsors

Sponsors, as previously stated, refer to the group members that need to broadcast the blinded keys associated with the nodes in a key tree during a rekeying operation. To determine which blinded keys each sponsor should broadcast, we first set an implementation requirement: each member only holds the blinded keys along its *co-path* [11], which is defined as the sequence of nodes whose siblings belong to the key path of the member. This implementation requirement is tighter than the assumption stated in our performance evaluation analysis in Chapter 5, that is, each group member holds *all* the blinded keys within the key tree. Thus, members can be benefited from less storage overhead for the blinded keys. As each member holds the blinded keys along its own co-path, the sponsors have to broadcast the blinded keys of the non-renewed nodes which are the children of the renewed nodes so that members can compute the secret keys of the renewed nodes. Broadcasting non-renewed blinded keys is essential for the new members which know nothing before they join the group as well as for the existing members whose co-path does not include the non-renewed nodes prior to the rekeying operation (we will later illustrate how it helps the existing members with an example). Therefore, in our implementation, we appoint the sponsors to broadcast the blinded keys of two types of nodes: (1) all renewed nodes and (2) the non-renewed nodes whose parents are renewed nodes.

In order that the group members only need to store the blinded keys in their co-paths, we refine the sponsor election criterion: in each rekeying interval, a member becomes a sponsor if it is the rightmost member of the subtree whose root is a non-renewed node but the parent of the root is a renewed node (note that if the member is the only member in the group, no sponsor is elected). It should be noted that all new members are elected to be sponsors based on this criterion.

After being elected, the sponsors have to decide the exact nodes whose corresponding blinded keys need to be broadcast. For efficiency, this decision-making process, which we call *sponsor coordination*, should satisfy three properties: (1) *self-computable*, i.e., the sponsors need not communicate with other sponsors to make the decision, (2) *lightweight*, i.e., the process itself is simple, and (3) *broadcast-efficient*, i.e., the process leads to a minimum number of broadcasts of blinded keys. These properties ensure that sponsor coordination introduces little processing overhead to rekeying operations.

Fig. 7.1 presents the pseudo-code of the sponsor coordination algorithm. To illustrate, consider Fig. 7.2, in which the key tree contains a number of renewed nodes (i.e., nodes 0, 1, 2, and 6). Based on our sponsor election criterion, M_1 , M_2 , M_5 , M_7 , and M_8 are elected to be sponsors. According to the algorithm in Fig. 7.1, member M_1 broadcasts BK_3 , M_2 broadcasts BK_4 and BK_1 , M_5 broadcasts BK_5 , M_7 broadcasts BK_{13} , and M_8 broadcasts BK_{14} , BK_6 , BK_2 , and BK_0 . Furthermore, Fig. 7.2 illustrates the need of broadcasting the blinded keys of non-renewed nodes (i.e., nodes 3, 4, 5, 13, and 14) to existing members. For example, M_6 and M_7 do not hold the blinded key of node 14 if it is promoted from one of its child nodes since the node is not originally on their co-paths. Therefore, they have to obtain the blinded key from the sponsors.

The sponsor coordination algorithm satisfies the three properties, i.e., self-computable, lightweight and broadcast-efficient. It is self-computable because it does not involve any communication between sponsors to determine which blinded keys to be broadcast. Also, it is lightweight because it only requires a member to traverse its key path *once*. Finally, it is broadcast-efficient since it broadcasts the keys in a minimum number of rounds. To elaborate the last property, we consider Fig. 7.3 in which a renewed node v_p is the root of a subtree and has two child nodes, v_l and v_r , whose corresponding sponsors are $M_{l(s)}$ and $M_{r(s)}$, respectively. To decide which sponsor is to be selected

Sponsor_Coordination (T)

```

/*  $T$  is the updated key tree with a number of renewed nodes */
1.  $broadcast\_list = \text{NULL}$ ;
2. if (sponsor) { /* responsibility of the sponsor */
3.    $k\_node =$  leaf node of the member's key path;
4.   while ( $k\_node$  is not  $T$ 's root and  $k\_node$ 's parent is not renewed)
5.      $k\_node = k\_node$ 's parent;
6.   insert  $k\_node$  into  $broadcast\_list$ ;
7.   while ( $k\_node \neq T$ 's root) {
8.     if (both  $k\_node$  and  $k\_node$ 's sibling are not renewed or
9.         both  $k\_node$  and  $k\_node$ 's sibling are renewed) {
10.      if ( $k\_node$  is the right child)
11.        insert  $k\_node$ 's parent into  $broadcast\_list$ ;
12.      else
13.        break the while loop;
14.    } else if ( $k\_node$  is not renewed and  $k\_node$ 's sibling is renewed) {
15.      break the while loop;
16.    } else if ( $k\_node$  is renewed and  $k\_node$ 's sibling is not renewed) {
17.      insert  $k\_node$ 's parent into  $broadcast\_list$ ;
18.    }
19.     $k\_node = k\_node$ 's parent;
20.  } /* end of while loop */
21. } /* end of if (sponsor) condition */
return  $broadcast\_list$ ;

```

Figure 7.1: Pseudo-code of the sponsors coordination algorithm.

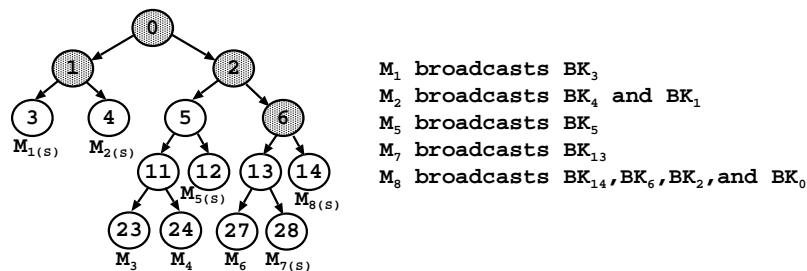
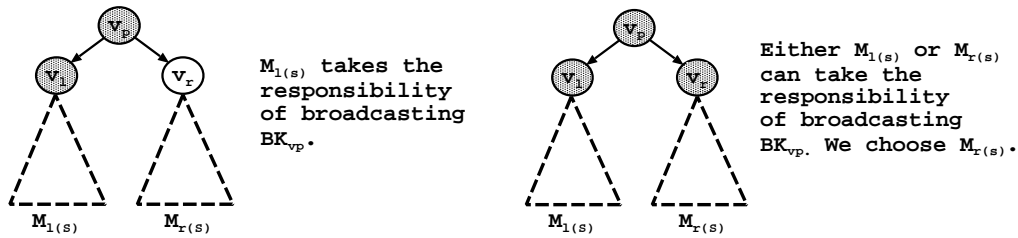


Figure 7.2: Example to illustrate the sponsor coordination algorithm in Fig. 7.1.

to broadcast the blinded key of v_p , we consider two cases. First, if only one child node is renewed, say v_l is renewed but v_r is not, $M_{l(s)}$ can compute the secret key of v_p based on the unchanged blinded key of v_r . This implies that $M_{l(s)}$ can broadcast the blinded keys of v_l and v_p in one round. We therefore select $M_{l(s)}$ to broadcast the blinded key of v_p . Second, if both child nodes are renewed, i.e., v_l and v_r are renewed nodes, both sponsors have to wait for the updated blinded keys of v_l and v_r to compute the secret key of v_p . They need two rounds to broadcast the blinded key of v_p . We can therefore select any one sponsor, say the sponsor $M_{r(s)}$ under the right child node, to take this responsibility. Combining two cases, we can apply similar arguments when v_l is not a renewed node but v_r is and when both v_l and v_r are not renewed nodes. Therefore, we conclude that the sponsor coordination algorithm is self-computable, lightweight and broadcast-efficient and is adequate to be put into implementation.



(a) case 1: one child node is renewed. (b) case 2: both child nodes are renewed.

Figure 7.3: Illustration of the broadcast-efficient property of the sponsor coordination algorithm.

7.1.3 Rekeying Operation

We summarize how the leader and sponsors co-operate with all group members in conducting a rekeying operation. At regular rekeying intervals, the leader broadcasts a rekeying message to signal all group members to start a rekeying operation. Upon receiving the rekeying message, all group members update their key tree based on the agreed interval-based algorithm. They then elect

the corresponding sponsors, which broadcast renewed blinded keys to all group members. Consequently, every group member can compute the group key.

7.2 System Architecture

In this section, we provide an architectural overview of SGCL in three areas: (1) preliminary requirements for the library, (2) description of the software components, and (3) implementation considerations.

7.2.1 System Preliminaries

SGCL is implemented in C under Linux and requires two toolkits: *Spread* [22] and *OpenSSL* [16]. *Spread* is a group communication model incorporated with reliable and ordered message delivery. It offers the view synchrony feature [7] that ensures all messages from a communication group are delivered error-free and in sequence under the same membership view. In our implementation, we require that SGCL first connects to a *Spread* daemon, which maintains the reliable and ordered group communication, and then uses the exported API functions from *Spread* to send or receive packets through the daemon. *OpenSSL*, on the other hand, is a security toolkit offering a cryptography library and a certificate generation tool. We use it to implement cryptographic algorithms, such as Diffie-Hellman, as well as to create public-key certificates for the authentication of group members. Both toolkits are the pre-requisites of the SGCL development.

We provide an optional member authorization feature, which is known as the *SIGNATURE* mode, to indicate that group members need to apply digital signatures to the packets to be sent. Prior to enabling the *SIGNATURE* mode, a group member should first obtain its public-key certificate and the corresponding long-term (or permanent) private key from a trusted certificate authority (CA). Then the member signs the packets with its long-term private

key before the packets are sent over the network. In our implementation, we select the X509 certificate standard [9] and the 1024-bit RSA [18] with SHA-1 [15] signature scheme. If the transmission channel is authentic itself, the non-SIGNATURE mode can be used to eliminate the costs of the signature and verification operations. In most cases, however, the SIGNATURE mode should be activated.

The implementation of SGCL contains several requirements. First, we require SGCL to support reliable and ordered message delivery in view synchrony and to implement cryptographic protocols, and hence both Spread and OpenSSL should be pre-installed in a Linux system. Besides, the Diffie-Hellman parameters, which are 1024-bit long in our implementation, should have been initialized and stored in the SGCL source directory before SGCL starts running. In addition, each group member should have a configuration file stating the unique member identifier, the membership details of all possible communication groups, and the connectivity information specifying which Spread daemon is to be connected. Furthermore, in the SIGNATURE mode, each group member should beforehand obtain its long-term private key and the certificates of other group members from a trusted CA. For consistency, a central repository can be set up to provide all necessary information related to the requirements.

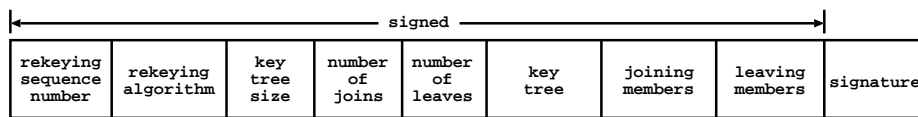
7.2.2 System Components

SGCL is composed of four types of components: (1) *engines*, the entities which hold variables and methods for various functionalities, (2) *queues*, the linked-list structures which store and dispatch packets in the first-in-first-out manner, (3) *threads*, the processes which handle all protocol operations, and (4) *packets*, the information which is exchanged between group members. Details of the components are summarized in Table 7.1.

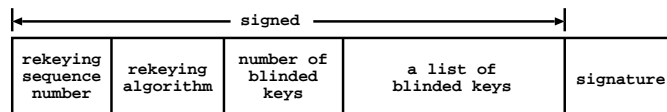
Types	Components	Synopsis
Engines	certkey_engine	It holds the long-term private key of the corresponding group member and the certificates of all group members. It also provides function calls for the signature and verification operations on the packets. It is used only if the SIGNATURE mode is activated.
	keytree_engine	It holds the key tree. In every rekeying operation, it updates the key tree based on the agreed interval-based algorithm and returns the resultant renewed nodes and sponsors.
	leader_engine	It stores the current leader in the group and performs leader election if necessary. It also holds the <i>rekeying sequence number</i> , which is used to identify the REKEY packets to be sent.
	member_engine	It holds the current membership information, including the existing members in the group, as well as the joining and leaving members to be processed in the next rekeying interval.
	packet_engine	It creates packets according to the given parameters and sends packets to the network.
	sesskey_engine	It represents the session key structure, which stores the Diffie-Hellman parameters, the secret and blinded keys along the key path of the corresponding member, as well as the group key.
Queues	message_queue	It stores the MESSAGE packets to be retrieved by the application.
	packet_queue	It stores all types of packets received from the group.
	rekey_queue	It stores the <i>rekeying signals</i> , which are later transformed to the REKEY packets to be sent to the group. It is used by the leader only.
Threads	receive_thread	It receives packets from the group and adds them to the packet queue.
	process_thread	It retrieves packets from the packet_queue and processes them.
	rekey_send_thread	It retrieves the REKEY packets from the rekey_queue and sends them to the group. It is used by the leader only.
	rekey_poll_thread	It periodically inserts a rekeying signal to the rekey_queue so that the rekey_send_thread can retrieve the signal and send a REKEY packet. It is used by the leader only.
Packets	JOIN packet	It indicates a joining member.
	LEAVE packet	It indicates a leaving member that gracefully leaves the group.
	DISCONNECT packet	It indicates a leaving member that ungracefully leaves the group.
	REKEY packet	It denotes the rekeying message signaling the group members to start a rekeying operation. It stores the rekeying sequence number for identifying the current rekeying operation, the rekeying algorithm to be used, the joining and leaving members, as well as the key tree that is to be updated in the rekeying operation.
	BKEY packet	It holds a sequence of blinded keys of part of the key tree nodes in a key path.
	MESSAGE packet	It represents the application-level data to be processed by the application. It also includes the rekeying sequence number to specify which group key is used for encryption.

Table 7.1: Description of components used in SGCL.

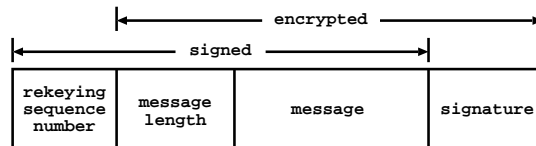
The SGCL packets are classified into two categories: *membership packets* and *regular packets*. Membership packets, including the JOIN, LEAVE, and DISCONNECT packets, are defined in the Spread specification [22]. They store the membership information essential for SGCL and the Spread daemons. Regular packets, however, are defined by SGCL. They are used by SGCL for rekeying operations and by underlying group-oriented applications for secure group communication. They include the REKEY, BKEY, and MESSAGE packets. Fig. 7.4 illustrates how SGCL defines the formats of the regular packets.



(a) REKEY packet



(b) BKEY packet



(c) MESSAGE packet

Figure 7.4: Formats of the regular packets.

SGCLs residing in the group-oriented applications operate and exchange packets with one another in order to achieve their functions. The general operations on the received packets can be summarized into several steps (assume that the SIGNATURE mode is activated): (1) The *received_thread* receives packets from the connected Spread daemon and adds them into the *packet_queue*. (2) The *process_thread* retrieves packets from the *packet_queue* if the queue is non-empty. (3) The *process_thread* verifies the signatures attached to the packets. (4) Based on the packet types, the *process_thread* carries out the corresponding

operations with the *member_engine*, the *leader_engine*, the *keytree_engine*, the *sesskey_engine*, and the *message_queue*. (5) If the *process_thread* needs to send packets, it creates packets with the *packet_engine*. (6) The *process_thread* signs the packets with the *certkey_engine*. (7) Finally, the *process_thread* sends the packets via the *packet_engine* to the connected Spread daemon and then to the communication group. Fig. 7.5 illustrates the general idea of the operations on the received packets.

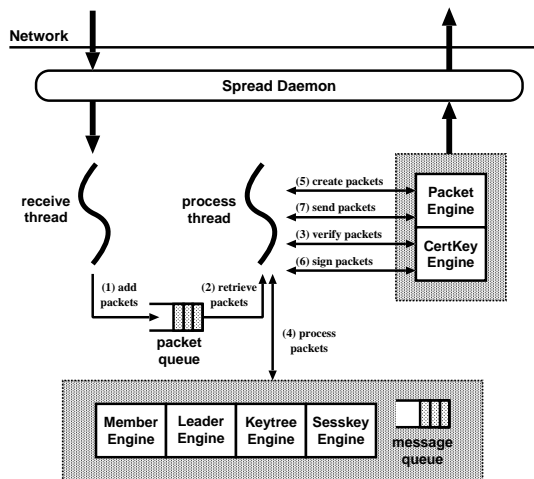


Figure 7.5: Overview of general operations on received packets.

Let us take a more detailed look into the operations on how the *process_thread* responds to the received packets according to different packet types:

- **Operations on a received JOIN/LEAVE/DISCONNECT packet:**

The *process_thread* inserts the joining or leaving member into the *member_engine*. Also, depending on the membership events, it performs leader election, and creates the leader-specific components if the member becomes the leader (the operations of the leader-specific components are described later in this subsection). It should be noted that the operations on the LEAVE and DISCONNECT packets are both identical.

- **Operations on a received REKEY packet:** The *process_thread* first

retrieves the rekeying information including the rekeying sequence number (the identifier of a REKEY packet and hence a rekeying operation), the joining and leaving members, as well as the key tree, from the received REKEY packet. The *process_thread* then starts the rekeying operation, which consists of (1) specifying the leader's identity in the *leader_engine*; (2) synchronizing the joining and leaving members with those in the *member_engine*; (3) updating the key tree in the *keytree_engine* based on the selected interval-based algorithm; and (4) updating the secret and blinded keys of the key path in the *sesskey_engine* and broadcasting any blinded keys if the group member becomes a sponsor.

- **Operations on a received BKEY packet:** The *process_thread* obtains the blinded keys from the packet, which is composed of a sequence of blinded keys of some key tree nodes in a key path. If the blinded keys help the group key generation, the *process_thread* computes the secret keys along the key path, which is maintained by the *sesskey_engine*. Besides, if the group member is a sponsor and the *sesskey_engine* contains the new blinded keys to be broadcast, the *process_thread* will broadcast a BKEY packet consisting of the blinded keys.
- **Operations on a received MESSAGE packet:** The *process_thread* inserts the MESSAGE packet, which contains the application-level messages, into the *message_queue*. The enqueued packets will later be retrieved by the *SGCL_recv()* function (described in the next subsection) and processed by the application.

If a group member is elected to be leader, the *process_thread* will create the leader-specific components, composed of the *rekey_poll_thread*, the *rekey_send_thread*, and the *rekey_queue*. To send a rekeying message to the group, the leader performs the following procedures: (1) The *rekey_poll_thread* periodically issues a rekeying signal (the indicator of performing a rekeying

operation) into the *rekey_queue* and notifies the *rekey_send_thread* to send REKEY packets. The rekeying signal also specifies the interval-based algorithm to be performed. (2) When the *rekey_send_thread* is notified, it removes the rekeying signal from the *rekey_queue*. (3) The *rekey_send_thread* gathers the rekeying sequence number from the *leader_engine*, the joining and leaving members from the *member_engine*, and the existing key tree from the *keytree_engine*. (4) Then the *rekey_send_thread* constructs the REKEY packet based on the gathered details. (5) The *rekey_send_thread* signs the REKEY packet with the *certkey_engine*. (6) Finally, the *rekey_send_thread* sends the packet over the network.

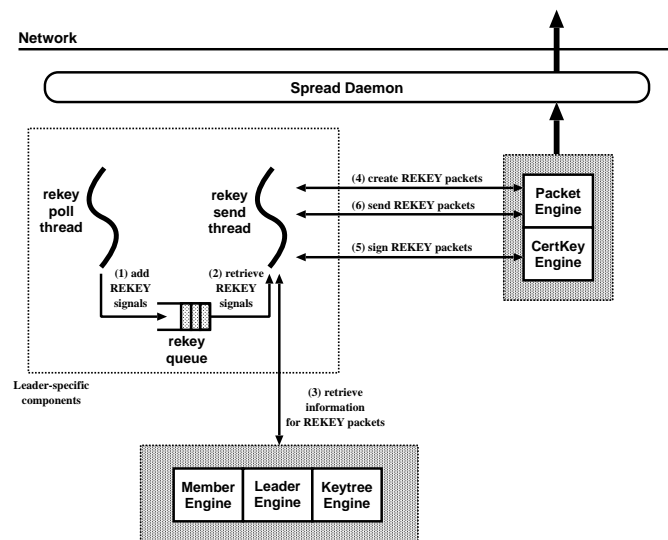


Figure 7.6: Overview of leader-specific components and their relationships with other components.

To ensure that the REKEY packet contains the updated information, the *rekey_send_thread* should retrieve the next rekeying signal from the *rekey_queue* and sends its corresponding REKEY packet *only after* all engines are updated regarding the previous rekeying operation. Fig. 7.6 illustrates the leader-specific components and their interactions with other components.

7.2.3 Implementation Considerations

In this subsection, we consider several implementation issues and suggest their possible solutions. These considerations are described as follows:

- **Message encryption/decryption:** Application-level messages are embedded in the MESSAGE packets, which are encrypted before being sent. The message encryption and decryption are based on Triple-DES-CBC [19]. In most cases, the latest group key is used for data encryption. However, if communication happens during a rekeying operation, the group key formed in the previous rekeying interval is used instead. If the SIGNATURE mode is activated, the approach “signature before encryption” should be adopted [19], as shown in Fig. 7.4.
- **Key confirmation:** Key confirmation [2] refers that every member assures other members actually obtain the group key. Providing complete key confirmation incurs high communication cost since it requires all members to demonstrate their knowledge of the group key to other members. In our implementation, we adopt a *weaker* key confirmation approach, in which we designate a sponsor to broadcast the blinded group key (i.e., the blinded key of the root of the key tree) which lets other members verify if their computed blinded group key is identical to the one they receive.
- **Robustness:** It is possible that group members may leave the group or encounter system failures during a rekeying operation. If one of those members is a sponsor and fails to broadcast the necessary blinded keys, the group key cannot be computed. To resolve the problem, our implementation requires the leader to broadcast a rekeying message for a new rekeying operation if a sponsor leaves the group and the blinded group key for the current rekeying operation is not received, that is, the group

key is not yet confirmed. The new rekeying operation should reflect the leave event of the departed sponsor. It should be noted that the renewed nodes that are supposed to be broadcast by the departed sponsor in the previous rekeying operation remain renewed in the new one since they are on the key path of the departed sponsor and the blinded keys of those renewed nodes will be broadcast by other sponsors. This so-called *self-stabilizing* property, which is discussed in [11], is realized in our implementation.

- **Defense mechanisms:** We identify several possible adversary attacks that can happen in our implementation. They include: (1) joining the communication group without valid certificates; (2) corrupting signed messages; (3) pretending to be the leader; and (4) disrupting rekeying operations via various means, say, via broadcasting forged blinded keys or replaying signed blinded keys from previous rekeying operations. Our implementation combats these attacks respectively through the following: (1) checking the existence of the certificate of every joining member; (2) verifying signatures; (3) checking if the claimed leader joins the group later than some members; and (4) validating the group key via key confirmation. If attacks do exist, warning messages are displayed, and if attacks are (2) to (4), the system aborts.

7.3 SGCL API

SGCL comprises a number of API functions that enables developers to implement the interval-based algorithms in their secure group-oriented applications. The operations of the API functions rely on an *SGCL session object*, which holds all the components constituting the system architecture of SGCL. Details of the API functions are described in Table 7.2.

Fig. 7.7 presents the flowchart of using the SGCL API in a typical secure

Functions	Synopsis	Return values
<code>SGCL_init()</code>	It creates and initializes an SGCL session object for subsequent SGCL operations.	An initialized session object on success and NULL on failure
<code>SGCL_set_passwd()</code>	It sets the password, critical for accessing the long-term private key, inside the SGCL session object. It takes no effect if member authentication (i.e., the SIGNATURE mode) is disabled.	1 on success and 0 on failure
<code>SGCL_join()</code>	It connects to the Spread daemon and joins the specified communication group. It also initializes all components inside the SGCL session object.	1 on success and 0 on failure
<code>SGCL_send()</code>	It encrypts the application messages with the current group key and sends them to the communication group.	1 on success and 0 on failure
<code>SGCL_recv()</code>	It receives the application messages from the communication group and decrypts them with the current group key.	1 on success and 0 on failure
<code>SGCL_read_membership()</code>	It reports the current group membership status including the existing members, the joining members and the leaving members.	1 on success and 0 on failure
<code>SGCL_leave()</code>	It disables any operations and frees the resources of all components inside the SGCL session object. It then leaves the communication group and disconnects from the Spread daemon.	1 on success and 0 on failure
<code>SGCL_destroy()</code>	It destroys the SGCL session object and free its resources.	1 on success and 0 on failure

Table 7.2: Description of the SGCL API functions.

group-oriented application. The flow is described as follows: (1) the application creates and initializes an SGCL session object with `SGCL_init()`; (2) it opens the file of the long-term private key, which should be password-protected, with `SGCL_set_passwd()`, provided that member authentication is enabled; (3) it requests to join the specified communication group with `SGCL_join()`; (4) it implements its application protocols with `SGCL_send()`, `SGCL_recv()` and `SGCL_read_membership()` in order to send messages, receive messages, and read membership status, respectively; (5) it leaves the group with `SGCL_leave()`; and (6) it either joins another or the same group with `SGCL_join()`, or destroys the SGCL session object with `SGCL_destroy()` and ends.

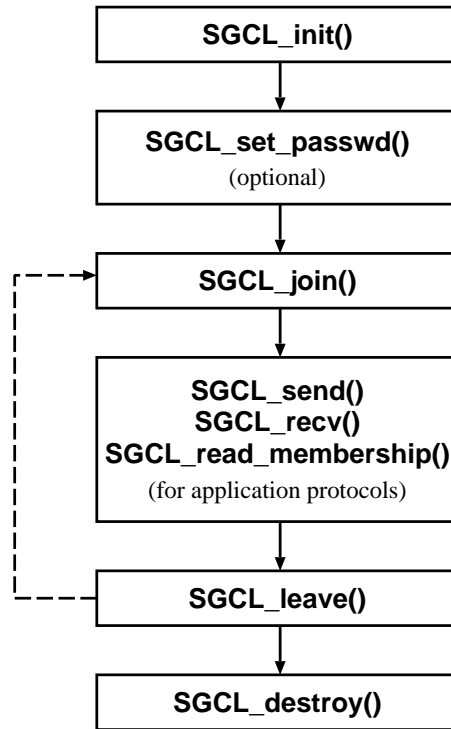


Figure 7.7: Flowchart of using the SGCL API.

7.4 Experiments

Motivated by the performance study in Chapter 5, we used *Gauger*, a performance testing tool developed with SGCL, to evaluate the performance of different interval-based algorithms under real network settings. In this section, we investigate two experiments, in which we are interested in the following metrics for a particular rekeying operation:

- *Rekeying time*: It measures the duration from starting a rekeying operation till confirming the correctness of the updated group key.
- *Number of exponentiations*: It measures the computation cost involving the exponentiation operations in the secret key and blinded key computations.
- *Number of broadcast blinded keys*: It measures the communication cost involving the number of blinded keys that are broadcast during a rekeying

operation.

- *Number of broadcast packets of blinded keys:* It measures the communication cost involving the number of broadcast packets of blinded keys during a rekeying operation. If the SIGNATURE mode is used, it also accounts for the computation cost in signing or verifying the broadcast packets. This metric is equivalent to the number of BKEY packets defined in Table 7.1. It should be noted each packet can contain more than one blinded key if they can be computed in one round.

The experiments were carried out under the following configurations. We fixed the group population to 40 Gauger applications, each of which corresponds to a group member that stays either inside or outside the same communication group for some time lengths. We assigned the group members evenly to eight Pentium 4/2.5GHz machines running Linux, that is, each machine had five Gauger applications installed. All eight machines were interconnected in a single local area network, so they were reachable from each other through broadcasts. A Spread daemon with configured parameters was running in each machine, and each Gauger application connected to the Spread daemon in the same local machine when it started execution. These configurations are assumed throughout the experiments.

The time lengths for which a Gauger application stays inside and outside the group are respectively given by $T_{in} + c$ and $T_{out} + c$, where T_{in} and T_{out} are exponentially distributed and c denotes a constant period. The reason that we add a constant to the time lengths is to avoid a particular member joining or leaving the communication group abruptly and hence we can guarantee a sufficient amount of time for the resources to be re-allocated between membership events. For accuracy, we pre-generated pattern files stating the occurrence times of join and leave events based on the distributions with various exponential averages of T_{in} and T_{out} , and then examined the performance of different

algorithms for a particular set of average parameters using the same pattern file.

The experiments assume several constant parameters. We let the rekeying interval, the regular period of performing rekeying operations, be 15 seconds, and the constant c , the minimum interval between two membership events for a group member, be 10 seconds. With the pre-generated pattern files, we ran the experiments for two hours, and then collected the recorded metrics for analysis.

Experiment 1: (Average analysis at different fixed T_{out} 's) This experiment evaluates the performance metrics of Rebuild, Batch and Queue-batch. We fixed T_{out} to be 30, 60, and 90 seconds, as well as varied T_{in} when conducting the experiment. Similar to the performance evaluation experiments in Chapter 5, the motivation of adopting fixed T_{out} 's is to control the rate that members join the communication group. After the experiment, we averaged the metrics over the number of existing members in the group at each rekeying interval.

The results are presented in Fig. 7.8. Among all T_{out} 's, we note that the metric costs are the largest at $T_{out} = 30$ seconds. It is because with a smaller T_{out} members tend to stay longer in the group and more members participate in a rekeying operation. This implies the key tree is bigger and therefore more operations are required to generate a group key. Besides, we observe Queue-batch outperforms the other two algorithms in all metrics. This shows the performance gain of Queue-batch in its dispersing the rekeying workload throughout the rekeying interval.

Experiment 2: (Average analysis of Batch and Queue-batch at different levels of membership dynamics) This experiment examines how the performance of the interval-based algorithms varies with respect to the frequencies of the join and leave occurrences. We only considered Batch and Queue-batch since they both demonstrate better performance than Rebuild.

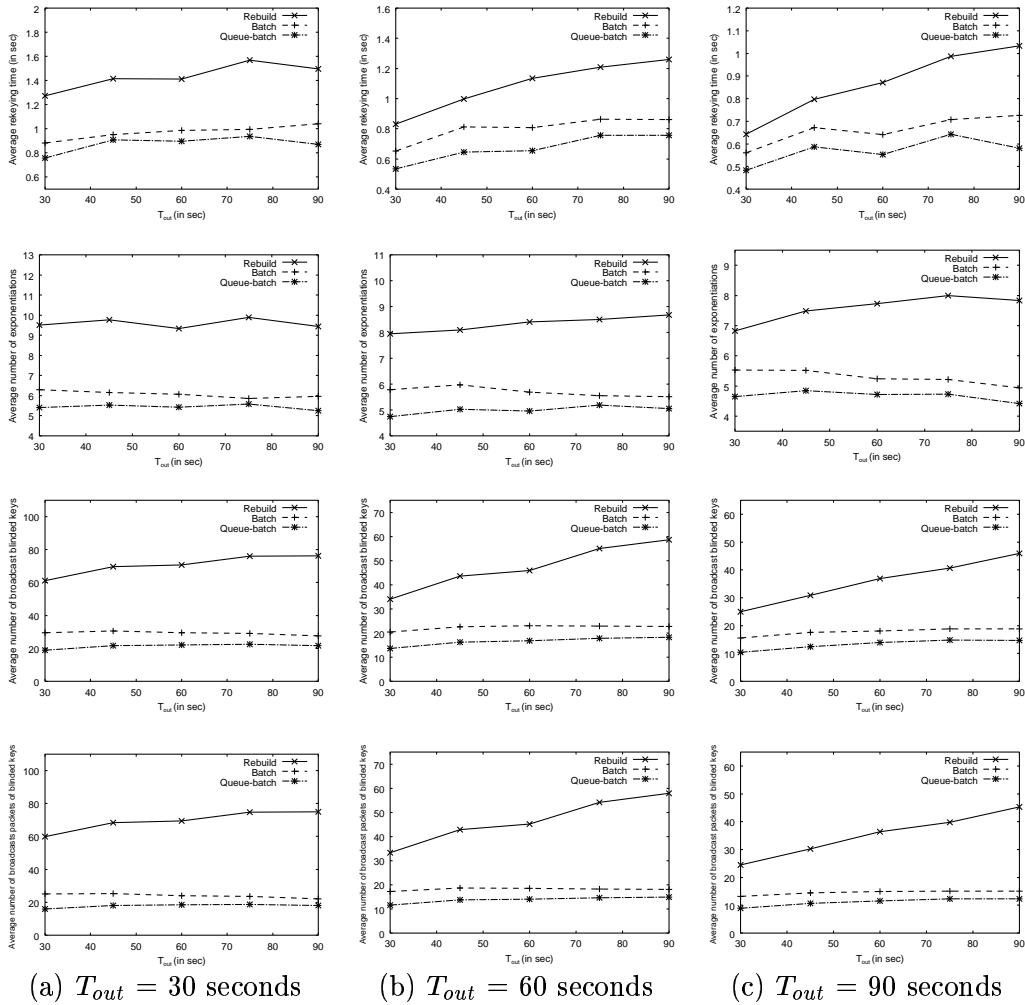
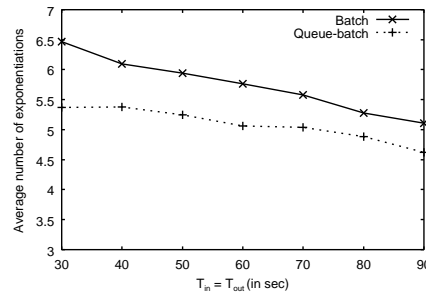


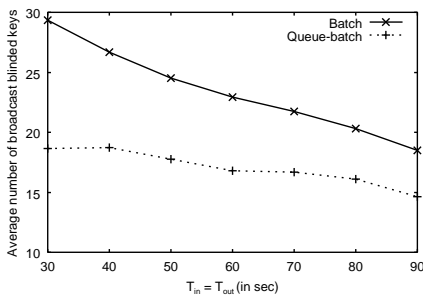
Figure 7.8: Average analysis at different fixed T_{out} 's.

Here, we set T_{in} equal to T_{out} , and changed the pair of T_{in} and T_{out} from 30 seconds to 90 seconds. Then we recorded the number of exponentiations, the number of broadcast blinded keys and the number of broadcast packets of blinded keys after the experiment. We did not consider the rekeying time since it is less stable and more machine-dependent compared to the other two metrics and results in a high deviation. Finally, we averaged the recorded metrics over the group size in each rekeying interval.

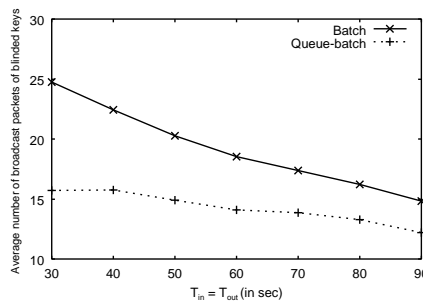
Fig. 7.9 illustrates the results. It shows that Queue-batch outperforms more than Batch when T_{in} and T_{out} are smaller. In other words, Queue-batch is more superior than Batch when the group is more dynamic. This conforms to the results presented in Chapter 5, which stated that the superior performance of Queue-batch becomes more obvious when the join and leave events occur more frequently and explained the reasons behind.



(a) Average number of exponentiations



(b) Average number of broadcast blinded keys



(c) Average number of broadcast packets of blinded keys

Figure 7.9: Average analysis at different levels of membership dynamics.

7.5 Applications

To demonstrate how SGCL can be realized in real-life applications, we deployed SGCL on a secure chat-room application called *Chatter*. Chatters allow individuals to participate in a communication group and to send communication messages securely to group members. The messages being sent undergo encryption with the group key and hence they are confidential against eavesdroppers when being transmitted over the network.

We implemented Chatter in both graphical and text modes, and its screenshots are illustrated in Figs. 7.10 and 7.11. The graphical mode lets users enjoy a colorful interface, but its implementation requires the presence of the X Window system in Linux. The text mode, therefore, is built to provide users with a text-based command console without the need of any graphical-compliant platform. Both interface modes are compatible and can be used together within the same communication group.

Apart from the chat-room applications, we envision that SGCL is feasible in a number of potential applications consisting of:

- **Audio/video conferencing systems:** Business parties may hold audio/video conferences with their laptop or desktop computers. The conferencing systems usually transfer massive streaming data among which there can be confidential business information. They can hence use SGCL to agree upon a secret group key to encrypt the streaming data.
- **File sharing tools:** File sharing is prevalent in peer-to-peer networks. Most shared files usually do not involve sensitive information, but some do. Therefore, if a file sharing application intends to distribute a private file to a group of users, SGCL will help protect the file data.
- **Router communication paradigms:** Routers may form a communication group in some situations. For instance, in defending against

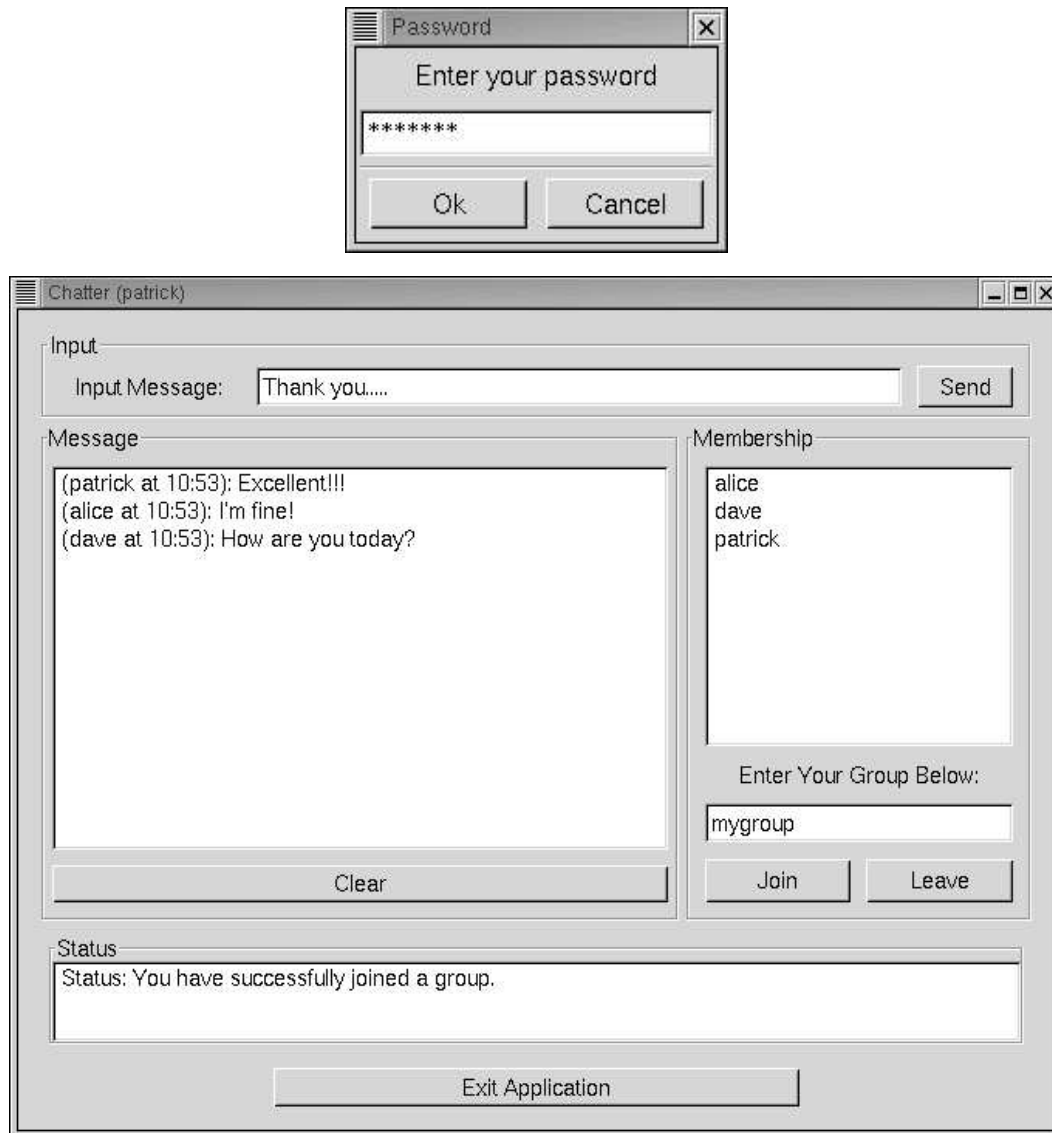
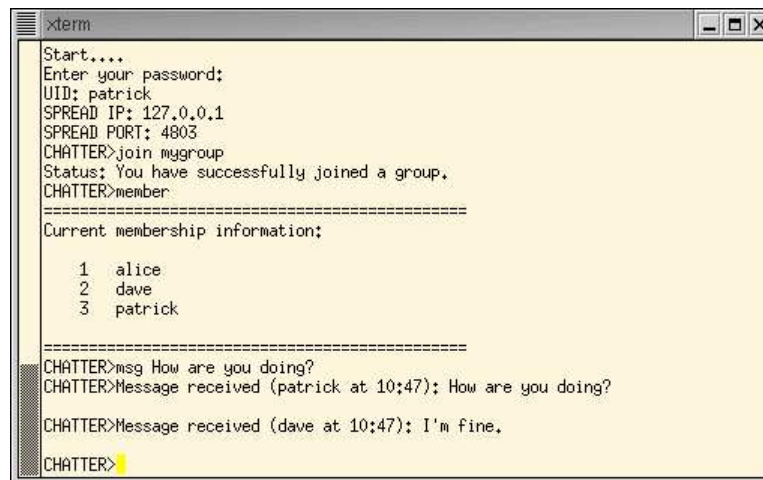


Figure 7.10: Illustration of Chatter in the graphical mode.



```
xterm
Start....
Enter your password:
UID: patrick
SPREAD IP: 127.0.0.1
SPREAD PORT: 4803
CHATTER>join mygroup
Status: You have successfully joined a group.
CHATTER>member
=====
Current membership information:
1  alice
2  dave
3  patrick
=====
CHATTER>msg How are you doing?
CHATTER>Message received (patrick at 10:47): How are you doing?
CHATTER>Message received (dave at 10:47): I'm fine.
CHATTER>
```

Figure 7.11: Illustration of Chatter in the text mode.

distributed denial-of-service (DDoS) attacks, routers may exchange information in order to pinpoint the location of attackers. Such exchanged information should be inaccessible to the attackers so that the detection succeeds. In this case, the exchanged information can be protected with SGCL.

- **Network games in strategy planning:** In network games, players may co-operate with each other in deciding the winning strategies over other competitors. This type of interaction involves numerous message exchanges. Thus, the games can use SGCL to encrypt the messages against any cheating attacks, such as eavesdropping and modification of the transmitted game data.

In short, SGCL implements the interval-based algorithms that achieves group key agreement without any centralized key server, and hence is adequate for any secure group-oriented applications in decentralized environments such as peer-to-peer networks or mobile ad hoc networks.

7.6 Future Extensions

Several enhancements can be made to enrich the current implementation. First, it is likely that we can achieve a higher degree of identity protection through A-TGDH (Authenticated Tree-Based Group Diffie-Hellman), the authenticated key agreement protocol presented in Chapter 6, by incorporating the long-term private key components of the group members into the group key. Second, our implementation aborts the operations when certain attacks are encountered. This approach, however, is not always desirable as it results in denial-of-service. Better recovery procedures are therefore needed to make the implementation more robust. Third, the current membership information is not protected in its transmission. In other words, when the Spread daemon reports the membership status to group members or other participating Spread daemons, the information is sent in plain and thus allows outsiders to recognize which are the existing members in a communication group. Although the exposure of such information may not be a critical security concern, it would be better to encrypt all data transmitted among Spread daemons and group members. Finally, we only explore the periodic interval-based approach for rekeying operations. Instead, we can switch to the *threshold-based* approach, for example, rekeying starts when the number of join and leave events exceeds a certain limit. Both approaches, in fact, can be incorporated into the system. Recall that a rekeying operation is sparked when the group leader broadcasts a rekeying message (represented as a REKEY packet in our implementation). We can define a set of policy rules stating when rekeying should start, such as after a fixed period or after the number of membership events reaches a threshold, and require the leader to issue the rekeying message to notify others to start rekeying when these conditions are met. We expect that putting these extensions into the implementation can make it more feasible for being used in practice.

Chapter 8

Conclusions and Future Directions

This chapter provides the conclusions for this dissertation and suggests some future directions for this research.

8.1 Conclusions

We considered several distributed collaborative key agreement protocols for dynamic peer groups. The key agreement setting is carried out such that there is no centralized key server maintaining and distributing the group key. We show that one can use the TGDH protocol to achieve such distributed and collaborative key agreement. To reduce the rekeying complexity, we proposed to use the interval-based distributed rekeying algorithms that allow us to group multiple join and leave requests and to process them at the same time. In particular, we showed that the Queue-batch algorithm can significantly reduce both computation and communication costs. This reduction can lead to a more efficient way to manage secure group communication. We also proposed an authenticated group key agreement scheme which offers protection against various attacks, as well as implemented SGCL to study the interval-based algorithms in a real network environment and to support the development of

secure group-oriented applications for dynamic peer groups.

8.2 Future Directions

We explore future directions that may enrich this research. We focus on two areas: *constructing a hybrid key tree with the physical and logical properties* and *extending the implementation*.

8.2.1 Construction of a Hybrid Key Tree with the Physical and Logical Properties

The key tree approach has been adopted by a number of centralized and decentralized group key management schemes [26, 27, 17, 20, 28, 11, 14, 12, 29], as well as our interval-based algorithms. Its intuitive idea is to maintain a key tree hierarchy for a number of keys that finally constitute the group key. During a rekeying operation, the key tree and hence the group key are updated in response to the dynamic join and leave events. The major advantage of using this approach is that the key tree itself is logical, and hence the rekeying operations do not rely on the support of intermediate routers and network devices while attaining a good rekeying efficiency.

In updating the key tree, all the proposed tree-based group key management schemes emphasize to preserve a balanced tree, yet do not take account of the underlying physical proximity. However, the rekeying performance may not be optimized in some circumstances. Consider Fig. 8.1 (a), which illustrates a wide area network (WAN)-based communication group dispersed in four different local area networks (LANs). Those LANs are assumed to be interconnected with low-bandwidth links. According to the classic rekeying approach, members can be located randomly under the key tree as long as the key tree is balanced. This implies that the sponsors are distributed in

different local area networks and have to exchange new blinded keys across the low-bandwidth links in each round. We therefore expect that those links will suffer heavy traffic load caused by a lot of blinded key transfers.

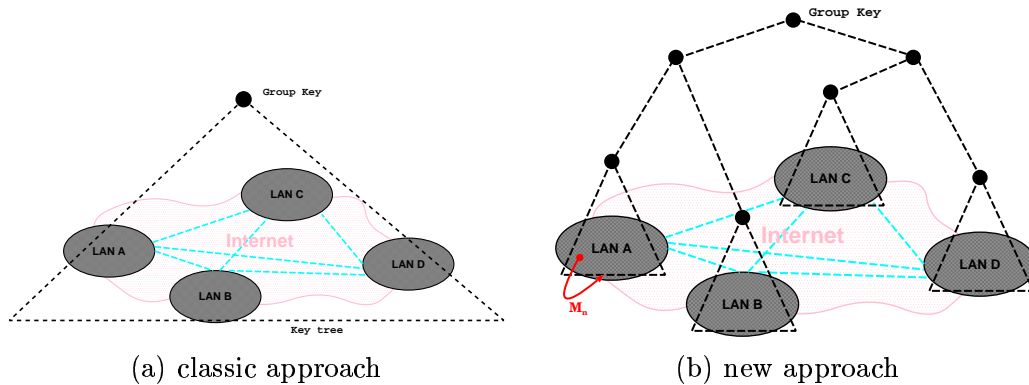


Figure 8.1: Approaches of updating the key tree.

To remedy the overhead in the low-bandwidth links, we change the arrangement of the key tree as follows. A communication group first divides itself into *clusters*, each of which is composed of members that are close together or that are interconnected with high-bandwidth links. In each cluster, the members can use a classic rekeying algorithm to form a subtree, as balanced as possible, and agree upon a subgroup key. The members then merge the subtree to form the key tree and finally use the subgroup keys to compute the resultant group key. When a new member joins the group, its associated leaf node should be put under the subtree of the cluster to which it belongs. Fig. 8.1(b) illustrates the new approach. In the figure, members in the same LAN are grouped to form a cluster. In other words, there are four clusters, each of which corresponds to a single LAN. The members in the same LAN first construct a subtree and obtain the subgroup key based on a classic rekeying algorithm. They then create the key tree with other clusters and obtain the final group key. When a new member, say M_n , joins the group, it should be put under the subtree in LAN A and the rekeying operation is then performed. With the new approach, we cut the loads of the low-bandwidth links by deferring the

blinded key transfers over them to the last few rounds. Hence we can achieve a better rekeying performance by combining the physical and logical properties in the update of the key tree.

To effectively realize both physical and logical properties, we need to resolve several questions: (1) “How many clusters are needed?”; (2) “How many members a cluster should contain?”; (3) “Can we define clusters with formal models?”; and (4) “What network environments are adequate for this hybrid rekeying approach?”. These questions can help derive a thorough solution regarding this problem.

8.2.2 Extended Implementation

In terms of implementation, we expect that there is room for expansion. In Section 7.6, we drafted a number of extensions aiming to improve the security of the implementation. Besides, it would be interesting to study the implementation performance in a different network environment, such as a wireless ad hoc network where the resources are constrained in the bandwidth of communication links and the computation power of mobile handsets. With the implementation extensions, we can have a better comprehension about the distributed and collaborative group key agreement protocols for a dynamic peer group.

Bibliography

- [1] Y. Amir, Y. Kim, C. Nita-Rotaru, and G. Tsudik. On the performance of group key agreement protocols. In *Proceedings of the 22th IEEE International Conference on Distributed Computing Systems*, Vienna, Austria, July 2002.
- [2] G. Ateniese, M. Steiner, and G. Tsudik. Authenticated group key agreement and friends. In *Proc. of 5th ACM Conference on Computer and Communications Security*, pages 17–26, November 1998.
- [3] S. Blake-Wilson and A. Menezes. Authenticated diffie-hellman key agreement protocols. In *Proc. of the 5th Annual Workshop on Selected Areas in Cryptography*, pages 339–361, 1998.
- [4] M. Burmester and Y. Desmedt. A secure and efficient conference key distribution system. *Advances in Cryptology – EUROCRYPT '94*, 950:275–286, 1995.
- [5] W. Diffie and M. Hellman. New directions in cryptography. *IEEE Transactions on Information Theory*, IT-22(6):644–654, 1976.
- [6] W. Diffie, P. C. van Oorschot, and M. J. Wiener. Authentication and authenticated key exchanges. *Designs, Codes and Cryptography*, 2(2):107–125, June 1992.

- [7] A. Fekete, N. Lynch, and A. Shvartsman. Specifying and using a partitionable group communication service. In *ACM PODC'97*, pages 53–62, August 1997.
- [8] C. G. Günther. An identity-based key exchange protocol. In *EUROCRYPT '89*, 1989.
- [9] ITU-T Recommendation X.509. Information Technology–Open Systems Interconnection–The Directory: Authentication Framework, November 1993.
- [10] M. Just and S. Vaudenay. Authenticated multi-party key agreement. In *Advances in Cryptology ASIACRYPT '96*, pages 36–49. LNCS 1163, Springer-Verlag, 1996.
- [11] Y. Kim, A. Perrig, and G. Tsudik. Simple and fault-tolerant key agreement for dynamic collaborative groups. In *Proc. of 7th ACM Conference on Computer and Communications Security*, pages 235–244, November 2000.
- [12] Y. Kim, A. Perrig, and G. Tsudik. Communication-efficient group key agreement. *Information Systems Security, Proceedings of the 17th International Information Security Conference IFIP SEC'01*, November 2001.
- [13] Y. Kim, A. Perrig, and G. Tsudik. Tree-based group key agreement. *In submission*, 2002.
- [14] X. S. Li, Y. R. Yang, M. G. Gouda, and S. S. Lam. Batch rekeying for secure group communications. In *Proceedings of Tenth International World Wide Web Conference (WWW10)*, Hong Kong, China, May 2001.
- [15] National Institute of Standards and Technology. The secure hash algorithm (SHA-1). NIST FIPS PUB 180-1, U.S. Department of Commerce, April 1995.

- [16] OpenSSL Project Team. OpenSSL: The Open Source toolkit for SSL/TLS. <http://www.openssl.org>.
- [17] A. Perrig. Efficient collaborative key management protocols for secure autonomous group communication. In *International Workshop on Cryptographic Techniques and E-Commerce (CrypTEC '99)*, pages 192–202, July 1999.
- [18] R. Rivest, A. Shamir, and L. Adleman. A method for obtaining digital signatures and public key cryptosystems. *Communications of the ACM*, February 1978.
- [19] B. Schneier. *Applied Cryptography*. Wiley, 1996.
- [20] S. Setia, S. Koussih, and S. Jajodia. Kronos: A scalable group re-keying approach for secure multicast. In *Proc. of IEEE Symposium on Security and Privacy 2000*, May 2000.
- [21] B. Song and K. Kim. Two-pass authenticated key agreement protocol with key confirmation. In *Proc. of Indocrypt2000*, volume LNCS vol. 1977, pages 237–249, December 2000.
- [22] Spread Concepts. The Spread Toolkit. <http://www.spread.org>.
- [23] W. Stallings. *Cryptography and Network Security: Principles and Practice*. Prentice Hall, 2nd edition, 1999.
- [24] M. Steiner, G. Tsudik, and M. Waidner. CLIQUES: A new approach to group key agreement. In *IEEE International Conference on Distributed Computing Systems*, pages 380–387, May 1998.
- [25] The Center for Networking and Distributed Systems (CNDS). The Secure Spread Project. http://www.cnds.jhu.edu/research/group/secure_spread.

- [26] D. M. Wallner, E. J. Harder, and R. C. Agee. Key management for multicast: Issues and architectures. Internet draft draft-wallner-key-arch-00.txt, Internet Engineering Task Force, July 1997.
- [27] C. K. Wong, M. Gouda, and S. S. Lam. Secure group communications using key graphs. *Proc. of ACM SIGCOMM'98*, September 1998.
- [28] C. K. Wong and S. S. Lam. Keystone: A group key management service. In *Proceedings International Conference on Telecommunications*, Acapulco, Mexico, May 2000.
- [29] Y. R. Yang, X. S. Li, X. B. Zhang, and S. S. Lam. Reliable group rekeying: A performance analysis. *Proc. of ACM SIGCOMM'01*, August 2001.