

Dynamic Balanced Key Tree Management for Secure Multicast Communications

Wee Hock Desmond Ng, Michael Howarth, Zhili Sun, and Haitham Cruickshank

Abstract—A secure multicast communication is important for applications such as pay-per-view and secure videoconferencing. A key tree approach has been proposed by other authors to distribute the multicast group key in such a way that the rekeying cost scales with the logarithm of the group size for a join or depart request. The efficiency of this key tree approach critically depends on whether the key tree remains balanced over time as members join or depart. In this paper, we present two Merging Algorithms suitable for batch join requests. To additionally handle batch depart requests, we extend these two algorithms to a Batch Balanced Algorithm. Simulation results show that our three algorithms not only maintain a balanced key tree, but their rekeying costs are lower compared with those of existing algorithms.

Index Terms—Multicast security, group key management, secure group communication.

1 INTRODUCTION

INTERNET Protocol (IP) multicast [1] allows a sender to transmit a single copy of some data, with network elements such as routers making copies as necessary for the receivers. This approach reduces sender-processing overhead and network bandwidth usage. This technology benefits many group communication applications [2], [3], [4], [5] such as pay-per-view, online teaching, and share quotes.

Before these group-oriented multicast applications can be successfully deployed, access control mechanisms must be developed such that only authorized members can access the group communication [6], [7], [8], [9], [10], [11], [12], [13], [14], [15], [16], [17]. The only way to ensure controlled access to data is to use a shared group key, known only to the authorized members, to encrypt the multicast data. As group membership might be dynamic, this group key has to be updated and redistributed securely to all authorized members whenever there is a change in the membership in order to provide forward and backward secrecy. Forward secrecy means that a departing member cannot obtain information about future group communication and backward secrecy means that a joining member cannot obtain information about past group communication. We assume the existence of a trusted entity, known as the Group Controller (GC), which is responsible for updating the group key. This allows the group membership to scale to large groups.

A number of scalable approaches have been proposed and one in particular, the key tree approach [10], [11], [12], [13], [14], is analyzed in detail in this paper. In short, the key tree approach employs a hierarchy of keys in which each

member is assigned a set of keys based on its location in the key tree. The rekeying cost of the key tree approach increases with the logarithm of the group size for a join or depart request. The operation for updating the group key is known as rekeying and the rekeying cost denotes the number of messages that need to be disseminated to the members in order for them to obtain the new group key.

Individual rekeying, that is, rekeying after each join or depart request, has two drawbacks [18]. First, it is inefficient since each rekey message has to be signed for authentication purposes and a high rate of join/depart requests may result in performance degradation because the signing operation is computationally expensive. Second, if the delay in a rekey message delivery is high or the rate of join/depart requests is high, a member may need a large amount of memory to temporarily store the rekey and data messages before they are decrypted. Batch rekeying [18], [19], [20], [21], [22], [23] has been proposed to alleviate these problems as a trade-off between performance and security. In this scheme, the GC does not perform rekeying immediately; instead, it consolidates the total number of joining and departing members during a time period before performing the rekeying. A short rekey interval does not provide much batch rekeying benefit, whereas a long rekey interval causes a delay to joining members and increases vulnerability from departing members who can still receive the data. A more detailed analysis of the trade-off among user dynamics, group size, and rekey interval has been presented in [19].

The efficiency of the key tree approach critically depends on whether the key tree is balanced [21], [24], [25], [26]. A key tree is considered balanced if the distance from the root to any two leaf nodes differs by not more than 1 [26]. For a balanced key tree with N members, the height from the root to any leaf node is $\log_k N$, where k is the outdegree of the key tree, but, if the key tree becomes unbalanced, then the distance from the root to a leaf node can become as high as N . In other words, this means that a member might need to perform $N - 1$ decryptions in order to get the group key. Furthermore, in an unbalanced key tree, some members

- W.H.D. Ng is with AT&T Singapore, Block 85 The Cavendish #03-05/08 Science Park Drive, Singapore 118261. E-mail: desmondng@gmail.com.
- M. Howarth, Z. Sun, and H. Cruickshank are with the Centre for Communication Systems Research, University of Surrey, Guildford, Surrey GU2 7XH, UK.

Manuscript received 6 Feb. 2005; revised 7 Dec. 2005; accepted 28 Aug. 2006; published online 13 Feb. 2007.

For information on obtaining reprints of this article, please send e-mail to: tc@computer.org, and reference IEEECS Log Number TC-0061-0205.
Digital Object Identifier no. 10.1109/TC.2007.1022.

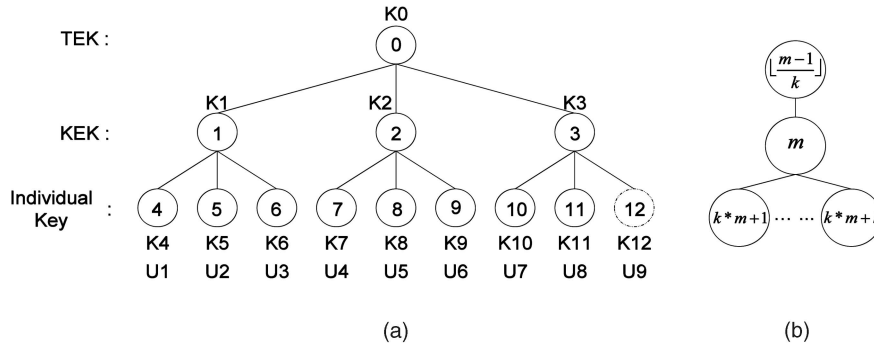


Fig. 1. (a) Logical key tree. (b) Node ID assignment.

might need to store N keys, whereas some might only need to store two keys.

In this paper, we propose two Merging Algorithms suitable for batch *join* events for combining subtrees together. These two Merging Algorithms not only balance the key tree, but have lower rekeying costs compared to existing algorithms. In other words, our Merging Algorithms allow all members in the multicast session to have similar storage and decryption requirements during each rekeying operation. Having a balanced key tree greatly benefits mobile devices since they generally have limited storage and computation power [4]; reducing the number of decryptions needed by the mobile devices can help to conserve the battery power. In order to additionally handle departing members, we extend these two Merging Algorithms to a Batch Balanced Algorithm where the tree height adapts to the change in the group membership. However, this requires a reorganization of the group members in the key tree. Simulation results show that our Batch Balanced Algorithm performs significantly better than existing algorithms when the number of joining members is greater than the number of departing members or when the number of departing members is around N/k , with no joining members. For similar numbers of joining and departing members, our Batch Balanced Algorithm achieves the same performance as that of existing algorithms. The rest of the paper is organized as follows: Section 2 covers the background and reviews existing work. We describe our proposed algorithms in Section 3. Section 4 describes our implementation and presents the simulation results. Section 5 discusses some of the application scenarios where our Batch Balanced Algorithm may be used. We present our conclusions in Section 6.

2 BACKGROUND

2.1 Key Tree Approach

In a typical key tree approach [10], [11], [14], as shown in Fig. 1a, there are three different types of keys: Traffic Encryption Key (TEK), Key Encryption Key (KEK), and individual key. The TEK is also known as the group key and is used to encrypt multicast data. To provide a scalable rekeying, the key tree approach makes use of KEKs so that the rekeying cost increases logarithmically with the group size for a join or depart request. An individual key serves the same function as KEK, except that it is shared only by the GC and an individual member. In the example in Fig. 1a, K0 is the TEK, K1 to K3 are the KEKs, and K4 to K12 are the individual

keys. The keys that a group member needs to store are based on its location in the key tree; in other words, each member needs to store $1 + \log_k N$ keys when the key tree is balanced. For example, in Fig. 1a, member U1 knows K0, K1, and K4 and member U7 knows K0, K3, and K10. The GC needs to store all of the keys in the key tree.

To uniquely identify each key, the GC assigns an ID to each node in the key tree. The assignment of the ID is based on a top-down and left-right order. The root has the lowest ID, which is 0. For a node with an ID of m , its parent node has an ID of $\lfloor (m-1)/k \rfloor$, with its children's IDs ranging from $km+1$ to $km+k$, as shown in Fig. 1b.

When a member is removed from the group, the GC must change all the keys in the path from this member's leaf node to the root to achieve forward secrecy. All the members that remain in the group must update their keys accordingly. If the key tree is balanced, the rekeying cost for a single departing member is $k \log_k(N) - 1$ messages. For example, suppose member U9 is departing in Fig. 1a. Then, all the keys that it stores (K0 and K3) must be changed, except for its individual key. Let $\{x\}y$ denote key x encrypted with key y and x' denote the new version of key x . Then, the GC needs to multicast the rekey messages $\{K3'\}K10$, $\{K3'\}K11$, $\{K0'\}K1$, $\{K0'\}K2$, and $\{K0'\}K3'$ to the members, giving a total of five encrypted keys.

If backward secrecy is required, then a join operation is similar to a depart operation in that the keys that the joining member receives must be different from the keys previously used in the group. The rekeying cost for a single joining member is $2 \log_k N$ messages when the key tree is balanced. Suppose member U9 is joining the group. Then, the GC needs to multicast the following rekey messages to the members: $\{K3'\}K3$, $\{K3'\}K12$, $\{K0'\}K0$, and $\{K0'\}K3'$.

The efficiency of the key tree approach critically depends on whether the key tree remains balanced. For a balanced key tree with N leaf nodes, the height from the root to the any leaf node is $\log_k N$. However, if the key tree becomes unbalanced, the distance from the root to a leaf node can become as high as N . Fig. 2 shows an unbalanced key tree. First of all, we can see that key storage among the group members varies from 3 to 6 rather than 4 in a balanced binary key tree of eight members. Second, U1 or U2 needs five decryptions if its sibling departs rather than three decryptions in a balanced key tree. Last, the rekeying cost is 9 when U1 or U2 departs since K0, K1, K3, K7, and K11 need to be changed. For a balanced key tree, the rekeying cost for a departing member is only 5. In this example, the difference between a balanced and an unbalanced key tree

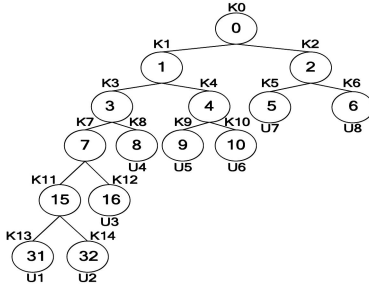


Fig. 2. An unbalanced key tree.

varies slightly as the group size is small. In a scenario such as pay-per-view, where the group membership varies from thousands to millions of members, an unbalanced key tree might lead to significant computation efforts for both the GC and group members.

2.2 Batch Rekeying

Batch rekeying is when join/depart requests are collected during a time interval called the rekey interval and are rekeyed together. This approach is used in [18], [19], [20], [21], [22], [23]. This not only alleviates the individual rekeying problems described in Section 1 (that is, inefficiency, rekey message delay, and high join/depart rates), but also reduces the number of group rekey events. Furthermore, the number of rekey messages that need to be multicast to the group can be much smaller than the number of rekey messages that would be generated if each membership change were processed individually due to the overlapping in paths from the leaf nodes to the root.

2.3 Related Work

Before we proceed further, we introduce some notations and definitions used in this paper. We use the term *ST* to indicate a subtree. We use “minimum height” to mean the minimum number of levels in a tree or subtree from the root to any leaf node. Similarly, we use “maximum height” to mean the maximum number of levels in a tree or subtree from the root to any leaf node. We define the following variables:

D	Number of departing members
J	Number of joining members
h	Height of key tree ($1 + \log_k N$)
H_{MIN}	Minimum height of leaf node in key tree
H_{MAX}	Maximum height of leaf node in key tree
H_{INSERT}	H_{MIN} of ST_A - H_{MAX} of ST_B
$H_{MIN_ST_X}$	H_{MIN} of ST_X
$H_{MAX_ST_X}$	H_{MAX} of ST_X

Marking Algorithms have been proposed to update the key tree and generate, at the end of each rekey interval, a rekey subtree with a collection of join and depart requests. Several variations of Marking Algorithms have been proposed [18], [20].

We refer to the algorithm in [18] as Marking Algorithm 1. For this algorithm, there are four cases to consider. If $J = D$, then all departing members are replaced by the joining members. If $J < D$, then we pick the J shallowest leaf nodes

from the departing members and replace them with the joining members. By the term “shallowest node,” we mean the leaf node of minimum height in our terminology. If $J > D$ and $D = 0$, then the shallowest leaf node is selected and removed. This leaf node and the joining members form a new key tree that is then inserted at the old location of the shallowest leaf node. Next, if $J > D$ and $D > 0$, then all departing members are replaced by the joining members. The shallowest leaf node is selected from these replacements and removed from the key tree. This leaf node and the extra joining members form a new key tree that is then inserted at the old location of the removed leaf node. Last, the GC generates the necessary keys and distributes them to the members.

The algorithm in [20] is referred to here as Marking Algorithm 2. There are only three cases to consider for this Marking Algorithm. Two of them, $J = D$ and $J < D$, are similar to the one mentioned above, except that the nodes of departing members that are not replaced by the joining members are marked as null nodes. For $J > D$, all departing members are replaced by the joining members. If there are null leaf nodes in the key tree, then they are also replaced by the joining members, starting from the null nodes with the smallest node ID. If there are still extra joining members, then the member with the smallest node ID is removed and it is inserted as a child, together with $k - 1$ joining members at its old location. The next smallest node ID member is selected if there are more joining members. This insertion continues until all of the joining members have been inserted into the key tree. As before, the GC distributes the new key to the members.

Balanced Batch Logical Key Hierarchy (LKH) [21], [22] has also been proposed to alleviate the inefficiency in Marking Algorithm 1 [18], but this algorithm is only suitable for a binary key tree ($k = 2$) and the author does not offer a solution for a key tree with other outdegrees.

3 BATCH REKEYING ALGORITHM

We now propose two Merging Algorithms [27] to combine subtrees together in a way that is suitable for batch join events. To handle all cases such as depart or both join and depart requests, we then extend these two Merging Algorithms into a Batch Balanced Algorithm. The two Merging Algorithms are used to combine two subtrees: ST_A and ST_B. We assume that ST_A has a greater height than ST_B and both subtrees are of the same outdegree k .

3.1 Merging Algorithm 1

This algorithm is only used when the difference in the maximum height between the two subtrees ST_A and ST_B is greater than or equal to 1. We will see why it is so in Section 4.

We now describe Merging Algorithm 1 [27] and illustrate it with some examples. The criteria for choosing Merging Algorithm 1 are when the difference between $H_{MAX_ST_A}$ and $H_{MIN_ST_B}$ is greater than 1 and when the difference between $H_{MAX_ST_A}$ and $H_{MAX_ST_B}$ is greater than or equal to 1. If both of these conditions are fulfilled, then the algorithm calculates H_{INSERT} , as illustrated in Fig. 3. The following steps are then performed:

Step 1. For $k > 2$, the algorithm searches for an empty child node in ST_A at either level H_{INSERT} or level $H_{INSERT} - 1$. If $H_{INSERT} = 0$, then levels 0 and 1 are searched. If such a

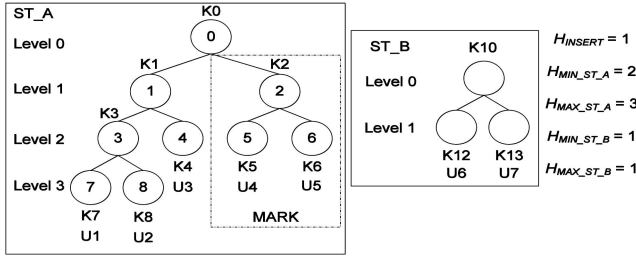


Fig. 3. ST_A (five members) and ST_B (two members).

node exists, then the algorithm inserts ST_B as the child of that particular key node.

Step 2. If an empty node is not found in Step 1, mark a suitable key node in ST_A at level H_{INSERT} for insertion as follows: If $H_{INSERT} = 0$, then a suitable key node at level 1 is marked. The marked key node is given by the one with the greatest number of leaf nodes at level $H_{MIN_ST_A}$. For example, in Fig. 3, there are two key nodes at level 1 ($= H_{INSERT}$). Of these, key node ID 2 is marked for insertion since it has two leaf nodes at level $H_{MIN_ST_A}$, whereas key node ID 1 has only one.

Step 3. For $k > 2$, when an empty node is not found in Step 1, the algorithm searches the root of ST_B for an empty node. If this exists, then the algorithm inserts the marked key node from Step 2 as the child of ST_B and inserts ST_B at the old location of the marked key node.

Step 4. For $k = 2$ or $k > 2$, if Steps 1 to 3 have not inserted ST_B into ST_A, then the algorithm creates a new key node at the old location of the marked key node (from Step 2) and inserts the marked key node and ST_B as its children.

Finally, the GC may need to multicast at most one update message to inform the affected members. We will discuss the update message later in this section. After updating the affected nodes, the members can identify the set of keys they need in the rekey messages.

Fig. 4 shows how we can combine the two subtrees by using Step 1 in Merging Algorithm 1. Fig. 4a shows only a part of ST_A. Suppose there are three joining members and no departing members that are formed into a new subtree ST_B, as shown in Fig. 4b. Since there is an empty node at level $H_{INSERT} - 1$ in ST_A (node ID 13), ST_B is inserted at that particular node. The resulting key tree is shown in Fig. 4c. In this example, no update message is needed.

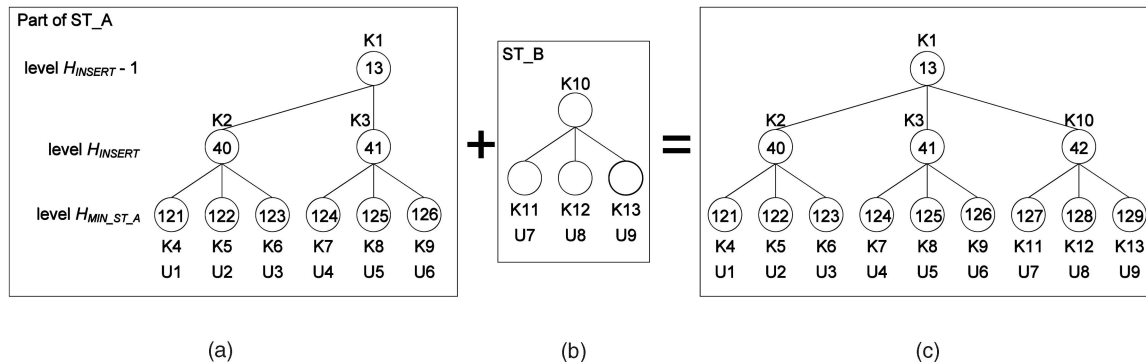


Fig. 4. (a) Part of the subtree in ST_A. (b) ST_B subtree (three members). (c) Resulting key tree.

Fig. 5 shows another example, where $k = 2$, and we therefore need to apply only Steps 2 and 4. We have a balanced subtree (ST_A) with eight members. Assume that two members wish to join the group. These two members form a new subtree (ST_B), as shown in Fig. 5b. For these subtrees, $H_{INSERT} = 2$. In Step 2, we perform our marking. Since all four key nodes at level 2 ($= H_{INSERT}$) have the same number of leaf nodes at level 3 ($= H_{MIN_ST_A}$), we arbitrarily choose key node 3 and mark it as shown in Fig. 5a. Finally, in Step 4, we create a new key node K18 and insert the marked key node and the new subtree, consisting of the new joining members, as its children, as shown in Fig. 5c. In this case, the GC needs to inform U1 and U2 that node ID 3 has shifted down one level to accommodate one new node. The update message therefore consists of the old node ID, which is 3, and the new node ID, which is 7.

The worst rekeying cost is $k(\lfloor \log_k N \rfloor - \lfloor \log_k J \rfloor + 1) + k[(J - 1)/(k - 1)]$ (see Appendix A.1).

Finally, there is an exception in Merging Algorithm 1. In cases where Step 4 is used to find the insertion point, $H_{MAX_ST_A} - H_{MIN_ST_A} = 1$ and $H_{MAX_ST_B} - H_{MIN_ST_B} = 1$ and, when there is at least one leaf node at level $H_{MAX_ST_A}$ in the marked key node, a slightly unbalanced key tree will exist if ST_A and ST_B are combined using the steps above. One way to alleviate this issue is to delete the root in ST_B and split it into at most k subtrees. Order these new subtrees in increasing order of H_{MIN} . Starting from the subtree with the minimum height, insert it into ST_A by using Merging Algorithm 1 and repeat until all new subtrees have been inserted into ST_A.

Fig. 6 shows an example. In this example, we delete the root in ST_B (K9) and use Step 4 twice to insert the two new subtrees. Member U8 is inserted as the sibling of member U3 (see MARK1) and both members U6 and U7 are inserted as the siblings of members U4 and U5 (see MARK2). In this case, the rekeying cost is slightly higher and two update messages are needed.

3.2 Merging Algorithm 2

We now describe our Merging Algorithm 2 [27]. This algorithm is only used for combining subtrees whose height difference is 0 or equal to 1.

The criteria for using Merging Algorithm 2 are when the difference between $H_{MAX_ST_A}$ and both $H_{MIN_ST_B}$ and

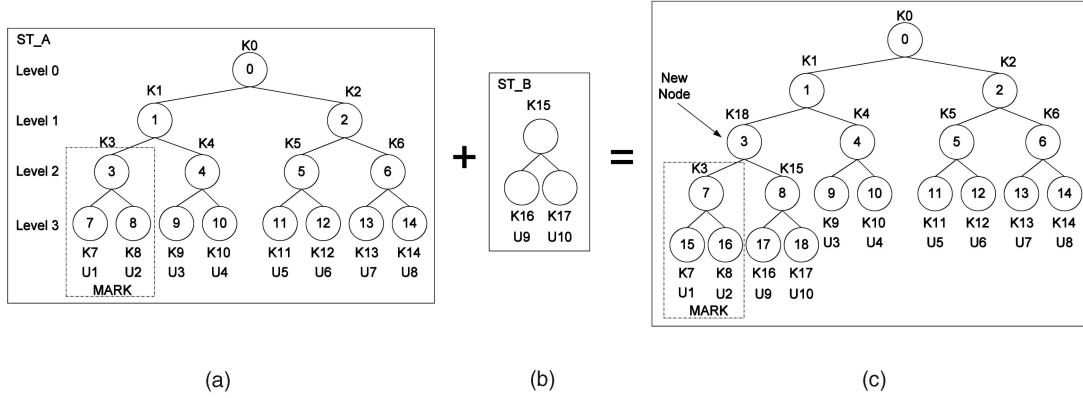


Fig. 5. (a) ST_A subtree (eight members). (b) ST_B subtree (two members). (c) Resulting key tree.

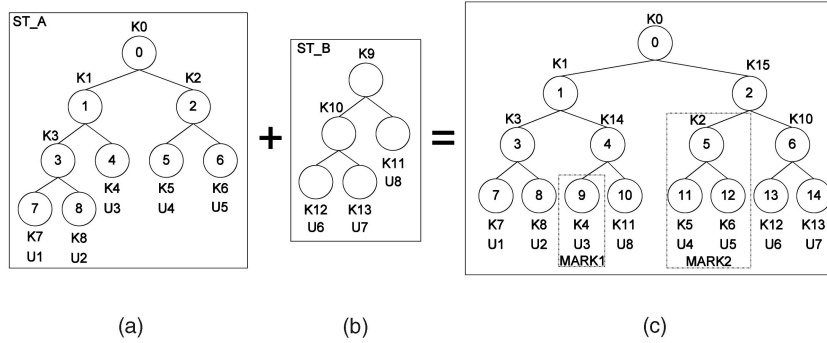


Fig. 6. (a) ST_A subtree (five members). (b) ST_B subtree (three members). (c) Resulting key tree.

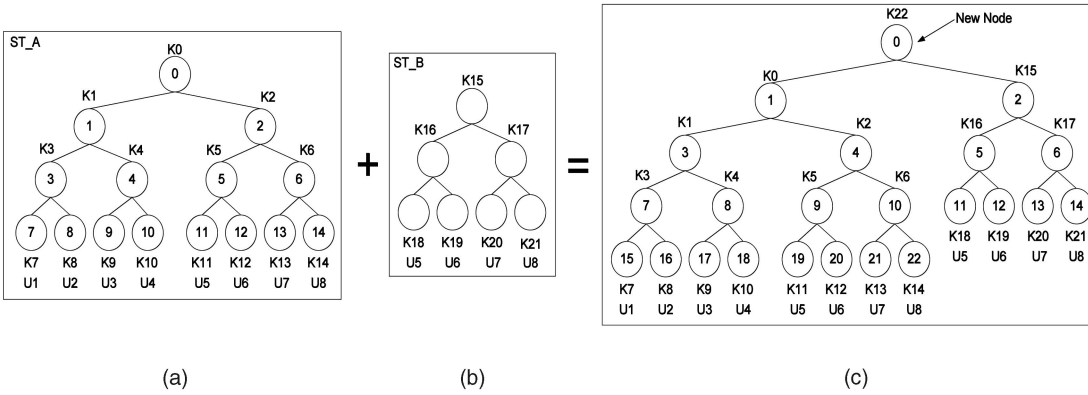


Fig. 7. (a) ST_A subtree (eight members). (b) ST_B subtree (four members). (c) Resulting key tree.

$H_{MAX_ST_B}$ is 0 or equal to 1. The algorithm performs the following steps:

Step 1. For $k > 2$, the algorithm searches the root of ST_A for an empty child key node. If it exists, then the algorithm inserts ST_B at the empty child key node.

Step 2. For $k = 2$ or when Step 1 is not valid for $k > 2$, the algorithm creates a new key node at the root and inserts ST_A and ST_B as its children.

The GC needs to multicast at most one update message to all existing members. After updating the affected node IDs, the members can identify the set of keys that they need in the rekey messages.

Fig. 7 shows an example of how Merging Algorithm 2 is used to combine two subtrees together by using Step 2. Since Merging Algorithm 2 creates a new node at the root,

the GC needs to inform U1 to U8 that the ID of the old root has changed from 0 to 1 by using the update message.

As shown in Appendix A.1, the rekeying cost is $k + k[(J - 1)/(k - 1)]$ messages.

3.3 Batch Balanced Algorithm

We now show how our two Merging Algorithms can be extended to produce an algorithm that we call Batch Balanced Algorithm that encompasses both joining and departing members.

There are six steps in our Batch Balanced Algorithm.

1. Identify and mark all key nodes that need to be updated. These key nodes are on the ancestor paths from each departing member to the root.

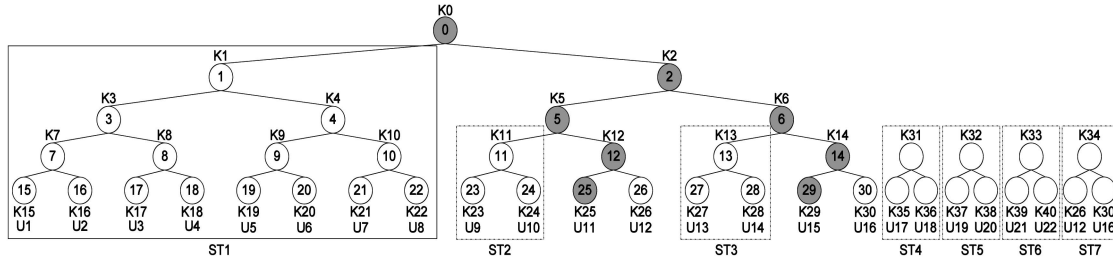


Fig. 8. Steps 1 to 4 of the Batch Balanced Algorithm.

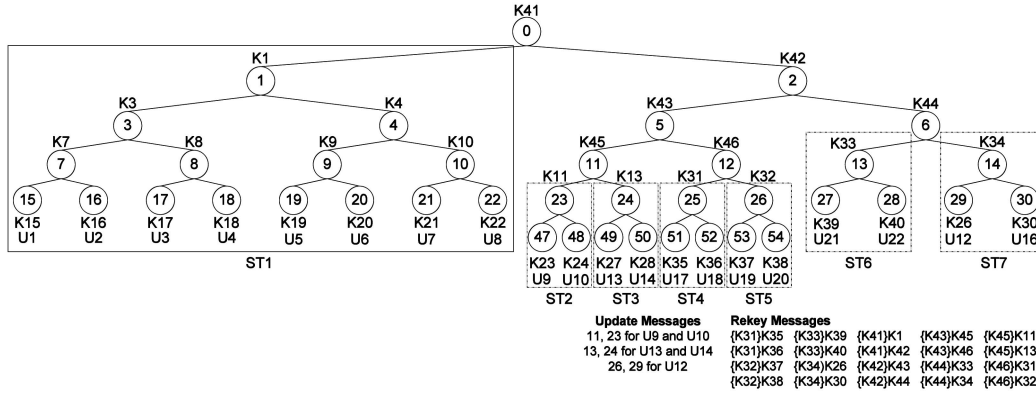


Fig. 9. Resulting key tree.

2. Remove all marked key nodes. After removal, there are only two types of element left: the remaining subtrees and the joining members.
3. Classify all siblings of the departing members as joining members since all of the KEKs that they store cannot be used.
4. Group the joining members into one or many subtrees, each with k members. If there are remaining members left, then they are grouped into another subtree of between 2 and $k - 1$ members unless there is only one member left. If there is only one member left, then treat it as a single-node subtree.
5. Starting from the subtree with the minimum height, compare it with another subtree with the next minimum height and if the Merging Algorithm 1 criteria are met, combine them using Merging Algorithm 1, else combine them using Merging Algorithm 2. Repeat this process until there is only one key tree.
6. Construct the update and rekey messages and multicast them to the members.

For clarity, we illustrate it with an example. Assume that we have a key tree with 16 members. Suppose members U11 and U15 are departing from the group and six new members, U17 to U22, are joining the group. All of the key nodes in the path from the departing members to the root are marked and removed (Steps 1 and 2). The siblings of departing members U12 and U16 form a new subtree, ST7, since the KEKs that they store are unusable (Step 3). The joining members form one or more subtrees of k members (Step 4). These usable subtrees ST1 to ST7 are identified as shown in Fig. 8.

In Step 5, we start with the minimum-height subtrees and merge them. Thus, ST2 forms a subtree with ST3, ST4 forms a subtree with ST5, and ST6 forms a subtree with ST7.

Then, the resulting subtree of ST2 and ST3 is combined with the resulting subtree of ST4 and ST5. This resulting subtree, in turn, forms another subtree with the resulting subtree of ST6 and ST7. Finally, the last two subtrees form a single key tree, as shown in Fig. 9. The GC sends out the update messages to inform the members of their new location. Those members that need to receive the update messages are U12 and the members in ST2 and ST3, which means that a total of three update messages is needed. In this example, we assume that member U16 and subtree ST1 are left intact at their old location. If their locations are changed, then two extra update messages are needed. For ST4, ST5, and ST6, no update message is needed since the members in the subtrees are newly joining members. At the same time, the GC can multicast the rekey messages to the members. The total rekeying cost is 20 messages.

If we use Marking Algorithm 1 [18] or Marking Algorithm 2 [20] in a similar situation, then Marking Algorithm 1 has the same rekeying cost, but it ends up with an unbalanced key tree. Although Marking Algorithm 2 can maintain a balanced key tree, it needs 28 rekey messages. From this, we can see that reorganizing the group members leads to saving on rekeying costs.

3.4 Update Messages

In order for the members to identify the keys that they need after the key tree has been reorganized, the GC needs to inform the members of their new location. An update message consists of the smallest node ID of the usable key tree m and the new node ID m' . With the new node ID m' , the members can update the remaining keys m_0 by using the following function:

$$f(m_0) = k^x(m' - m) + m_0, \quad (1)$$

where x denotes the level of the usable key tree.

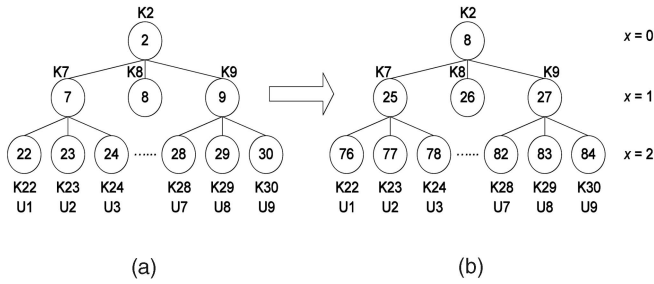


Fig. 10. (a) Usable and (b) new updated key trees.

For example, in Fig. 10, the smallest node ID in the usable key tree m is 2 and the smallest new node ID m' is 8. Each member just needs to insert the remaining node ID m_0 that they store into the function above to obtain the new node ID.

4 PERFORMANCE EVALUATION

In this section, we study the performance of our proposed algorithms and compare them with the Marking Algorithms described in [18] and [20], which we have already labeled as Marking Algorithm 1 and Marking Algorithm 2, respectively. We consider four performance metrics:

- rekeying cost,
- update cost,
- minimum and maximum height in the key tree, and
- key storage.

The rekeying cost denotes the total number of rekey messages that need to be sent to all authorized group members in order for them to learn the new group key. A higher rekeying cost means that more bandwidth is needed for the transmission. Although Marking Algorithm 2 adopts the User-Oriented Key Assignment Algorithm (UKA) [20], where all of the encryptions for a member are assigned in a single packet, we ignore the UKA when we calculate the rekeying costs since it leads to a significant number of duplications in rekey messages. Instead, we just calculate the total number of rekey messages that are needed without any duplication. The update cost denotes the total number of update messages that need to be sent to all affected members after the key tree has been reorganized in order for them to identify the keys that they need. As for the minimum and maximum height, they affect the members' key storage and, thus, the number of decryptions needed by each member and may even increase the rekeying costs, too, as explained in Section 2. Last, the key storage denotes the number of keys each member need to store.

We ran our algorithms on a Linux terminal with a 512 Mbyte RAM on a 2 GHz processor. To give an indication of runtime, for a tree size of 4,096 members, runtimes are typically in the range of 1 to 5 sec and, for a tree size of 65,536 members, runtimes are typically in the range of 1 to 40 sec, both results being less than or equal to approximately 2,000 departing and joining members.

4.1 Merging Algorithm Performance Evaluation

We have performed some simulations to compare the performance of both of our Merging Algorithms with existing work for batch join requests. For our simulations, we used a

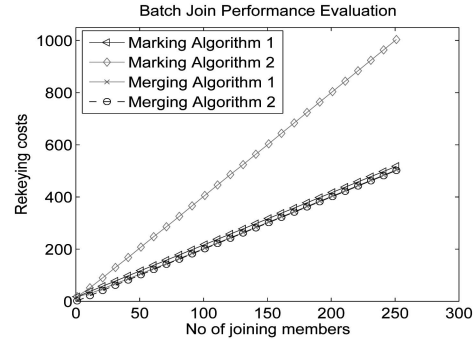


Fig. 11. Batch join rekeying costs.

balanced binary key tree of 256 members with a height of 8. The number of joining members varies from 1 to 250.

4.1.1 Rekeying Cost

In Fig. 11, we can see that Marking Algorithm 2 has the highest rekeying cost. This is because the joining members are inserted one by one at each leaf node, which affects the paths from the affected leaf nodes to the root. As the number of joining members increases, the number of affected nodes increases significantly. On the other hand, the other three algorithms have similar rekeying costs since they try to minimize the number of affected nodes. Marking Algorithm 1 minimizes the rekeying costs by placing the new subtree, which consists of joining members and one removed member on the shallowest height, at the old location of the removed member. Hence, only the path from that leaf node to the root is affected, regardless of the number of joining members. In other words, the rekeying cost consists of the rekey messages that need to be multicast to the joining members and $2 \log_k N$ messages to update the keys from that affected leaf node to the root. Merging Algorithm 1 inserts the new subtree consisting of the joining members into one of the key nodes in the key tree at a location that depends on the number of the joining members; thus, as the number of joining members increases, the number of affected nodes is reduced since the key node selected for insertion gets closer to the root. For Merging Algorithm 2, a new root is created with the existing subtree and the new subtree consisting of the joining members, which are inserted as its children.

4.1.2 Update Cost

Of the four algorithms, only Marking Algorithm 2 does not need to distribute update messages to the members. Marking Algorithm 1 needs to send one update message to inform the removed leaf node of its new location. Similarly, both Merging Algorithms need to send out one update message to inform the affected members of the newly created node.

4.1.3 Minimum and Maximum Height

Fig. 12 shows the maximum height of the key tree after the joining members have been inserted into the key tree for all algorithms. Only Marking Algorithm 2 and Merging Algorithm 2 maintain at a fixed height, regardless of the number of joining members. Marking Algorithm 2 alleviates the inefficiency of Marking Algorithm 1 by inserting the joining members one by one at each leaf node, whereas

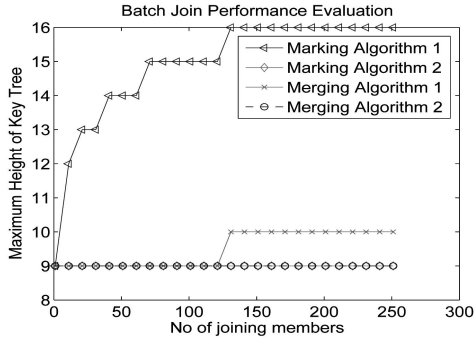


Fig. 12. Maximum height of the key tree.

Merging Algorithm 2 creates a new root and inserts the existing key tree and the joining member key tree as its children. Merging Algorithm 1 has the same performance as Marking Algorithm 2 and Merging Algorithm 2 when the number of joining members is less than or equal to half the group size. However, once the number of joining members exceeds half the group size, the maximum height increases by 1. This is why we set the criteria in Section 3.1, because the selection of the suitable key node is always at the child of the root once the joining members are greater than half the group size. In the case of Marking Algorithm 1, the maximum height increases significantly as the number of joining members increases because all of the joining members form a new subtree with one member at the minimum height. This new tree is inserted at the old location of the removed member, causing the maximum height to increase considerably.

Fig. 13 shows the maximum difference in height of the key tree, which indicates whether the key tree is balanced. The maximum difference in height for Marking Algorithm 1 increases considerably as the number of joining members increases. Similarly, our Merging Algorithm 2 is not a balanced key tree when the number of joining members is less than half the group size and it only maintains a balanced key tree when the number of joining members is greater than or equal to half the group size. This is why we set the criteria for Merging Algorithm 2 in Section 3.2. As for our Merging Algorithm 1, it maintains a balanced key tree when the number of joining members is less than or equal to half the group size. The difference in height in Merging Algorithm 1 increases by 1 once the number of joining members exceeds half the group size since the child of the root is selected for the insertion. Marking Algorithm 2 is the only algorithm that creates a balanced key tree, regardless of the number of joining members. However, this comes with the drawback of the high rekeying costs, as shown in Fig. 11. On the other hand, if we can choose

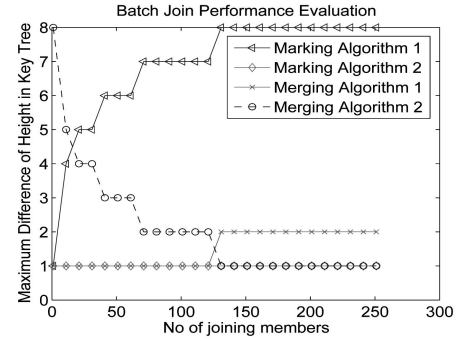


Fig. 13. Maximum difference in height.

appropriately between both of our Merging Algorithms depending on the number of joining members, we can create a balanced key tree without extra costs.

4.1.4 Key Storage

Table 1 shows the minimum and maximum number of keys that a member needs to store for the four algorithms for batch join events. We can see that the maximum number of keys that a joining member needs to store in Marking Algorithm 1 is dependent on the number of joining members at that particular interval. A large number of joining members results in a great difference in key storage among members. Marking Algorithm 2 does not suffer from the storage inefficiency as in Marking Algorithm 1, but it comes at the expense of the large rekeying costs, as shown in Fig. 11. Our Merging Algorithms can achieve the same efficiency of Marking Algorithm 2 if the Merging Algorithm is chosen appropriately, depending on the number of joining members.

4.2 Batch Balanced Algorithm

4.2.1 Rekeying Cost

We have performed a theoretical analysis for the rekeying cost of our Batch Balanced Algorithm, the details of which are discussed in Appendix A. To verify our analysis, we have also built a simulator for the algorithm. The simulator first constructs a balanced key tree with 1,024 members for $k = 2$. Departing members are either randomly selected or selected so as to give either the best or worst rekeying costs. Joining members are then inserted into the key tree and the rekeying costs are calculated.

Figs. 14a and 14b show the computed and simulated best and worst rekeying costs for a binary key tree. Our analysis and simulated results match so well that we could not distinguish between the two. For the best case, the rekeying

TABLE 1
Minimum and Maximum Key Storage for Batch Join Events

	Marking Algorithm 1	Marking Algorithm 2	Merging Algorithm 1	Merging Algorithm 2
Min key storage	$\lfloor \log_k(N) \rfloor$	$\lfloor \log_k(N + J) \rfloor$	$\lfloor \log_k(N + J) \rfloor$	$\lfloor \log_k(N + J) \rfloor$
Max key storage	$\lceil \log_k(N) + \log_k(J + 1) \rceil$	$\lceil \log_k(N + J) \rceil$	$\lceil \log_k(N + J) \rceil$	$\lceil \log_k(N + J) \rceil$

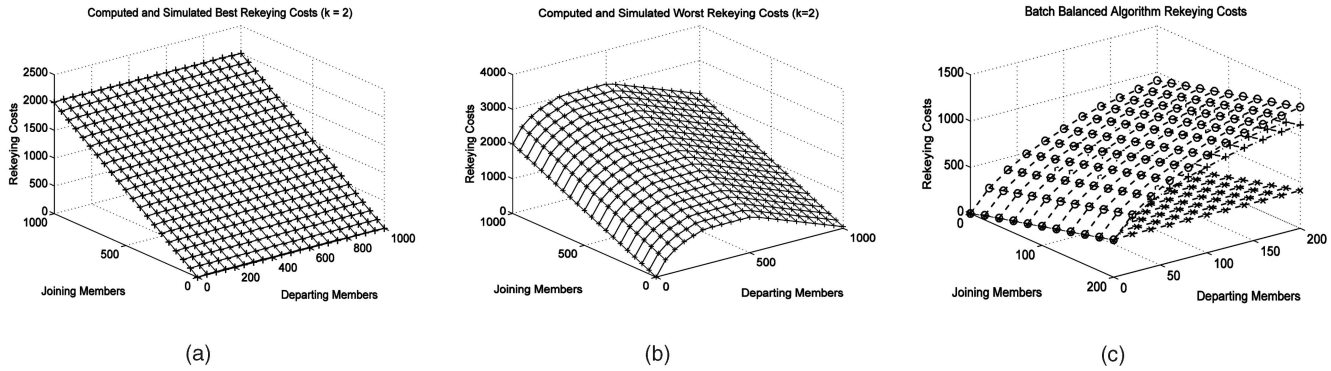


Fig. 14. (a) Best case. (b) Worst case. (c) Best, average, and worst rekeying costs for $k = 2$.

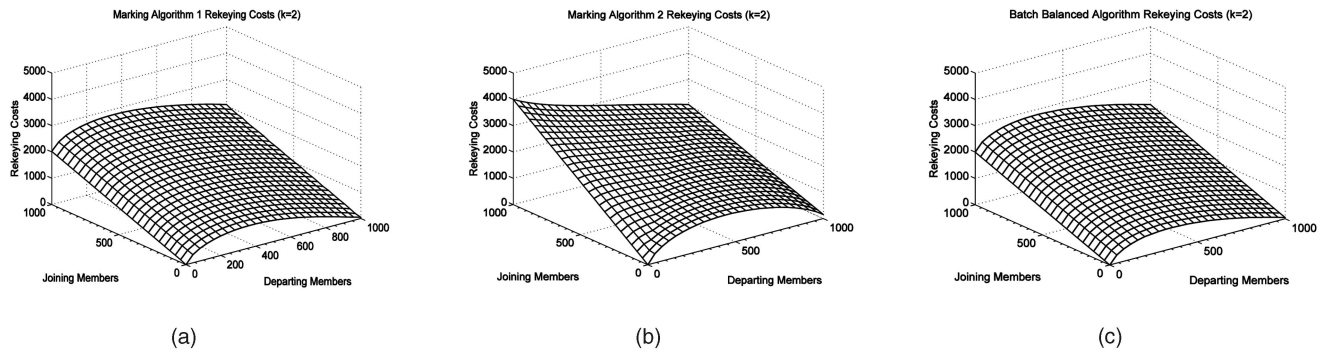


Fig. 15. Rekeying costs for (a) Marking Algorithm 1, (b) Marking Algorithm 2, and (c) Batch Balanced Algorithm.

costs are not affected by the number of departing members; rather, they are based purely on the number of joining members. This is because the number of affected nodes is minimized as the departing members are concentrated on one area of the key tree. On the other hand, if the departing members are spread fairly on the key tree, as in the worst case, then it maximizes the number of affected nodes in the key tree. The highest rekeying cost occurs when the number of departing members approaches half the group size, which means that most or all the key nodes in the key tree cannot be used.

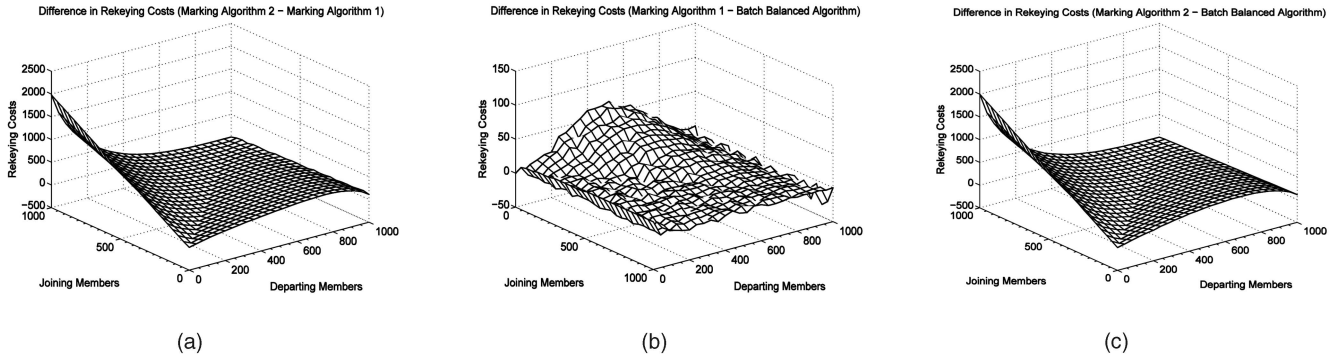
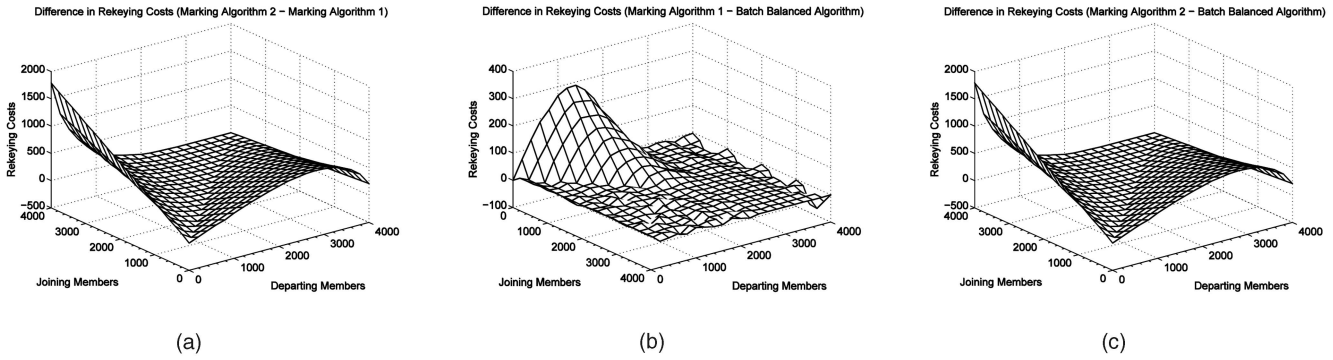
If the departing members are randomly selected, then we obtain the mean rekeying costs that lie between the theoretical best and worst cases. Fig. 14c shows the calculated best and worst and also the simulated average rekeying cost calculated as an average of 100 simulation runs. We can see that the average rekeying cost is closer to the worst rekeying cost than to the best rekeying cost. Generally, we can predict the rekeying costs for a key tree of any outdegree k if we are able to group the members according to their departing probability since it is based purely on the number of joining members rather than the number of departing members. However, if the departing members are spread around as in the worst case, the highest rekeying cost happens when the number of departing members is around N/k since most or all of the KEKs that the members store cannot be used.

We also built simulators for the Marking Algorithms [18], [20] to compare their performance with that of our Batch Balanced Algorithm. All simulators first construct a balanced key tree and then randomly pick departing members, with all members having an equal probability of departing. The joining members are inserted into the key tree and, finally, the rekeying costs are calculated. The

rekeying costs are again based on the average of 100 runs. We use a tree of 1,024 members with a height of 10 for a balanced binary key tree and a tree of 4,096 members with a height of 6 for a balanced key tree of $k = 4$.

Fig. 15 shows the rekeying costs for the three algorithms for $k = 2$. Marking Algorithm 1 and the Batch Balanced Algorithm have similar rekeying costs. Marking Algorithm 2 has twice the rekeying costs compared to both Marking Algorithm 1 and the Batch Balanced Algorithm when the number of joining members approaches the group size and there are no departing members. Generally, Marking Algorithm 2 has the highest rekeying costs when the number of joining members is greater than the number of departing members.

To investigate the rekeying costs in detail, we calculate the difference in the rekeying costs for the three algorithms, as shown in Fig. 16. For clarity, the y -axis of Fig. 16b is reversed. All three algorithms have similar rekeying costs when the number of joining members and the number of departing members are comparable. Marking Algorithm 1 has the highest rekeying costs compared to the Batch Balanced Algorithm when the number of departing members approaches half the group size, especially when there are no joining members. This is because most of the key nodes in the key tree are affected by the departing members. By reorganizing the remaining members in the key tree, we can reduce the rekeying costs. Replacing some of the departing members with the joining members can help to diminish this effect for both Marking Algorithm 1 and Marking Algorithm 2. We can see that Marking Algorithm 2 has the highest rekeying costs compared to Marking Algorithm 1 and the Batch Balanced Algorithm when the number of joining members is greater than the number of departing members. Another high cost in

Fig. 16. Difference in rekeying costs for $k = 2$.Fig. 17. Difference in the rekeying costs for $k = 4$.

Marking Algorithm 2 is when the number of departing members approaches half the group size. All of these extra rekeying costs resulted from the use of the null nodes and the way Marking Algorithm 2 inserts the joining members.

Fig. 17 shows the difference in the rekeying costs for $k = 4$. As before, the y -axis for the difference in rekeying between Marking Algorithm 1 and the Batch Balanced Algorithm is reversed for a clearer view. When the number of joining members and the number of departing members are comparable, all three algorithms have similar rekeying costs. As before, Marking Algorithm 2 has two regions of high rekeying costs compared to Marking Algorithm 1 and the Batch Balanced Algorithm. The first occurs when the number of joining members approaches the group size with no departing members. The other occurs when the number of departing members is around N/k with no joining members. Marking Algorithm 1 has the highest rekeying costs compared to the Batch Balanced Algorithm when the number of departing members is in the region of N/k with no joining members. Replacing the departing members with the joining members helps to lower the rekeying costs for both Marking Algorithm 1 and Marking Algorithm 2.

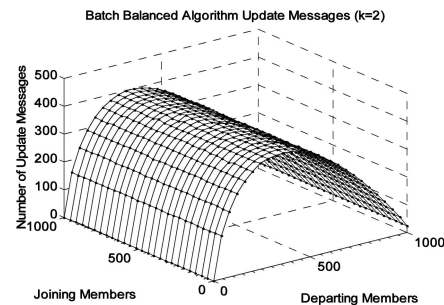
4.2.2 Update Cost

For the Batch Balanced Algorithm, there are some overheads incurred since we reorganize the group members in the key tree. This requires the GC to send update messages to inform the members of their new location. It is important to note that the GC in Marking Algorithm 1 needs to multicast update messages to the members as well.

Fig. 18 shows the total update messages that need to be sent to the remaining group members, including the siblings of the departing members, in order for them to update their new key node IDs. As expected, the update

messages are purely dependent on the number of departing members. The number of update messages increases as the number of departing members increases to around half the group size. This is because more key nodes in the key tree are affected by the departing members. However, once the number of departing members exceeds half the group size, the number of update messages decreases since there are fewer members left in the group.

If we assume that a key is 128 bits long and the node ID is 20 bits (that is, up to 2^{20} members), then a rekey message is at least 148 bits, excluding other overheads. An update message consists of the old node ID and the new node ID and, ignoring overheads, is therefore 40 bits long. In other words, a rekey message is 3.7 times the length of an update message; thus, the maximum update cost is equivalent to 109 rekey messages. In Section 5, we show how we can reduce the number of update messages needed by the group members for some rekey events.

Fig. 18. Update messages for the Batch Balanced Algorithm ($k = 2$).

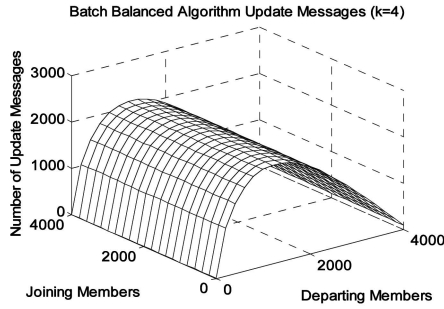


Fig. 19. Update messages for the Batch Balanced Algorithm ($k = 4$).

Fig. 19 shows the total number of update messages that need to be multicast to the members for $k = 4$. We can see that there is a sharp increase in update messages compared with a binary key tree. This is because, for every departing member, the GC needs to send three update messages to its siblings so that they can update the new location. The highest number of update messages occurs when the number of departing members is in the region of N/k .

4.2.3 Minimum and Maximum Height

Fig. 20 shows the minimum and maximum height for Marking Algorithm 1, showing that a small percentage of joining or departing members can increase the difference in height significantly. In the case where the number of joining members is greater than the number of departing members, only the maximum height is affected, whereas the minimum height is left unchanged and vice versa when the number of departing members is greater than the number of joining members. Marking Algorithm 1 can only maintain a balanced key tree when the number of joining members is equal to the number of departing members.

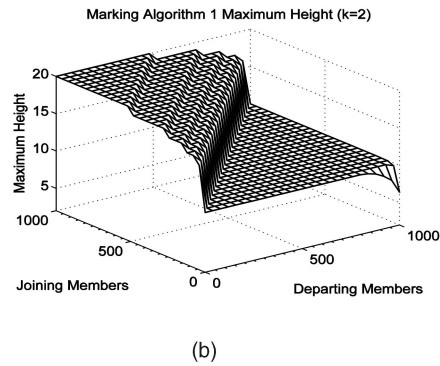
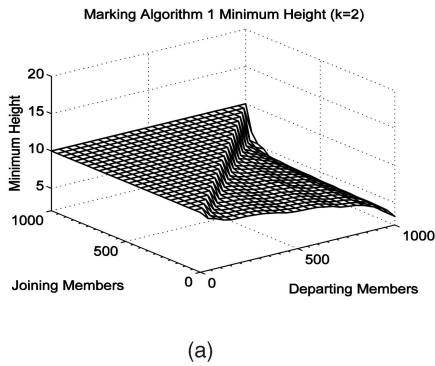


Fig. 20. (a) Minimum and (b) maximum height for Marking Algorithm 1.

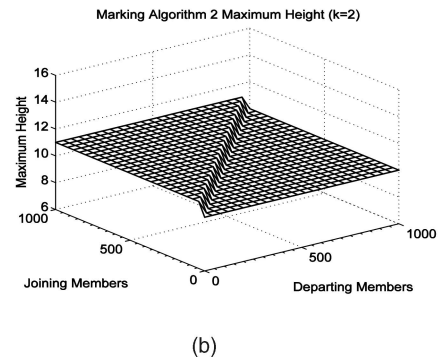
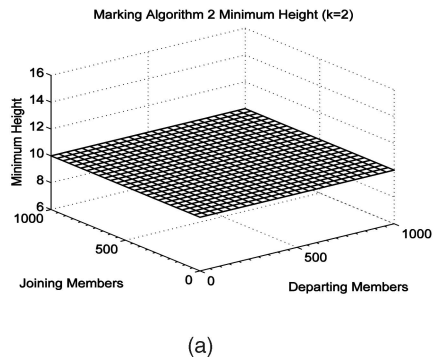


Fig. 21. (a) Minimum and (b) maximum height for Marking Algorithm 2.

Fig. 21 shows the minimum and maximum height for Marking Algorithm 2. It can be seen that the rapid increase in height as in Marking Algorithm 1 is not visible in this case since the joining members are inserted one by one at each leaf node. As for the minimum height, Marking Algorithm 2 alleviates the inefficiency in Marking Algorithm 1 with the use of the null node. However, one problem with Marking Algorithm 2 is that the key tree has now become a static key tree that can increase its minimum height to accommodate more joining members into the group but cannot decrease its minimum height with the departure of the members since nodes that are not occupied are marked as null nodes. This causes unnecessary key storage and encryptions or decryptions for both the GC and group members. There is no way to overcome this issue unless the whole key tree is rekeyed, which adds extra network costs.

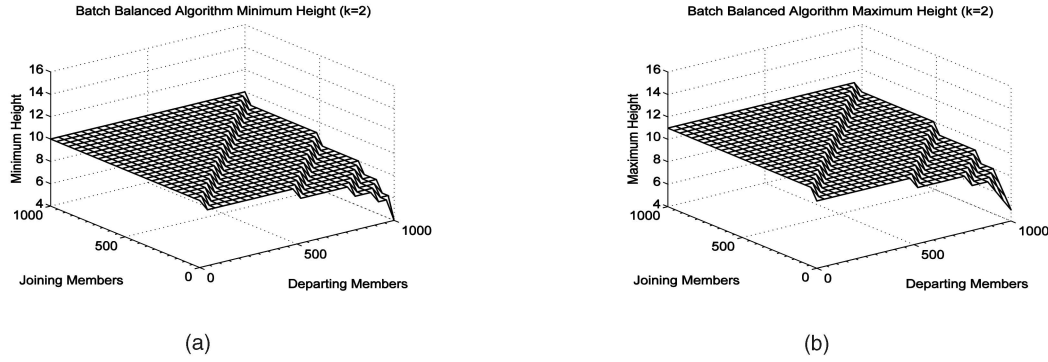
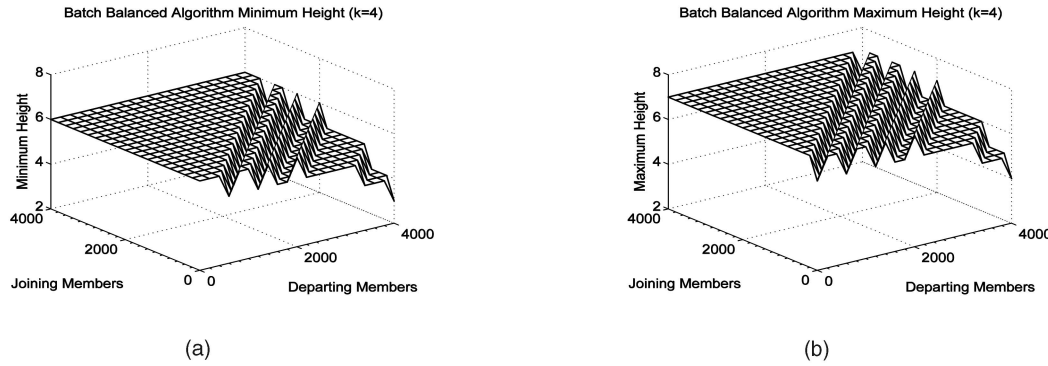
Fig. 22 shows the minimum and maximum heights for the Batch Balanced Algorithm. Regardless of the number of joining or departing numbers, both minimum and maximum height adapt to the changes in the group membership.

We omit the minimum and maximum heights performance for Marking Algorithm 1 and Marking Algorithm 2 for $k = 4$ since the analysis described above is applicable.

Fig. 23 shows the minimum and maximum heights for the Batch Balanced Algorithm for $k = 4$. Both the minimum and maximum height have similar output. Since the single key tree is formed using a bottom-up method, some of the root's child key nodes might be less than k in some case, which results in one extra level in the minimum and maximum height.

4.2.4 Key Storage

Table 2 shows the minimum and maximum key storage for the three algorithms. Since a full calculation for key storage

Fig. 22. (a) Minimum and (b) maximum height for the Batch Balanced Algorithm ($k = 2$).Fig. 23. (a) Minimum and (b) maximum height in the Batch Balanced Algorithm ($k = 4$).

of Marking Algorithm 1 is complex, we have assumed a balanced key tree when we calculate its maximum and minimum storage. The maximum key storage for a joining member in Marking Algorithm 1 then depends on the number of joining and departing members at that particular interval. For Marking Algorithm 2, the member needs additional key storage since the minimum and maximum key storage values are dependent on the maximum group size ever reached, N_{MAX} , regardless of the current group size. The Batch Balanced Algorithm has a lower maximum key storage than either Marking Algorithm 1 or Marking Algorithm 2.

5 DISCUSSION

5.1 Optimization

From the above simulations, we observe that the Batch Balanced Algorithm has identical rekeying costs compared to existing algorithms when the number of joining members and the number of departing members are comparable. Therefore, one optimization that we can apply to our Batch

Balanced Algorithm is not to reorganize the members in the key tree for the following condition:

$$D \leq J \leq (D - D_{min}) + kD_{min}, \quad (2)$$

where D_{min} is the number of departing members at the minimum height.

For the case where J is equal to D , we replace all D departs by J joins. If J is greater than D and provided that J is smaller or equal to $[(D - D_{min}) + kD_{min}]$, then we replace all $(D - D_{min})$ departs at the maximum height with $(D - D_{min})$ joins. The remaining joining members are split across the D_{min} nodes. We illustrate it with an example. Suppose, in Fig. 24a, U3, U6, and U7 depart and U8 to U12 join. Then, J must lie between 3 and 5 in order to fulfill the above condition. The resultant key tree is shown in Fig. 24b.

Fig. 25 shows the update messages for our optimized Batch Balanced Algorithm for $k = 2$. We can see that there are some cases where no update message is needed since there is no reorganization in the group. The rekeying costs still remain the same as shown in Fig. 15. There is no way to maintain a balanced key tree without reorganizing the key tree when the number of departing members is greater than the number of joining members.

5.2 Application Scenarios

Based on our simulation results, we now make some recommendations on the various applications where our Batch Balanced Algorithm can outperform existing work. Our recommendations are based on application requirement, operation environments, and expected group membership behavior.

TABLE 2
Minimum and Maximum Key Storage for
Batch Join and/or Depart Events

	Marking Algorithm 1	Marking Algorithm 2	Batch Balanced Algorithm
Min Key Storage	$\lfloor \log_k(N) \rfloor$	$\lfloor \log_k(N_{MAX}) \rfloor$	$\lfloor \log_k(N + J - D) \rfloor$
Max key storage	$\lfloor \log_k(N) \rfloor +$ $\lceil \log_k(J - D + 1) \rceil$	$\lceil \log_k(N_{MAX}) \rceil$	$\lceil \log_k(N + J - D) \rceil$

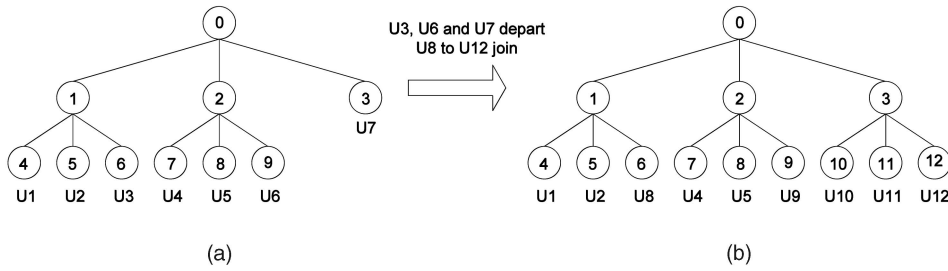


Fig. 24. (a) Key tree with join and depart requests. (b) Resulting key tree.

- *Satellite pay-per-view TV distribution.* For this type of scenario, generally, the number of joining members is greater than the number of departing members at the beginning of the session. During the session, the number of joining members and departing members might be comparable. Finally, at the end of the session, the number of departing members is greater than the number of joining members. In this case, our Batch Balanced Algorithm can overcome the inefficiency in Marking Algorithm 1 and Marking Algorithm 2 at the beginning and end of the session. In the middle of the session, the rekeying costs for the three algorithms are expected to be similar.
- *Group-oriented mobile commerce* [4], [5]. Generally, mobile devices have limited storage and computation [4], [27]. In this case, it might be better to minimize the number of keys that they need to store and the number of decryptions during each batch. By reducing the number of decryptions needed, we can help to conserve the battery power too. Furthermore, the energy consumption of decryption is 30 percent more than encryption [28]. In this type of scenario, the Batch Balanced Algorithm has better performance compared to both Marking Algorithms since its minimum and maximum height adapt to the changes in the group membership. In addition, its rekeying cost is lower than or similar to those of the existing algorithms. This is crucial, especially in wireless multicast scenario, since the bandwidth is limited and data typically experience a higher transmission error rate than in conventional environments [29].
- *Multicast session with long duration.* There is usually at least one peak period and one nonpeak period for this type of scenario. If Marking Algorithm 2 is used for this type of scenario, then additional rekeying costs are needed during the off-peak period since the minimum and maximum height of the key tree stay

at the same level as in the peak period. Furthermore, there are additional key storage and encryption/decryption requirements for both the GC and group members. As for Marking Algorithm 1, the probability of the key tree becoming unbalanced is very high [21], which can result in a higher decryption for some of the members.

6 CONCLUSION

In this paper, we have presented two Merging Algorithms that are suitable for batch join events. To additionally handle batch depart requests, we have extended these two Merging Algorithms into a Batch Balanced Algorithm. All three algorithms try to minimize the difference in height in the key tree without adding extra network costs. However, all of the algorithms require the GC to update the affected members on their node position by using update messages. By minimizing the differences in height, we minimize the number of key storage and decryptions needed by each member. This is critical for terminals with limited computation and storage. Furthermore, reducing the number of decryptions can help to reduce the energy consumption, which, in turn, leads to battery saving.

For batch join events, the way the joining members are inserted has a significant effect on the key tree, especially when there are a large number of join requests in a batch. The key tree can become unbalanced even if the insertion is at the minimum height. Existing algorithms do not simultaneously consider both the balancing of key tree and rekeying costs and therefore lead to either an unbalanced key tree or high rekeying costs. Our proposed Merging Algorithms provide a good compromise compared to existing algorithms, producing a balanced key tree with low rekeying costs. Merging Algorithm 1 requires the GC to multicast at most two update messages to inform the affected members, whereas Merging Algorithm 2 needs at most one update message. Compared with that of a rekey message, the size of an update message is several times smaller since it consists of only the old node ID and the new node ID.

As for other events, our Batch Balanced Algorithm outperforms existing algorithms when the number of joining members is greater than the number of departing members and when the number of departing members is around N/k with no joining member. However, our algorithm requires the GC to multicast update messages to the members. For cases where the number of joining members and the number of departing members are comparable, our Batch Balanced Algorithm has a similar performance compared to existing work. We further observe that, if we are able to group the members according

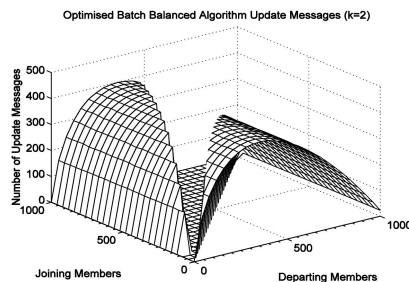


Fig. 25. Update message for the optimized Batch Balanced Algorithm ($k = 2$).

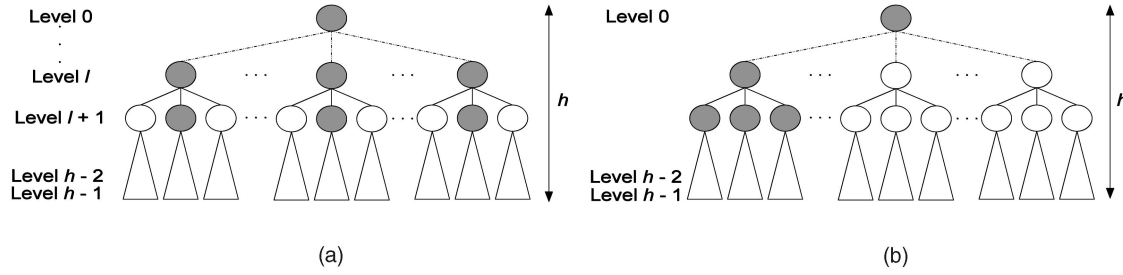


Fig. 26. (a) Worst and (b) best case rekeying costs.

to their departing probability, then we are able to predict the rekeying costs based on the number of joining members. However, if the departing members are spread evenly across the key tree, then the highest rekeying cost happens at around N/k since most or all of the KEKs that the members store cannot be used.

APPENDIX A

ANALYSIS

The performance of the key tree approach depends on storage, computation, and communication overhead. Following [10], [11], [12], [13], [14], we assume that the key tree investigated in this work is fully loaded and balanced, with height h . All members have equal probability of departing. For ease of analysis, we assume that all of the subtrees are combined using Merging Algorithm 2 when the number of departing members is greater than zero.

In order for the GC to perform the rekeying, it needs to store all of the keys in the key tree; therefore, its storage is $\lceil (kN - 1)/(k - 1) \rceil$ keys. The members, on the other hand, only need to store the keys on the ancestral path from their location in the key tree to the root, which means that each member needs to store $\log_k(N) + 1$ keys.

A.1 Merging Algorithm Analysis

When there is no departing member, we can immediately apply Merging Algorithm 1 or Merging Algorithm 2, depending on the number of joining members. The worst rekeying cost for Merging Algorithm 1 occurs when the two subtrees are combined using Step 4 since a new key node is created for insertion. All of the KEKs from the root to level H_{INSERT} are affected by the insertion; in other words, the number of affected key nodes, including the newly created key node, is $\lfloor \log_k N \rfloor - \lfloor \log_k J \rfloor + 1$. In addition, we include the cost for constructing a subtree for the joining members $k \lceil (J - 1)/(k - 1) \rceil$. Similarly, the worst rekeying cost for Merging Algorithm 2 occurs when the two subtrees are combined using Step 2 since a new root is created for insertion. The rekeying cost is simply the outdegree k for creating a new root plus the cost for constructing a new key tree for the joining member. Thus, the worst rekeying cost RC for a batch join event is given by

$$\begin{aligned} \text{Worst RC}_{(J \geq 0, D=0)} = & \\ \begin{cases} k(\lfloor \log_k N \rfloor - \lfloor \log_k J \rfloor + 1) & \text{Merging Algorithm 1} \\ + k \lceil \frac{(J-1)}{(k-1)} \rceil & \end{cases} & (3) \\ \begin{cases} k + k \lceil \frac{(J-1)}{(k-1)} \rceil & \text{Merging Algorithm 2.} \end{cases} \end{aligned}$$

In the best case for the two Merging Algorithms, a new node is not required and, therefore, the best rekeying cost is given by Best RC = Worst RC - k .

A.2 Batch Balanced Algorithm Worst Case Analysis

For simplicity, we first assume that the number of departing members is some value $D = k^l$ for some integer l .

In the worst case, the departing members are spread evenly at the leaf nodes, as shown in Fig. 26a. This means that $(k^{l+1} - 1)/(k - 1)$ key nodes from level 0 to level l cannot be used. From level $l + 1$ to $h - 2$, there are $D(k - 1)$ usable subtrees at each level; in other words, the total number of usable subtrees is $D(k - 1)(h - 2 - l)$. At level $h - 1$, there are $(k - 1)$ siblings for each departing member, which means a total of $D(k - 1)$ siblings. The worst update cost is $D(k - 1)(h - 2 - l) + D(k - 1)$. Using our Batch Balanced Algorithm, all J joining members and $D(k - 1)$ siblings of the departing members form one or more subtrees with k members. Finally, all subtrees form a single key tree. The rekeying cost is therefore given by

$$\begin{aligned} \text{Worst RC}_{(J \geq 0, D \neq 0)} &= \\ &= \frac{k(D(k - 1)(h - 2 - l) + \lceil \frac{D(k-1)+J}{k} \rceil - 1)}{k - 1} + D(k - 1) + J \\ &= kD(\log_k \frac{N}{D}) - D + J + \frac{k \lceil \frac{D(k-1)+J}{k} \rceil - k}{k - 1}. \end{aligned} \quad (4)$$

Now, suppose that we have $D = k^l + r$, where r lies between 0 and $(k - 1)k^l$. The analysis is split into two portions. For the k^l portion, the previous analysis still applies. As for r , $r[(h - 3 - l)(k - 1) - 1]$ usable subtrees are produced. The worst update cost is $k^l(k - 1)(h - 2 - l) + r(h - 3 - l)(k - 1) + (k^l + r)(k - 1)$. In addition, the $r(k - 1)$ siblings of the r departing members form one or more subtrees with the siblings of the k^l departing members and joining members. The rekeying cost therefore becomes

$$\begin{aligned} \text{Worst RC}_{(J \geq 0, D \neq 0)} &= \frac{k[k^l(k - 1)(h - 2 - l) + \lceil \frac{(k^l+r)(k-1)+J}{k} \rceil - 1 + r(h - 3 - l)(k - 1) - r]}{k - 1} + (k^l + r)(k - 1) + J \\ &= (k^{l+1} + rk) \log_k \frac{N}{k^l + r} + \frac{k \lceil \frac{(k^l+r)(k-1)+J}{k} \rceil - k - rk^2}{k - 1} - k^l - r + J. \end{aligned} \quad (5)$$

A.3 Batch Balanced Algorithm Best Case Analysis

Again, we first assume that $D = k^l$ for some integer l .

In the best case, the number of key nodes affected by the departing members is minimized, which means that all of the

```

if  $J > 0$  and  $D = 0$  then
  if  $(H_{MAX\_ST\_A} - H_{MIN\_ST\_B} > 1)$  and  $(H_{MAX\_ST\_A} - H_{MAX\_ST\_B} \geq 1)$  then
    if  $k > 2$  then
      if  $(H_{MIN\_ST\_A} - H_{MAX\_ST\_B} = 0)$  then
         $H_{INSERT} = 1$ 
      else
         $H_{INSERT} = H_{MIN\_ST\_A} - H_{MAX\_ST\_B}$ 
      end if
      if empty key node between  $H_{INSERT}$  and  $H_{INSERT} - 1$  in  $ST\_A$  then
        insert  $ST\_B$  as the child at empty key node
      else
        mark suitable key node at level  $H_{INSERT}$  in  $ST\_A$ 
        if empty key node at root of  $ST\_B$  then
          insert marked key node as the child of  $ST\_B$ 
        else
          create a new key node
          insert marked key node and  $ST\_B$  as its children
          insert new key node at old location of marked key node
        end if
      end if
    end if
  if  $k = 2$  then
    mark suitable key node at level  $H_{INSERT}$  in  $ST\_A$ 
    create a new key node
    insert marked key node and  $ST\_B$  as its children
    insert new key node at old location of marked key node
  end if
end if

```

```

if  $(H_{MIN\_ST\_A} - H_{MIN\_ST\_B} = 1)$  and  $(H_{MAX\_ST\_A} - H_{MAX\_ST\_B} = 1)$  and
  (marked key node has at least one leaf nodes at level  $H_{MAX\_ST\_A}$ ) then
  delete the root of  $ST\_B$  and split it into  $k$  subtrees
  Arrange these subtrees by increase order of  $H_{MIN}$ 
  while number of subtrees  $> 1$  then
    insert subtrees into  $ST\_A$  using Merging Algorithm 1
  end while
end if

```

Fig. 27. Merging Algorithm 1 and its exception.

departing members will be concentrating on one area of the key tree, as shown in Fig. 26b. In this case, there are $(k-1)$ usable subtrees in the affected branch from level $\log_k D_o$ to $h-2$ and the $k - \lceil kD/N \rceil$ children of the root are not affected by the departing members, where D_o lies between 0 and N/k ($D = CN/k + D_o$) and C is between 0 and k . The best update cost is $(k-1)(h-2 - \log_k D_o) + k - \lceil kD/N \rceil$. All joining members form one or more subtrees with k members.

$$\text{Best } RC_{(J \geq 0, D \neq 0)} = \frac{k[(k-1)(h-2 - \log_k D_o) + \lceil \frac{J}{k} \rceil + (k - \lceil \frac{kD}{N} \rceil) - 1]}{k-1} + J. \quad (6)$$

Suppose we have $D = k^l + r$, where r lies between 0 and $(k-1)k^l$.

The same analysis still applies, except that r produces another Y subtrees, where Y is the sum of the digit of $k^{\lceil \log_k D_o \rceil} - D_o$ when written in radix k . In other words, the total number of usable subtrees is $(k-1)(h-2 - \lceil \log_k D_o \rceil) + k - \lceil kD/N \rceil + Y$. There are $D \bmod k$ siblings of the departing members. Therefore, the best update cost is

$$(k-1)(h-2 - \lceil \log_k D_o \rceil) + k - \lceil kD/N \rceil + D \bmod k.$$

```

if  $J > 0$  and  $D = 0$  then
  if  $(H_{MAX\_ST\_A} - H_{MIN\_ST\_B} \leq 1)$  and  $(H_{MAX\_ST\_A} - H_{MAX\_ST\_B} \leq 1)$  then
    if  $k > 2$  then
      if empty key node at root of  $ST\_A$  then
        insert  $ST\_B$  as the child at empty key node
      else
        create a new key node
        insert root of  $ST\_A$  and  $ST\_B$  as children of the new key node
      end if
    end if
    if  $k = 2$  then
      create a new key node
      insert root of  $ST\_A$  and  $ST\_B$  as children of the new key node
    end if
  end if
end if

```

Fig. 28. Merging Algorithm 2.

```

Identify and mark all key nodes along the path from departing members to root
Remove all marked key nodes
Classify siblings of departing members as joining members
while joining members  $> 0$  do
  if joining members  $\geq k$  then
    group joining members into one or more key trees with  $k$  members
  end if
  if joining members  $< k$  and joining member  $> 1$  then
    group remaining members into one key tree
  end if
  if joining members = 1 then
    treat it as a key tree of its own
  end if
end while
while number of subtrees  $> 1$  do
  select 2 key trees with minimum height
  combine them together using Merging Algorithm 1 or Merging Algorithm 2
end while
Construct update and rekey messages

```

Fig. 29. Batch Balanced Algorithm without optimization.

The $D \bmod k$ sibling of the departing members and the joining members form one or more subtrees with k members.

$$\text{Best } RC_{(J \geq 0, D \neq 0)} = \frac{k[(k-1)(h-2 - \lceil \log_k D_o \rceil) + \lceil \frac{J}{k} \rceil + (k - \lceil \frac{kD}{N} \rceil) + Y - 1]}{k-1} + J + D \bmod k. \quad (7)$$

APPENDIX B

The operations for both Merging Algorithms and the Batch Balanced Algorithm without optimization are shown in Figs. 27, 28, and 29, respectively.

REFERENCES

- [1] S.E. Deering, "Host Extensions for IP Multicasting," IETF RFC 1112, Aug. 1989.
- [2] S. Paul, *Multicast on the Internet and Its Applications*. Kluwer Academic, 1998.
- [3] U. Varshney, "Multicast over Wireless Networks," *Comm. ACM*, vol. 45, no. 12, pp. 31-37, Dec. 2002.
- [4] U. Varshney, "Multicast Support in Mobile Commerce Application," *Computer*, vol. 35, no. 2, pp. 115-117, Feb. 2002.
- [5] N. Shi, *Mobile Commerce Applications*. Idea Group, 2004.
- [6] R. Canetti, J. Garay, G. Itkis, D. Micciancio, M. Noar, and B. Pinkas, "Multicast Security: A Taxonomy and Efficient Constructions," *Proc. IEEE INFOCOM*, vol. 2, pp. 708-716, Mar. 1999.
- [7] A. Ballardie, "Scalable Multicast Key Distribution," IETF RFC 1949, 1996.
- [8] H. Harney and C. Muckerhirm, "Group Key Management Protocol (GKMP) Specification," IETF RFC 2093, July 1997.

- [9] H. Harney and C. Muckenhirn, "Group Key Management Protocol (GKMP) Architecture," IETF RFC 2094, July 1997.
- [10] D.M. Wallner, E.J. Harder, and R.C. Agee, "Key Management for Multicast Issues and Architectures," IETF RFC 2627, June 1999.
- [11] C. Wong, M. Gouda, and S. Lam, "Secure Group Communication Using Key Graphs," *IEEE/ACM Trans. Networking*, vol. 8, pp. 12-23, Feb. 2000.
- [12] D. Balenson, D. McGrew, and A. Sherman, "Key Management for Large Dynamic Groups: One-Way Function Trees and Amortized Initialization," Internet Draft, draft-irtf-smug-groupkeymgmt-00.txt, Aug. 2000.
- [13] M. Valdvogel, G. Caronni, D. Sun, N. Weiler, and B. Plattner, "The Versakey Frameworks: Versatile Group Key Management," *IEEE J. Selected Areas in Comm. (JSAC)*, vol. 17, no. 9, pp. 1614-1631, Sept. 1999.
- [14] M.P. Howarth, S. Iyengar, Z. Sun, and H. Cruickshank, "Dynamics of Key Management in Secure Satellite Multicast," *IEEE J. Selected Areas in Comm. (JSAC)*, Feb. 2004.
- [15] S. Mitra, "Iolus: A Framework for Scalable Secure Multicasting," *Proc. ACM SIGCOMM*, vol. 27, pp. 277-288, Sept. 1997.
- [16] B. DeCleene et al., "Secure Group Communication for Wireless Networks," *Proc. Military Comm. Conf. (MILCOM)*, Oct. 2001.
- [17] A. Perrig, "Efficient Collaborative Key Management Protocol for Secure Autonomous Group Communication," *Proc. Int'l Workshop CryptEC*, 1999.
- [18] X.S. Li, Y.R. Yang, M. Gouda, and S. Lam, "Batch Rekeying for Secure Group Communications," *Proc. 10th Int'l WWW Conf.*, May 2001.
- [19] S. Setia, S. Koussih, and S. Jajodia, "Kronos: A Scalable Group Rekeying Approach for Secure Multicast," *Proc. IEEE Symp. Security and Privacy*, 2000.
- [20] X.B. Zhang, S. Lam, D.Y. Lee, and Y.R. Yang, "Protocol Design for Scalable and Reliable Group Rekeying," *IEEE/ACM Trans. Networking*, vol. 11, pp. 908-922, Dec. 2003.
- [21] J. Pegueroles and F. Rico-Novella, "Balanced Batch LKH: New Proposal, Implementation and Performance Evaluation," *Proc. IEEE Symp. Computers and Comm. (ISCC)*, June 2003.
- [22] J. Pegueroles, J. Hernandez-Serrano, F. Rico-Novella, and M. Soriano, "Adapting GDOI for Balanced Batch-LKH," Internet draft, draft-irtf-gsec-gdoi-batch-lkh-00.txt, June 2003.
- [23] P.P.C. Lee, J.C.S. Lui, and D.K.Y. Yau, "Distributed Collaborative Key Agreement Protocols for Dynamic Peer Groups," *Proc. IEEE Int'l Conf. Network Protocols (ICNP)*, Nov. 2002.
- [24] A.M. Eskicioglu, "Multimedia Security in Group Communication: Recent Progress in Key Management, Authentication and Watermarking," *ACM Multimedia Systems J.*, special issues on multimedia security, pp. 239-248, Sept. 2003.
- [25] W. Ng and Z. Sun, "Multi-Layers LKH," *Proc. IEEE Int'l Conf. Comm. (ICC)*, May 2005.
- [26] M.J. Moyer, J.R. Rao, and P. Rohatgi, "Maintaining Balanced Key Trees for Secure Multicast," Internet Research Task Force (IRTF), Internet draft, draft-irtf-smug-key-tree-balance-00.txt, June 1999.
- [27] W.H.D. Ng, H. Cruickshank, and Z. Sun, "Scalable Balanced Batch Rekeying for Secure Group Communication," *Elsevier Computers and Security*, vol. 25, pp. 265-273, June 2006.
- [28] A. Hodjat and I. Verbauwhede, "The Energy Cost of Secrets in Ad Hoc Networks (Short Paper)," *Proc. IEEE Circuits and Systems Workshop (CAS)*, 2002.
- [29] Y. Sun, W. Trappe, and K.J. Ray, "A Scalable Multicast Key Management Scheme for Heterogeneous Wireless Networks," *IEEE/ACM Trans. Networking*, vol. 12, Aug. 2004.



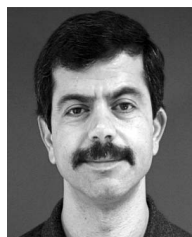
Michael Howarth received the bachelor's degree in engineering science and the DPhil degree in electrical engineering, both from Oxford University and the MSc degree in telecommunications from the University of Surrey, United Kingdom. Prior to joining the University of Surrey, he worked for several networking and IT consultancies. He is a lecturer in networking at the Centre for Communication Systems Research (CCSR), University of Surrey. His research interests include traffic engineering, quality of service, security systems, protocol design, and optimization of satellite communications. He is a chartered electrical engineer and a member of the United Kingdom IET.



Zhili Sun received the BSc degree in mathematics from Nanjing University, China, and the PhD degree in computing science from Lancaster University, United Kingdom. He is the chair of Communication Networking in the Centre for Communication Systems Research (CCSR), Department of Electronic Engineering, School of Electronics and Physical Sciences, University of Surrey, United Kingdom. He is a lecturer in data and Internet networking, as well as satellite communication courses, at the University of Surrey. He also teaches MSc, undergraduate, and industrial courses on satellite networking, computer and data networks, and Internet traffic engineering. He was a postdoctoral researcher from 1989 to 1993 in the Telecommunications Group, Queen Mary and Westfield College, University of London, before coming to Surrey. He has been a principal investigator and technical coordinator for many European projects, including the European Strategic Programme for Research and development in Information Technology on Broadband Integrated Satellite Network Traffic Evaluation (Esprit BISANTE) project on evaluation of broadband traffic over satellite using simulation approach, Validation of IP Telephony over Euroskyway Network (VIP-TEN) project on quality of service (QoS) of IP telephony over satellite, EU Fifth and Sixth Framework Programme GEOCAST projects on IP multicast over satellites, ICEBERGS project on IP-based Multimedia Conference over Satellite, SatLife project on IP over digital video broadcasting satellite/radar cross-section (DVB-S/RCS), satellite-based communications systems within IPv6 networks (SATSIX) project on IPv6 over satellite, and Euro-NGI project on next-generation Internet. He has also been a principal investigator for the UK Engineering and Physical Sciences Research Council (EPSRC), European Space Agency (ESA), and industrial projects on IP multicast security. He has supervised many PhDs and research fellows. He has also been a member of the technical committees for international conferences and of reviewers for EU and UK research proposals. He has acted as an external examiner for PhD viva at many universities in the UK, Europe, Singapore, and China. Since January 1996, he has been working on several European research projects in the Advanced Computational Software (ACTS), Esprit, Trans-European Telecommunications Network (TEN-TELECOM), and Information Society Technologies (IST) programs. His main research interests include network security, satellite network architectures, Voice over IP (VoIP), and IP conferencing over satellites. He has published a book titled *Satellite Networking* (Wiley) and more than 120 papers in international journals and conferences. He is a member of the Satellite and Space Communications Committee of the IEEE Computer Society and a chartered engineer and corporate member of the IET in the United Kingdom.



Wee Hock Desmond Ng received the BEng and PhD degrees in electronics and electrical engineering, both from the University of Surrey, United Kingdom. He is currently working at AT&T Singapore. His research interests include network security and next-generation network.



Haitham Cruickshank is a lecturer in data and Internet networking and satellite communication courses at the University of Surrey, United Kingdom. Since January 1996, he has been working on several European research projects in the Advanced Computational Software (ACTS), Esprit, Trans-European Telecommunications Network (TEN-TELECOM), and Information Society Technologies (IST) programs. His main research interests include network security, satellite network architectures, Voice over IP (VoIP), and IP conferencing over satellites. He is a member of the Satellite and Space Communications Committee of the IEEE Computer Society and a chartered engineer and corporate member of the IET in the United Kingdom.