CHAPTER 5

REVISED TWO PHASE BATCH REKEYING ALGORITHM

5.1 INTRODUCTION

Group communication enabled by multicast techniques is a topic of considerable interest today due to the growth of the internet and the widespread availability of high bandwidth connections. The need for secure multicast has also been recognized. Many applications, such as pay-per-view services and secure teleconferencing, require group access control and message privacy to be viable. Secure multicast techniques must gracefully tolerate frequent fluctuations in group membership. Members are generally allowed to join and leave groups at will, and access to multicast transmissions must be granted and revoked with minimal system overhead.

One way to implement secure multicast transmissions is through the use of message encryption. This requires that each authorized member of a secure multicast group has knowledge of a session encryption key shared by the entire group. The multicast host uses this session key for encrypting data packets before sending them to the group. When a member is evicted from the group, the session key must be changed in order to maintain message privacy. All the remaining group members receive the new session key by secure transmission, which is typically accomplished by multicasting an encrypted message containing the new key to the group. The message must be indecipherable to the evicted member, which means that each remaining group member must have one or more administrative encryption keys known also to the group controller (GC). The administrative keys are used only for
rekeying operations that take place when group membership changes. Each member is also assumed to possess a personal key, known only to the member and to the GC.

A number of scalable approaches have been proposed and one in particular, the key tree approach is analyzed in detail and extended in this chapter. 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. 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 techniques have been recently presented as a solution to overcome this problem. In such methods, a departed user will remain in the group longer and a new user has to wait longer to be accepted. All join and leave requests received within a batch period are processed together at the same time. A short rekey interval does not provide much batch rekeying benefit, whereas a long rekey interval causes a delay to the joining members and increases the vulnerability from departing members who can still receive the data.
The efficiency of the key tree approach critically depends on whether the key tree is balanced. 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 out degree 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 \( N \). This means that a member might need to perform \( N-1 \) decryptions in order to get the group key.

Recently, two Merging Algorithms suitable for batch join events for combining subtrees together was proposed. These two Merging Algorithms not only balance the key tree, but have lower rekeying costs compared to those of the existing algorithms. In order to additionally handle departing members, the above algorithm is extended 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. But this Batch Balanced Algorithm performs significantly better than the existing algorithms only 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. Our approach extends this algorithm further by using the two phase batch rekeying interval. This will serve to avoid the number of departing members exceeding the number of joining members as much as possible, and in turn, lead to improve the overall performance when compared with existing works.

5.1.1 Need for the Algorithm

For large multicast groups, the number of keys may become quite large, making efficient key management a non-trivial problem. Some end-user devices, such as mobile phones and PDAs, are memory constrained, so it is desirable to keep the number of keys stored by each member to a minimum. The key server must store the keys for the entire group, so the total number of keys must also be minimized. In order to save more bandwidth the update
messages should be optimized so that we can achieve optimum update cost of the key tree. Furthermore, group membership may change frequently, so the number of rekey messages needed to re-establish security when a member is evicted must be minimized as well. These concerns give rise to the need for efficient key management techniques that minimize both the number of keys and the number of rekey messages.

Secure group communication has applications beyond multicasting on the Internet. There are other situations that require the restricted sharing of information in groups whose membership fluctuates. Examples include various military applications, diplomatic communications, e-learning, air traffic control, etc. Further, end-users need not be human. Secure network management of computer or telecommunication networks could be facilitated by the use of secure group communications. Encrypted network management signaling messages could be multicast to trusted network entities, and key management techniques would be needed if the trusted network is large and experiences fluctuations in membership. Regardless of the underlying communication network the same concerns arise when the network gets large. The number of keys per user and the number of rekey messages needed to re-establish security should be as close to optimal as possible. To achieve all the above aims a new algorithm is proposed and the performance is evaluated along with that of the existing techniques.

5.2 BATCH REKEYING ALGORITHM

In batch rekeying, all join and depart requests received within a batch period are processed together at the same time. To do this we use two marking algorithms as well as two merging algorithms.
5.2.1 Batch Rekeying Approach Notations

Before we proceed with our work, we introduce some notations and definitions used in this chapter. We use “minimum height” to mean the minimum number of levels in a tree or sub tree from the root to any leaf node. We define the following variables:

<table>
<thead>
<tr>
<th>Variable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ST</td>
<td>Sub Tree</td>
</tr>
<tr>
<td>J</td>
<td>Number of joining members</td>
</tr>
<tr>
<td>D</td>
<td>Number of departing members</td>
</tr>
<tr>
<td>h</td>
<td>Height of key tree ((1+\log_k N))</td>
</tr>
<tr>
<td>H_MIN</td>
<td>Minimum height of the leaf node</td>
</tr>
<tr>
<td>H_MAX</td>
<td>Maximum height of the leaf node</td>
</tr>
<tr>
<td>H_INSERT</td>
<td>(H_{\text{MIN}}) of (ST_A)-(H_{\text{MAX}}) of (ST_B)</td>
</tr>
<tr>
<td>BI_MIN</td>
<td>Minimum Batch rekeying Interval</td>
</tr>
<tr>
<td>BI_MAX</td>
<td>Maximum Batch rekeying Interval</td>
</tr>
<tr>
<td>BRI</td>
<td>Batch rekeying interval</td>
</tr>
</tbody>
</table>

5.2.2 Marking Algorithms

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 available. We use the following two algorithms.

5.2.2.1 Marking algorithm 1

For this algorithm, there are four cases to consider. If \(J=D\), then all the 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. 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 the 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. Lastly, the GC generates the necessary keys and distributes them to the members.

### 5.2.2.2 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 the departing members that are not replaced by the joining members are marked as null nodes. For \( J > D \), all the 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. As before, the GC distributes the new key to the members.

### 5.2.3 Merging Algorithms

We now propose two Merging Algorithms to combine the subtrees together in a way that is suitable for batch join events. To handle all cases such as leave, or both join and leave 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 out degree k.

### 5.2.3.1 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 now describe Merging Algorithm 1. The criteria for choosing Merging Algorithm 1 is when the difference between $H_{\text{MAX}_{\text{ST}_A}}$ and $H_{\text{MIN}_{\text{ST}_B}}$ is greater than 1 and when the difference between $H_{\text{MAX}_{\text{ST}_A}}$ and $H_{\text{MAX}_{\text{ST}_B}}$ is greater than or equal to 1. If both these conditions are fulfilled, then the algorithm calculates $H_{\text{INSERT}}$. The following steps are then performed:

1. For $k > 2$, the algorithm searches for an empty child node in ST_A at either level $H_{\text{INSERT}}$ or level $H_{\text{INSERT}} - 1$. If $H_{\text{INSERT}} = 0$, then levels 0 and 1 are searched. If such a node exists, then the algorithm inserts ST_B as the child of that particular key node.

2. If an empty node is not found in Step 1, mark a suitable key node in ST_A at level $H_{\text{INSERT}}$ for insertion as follows: If $H_{\text{INSERT}} = 0$, 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 $H_{\text{MIN}_{\text{ST}_A}}$.

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.
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.

### 5.2.3.2 Merging algorithm 2

We now describe our Merging Algorithm 2. 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 \( H_{\text{MAX}_{ST_A}} \) and both \( H_{\text{MIN}_{ST_B}} \) and \( H_{\text{MAX}_{ST_B}} \) is 0 or equal to 1. The algorithm performs the following steps:

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.

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 the existing members. After updating the affected node IDs, the members can identify the set of keys that they need in the rekey messages.

### 5.2.4 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 the key nodes that need to be updated. These key nodes are on the ancestor paths from each departing member to the root.

2. Remove all the marked key nodes. After removal, there are only two types of elements left: the remaining subtrees and the joining members.

3. Classify all siblings of the departing members as joining members since all 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 any 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 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 Figure 5.1. 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 Figure 5.2. 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.

5.2.5  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 \( m_0 \). With the new node ID \( m_0 \), the members can update the remaining keys \( m_0 \) by using the following function:

\[
f(m_0) = k^x (m' - m) + m_0.
\]

where \( x \) denotes the level of the usable key tree.

![Figure 5. 3 (a) Usable and (b) new updated key trees](image-url)
For example, in Figure 5.3., 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.

5.3 REVISED TWO PHASE BATCH REKEYING ALGORITHM

In this proposed algorithm a variable length rekey interval was used. The batch balanced algorithm works well only if $J \geq D$. So our approach tries to keep this condition as much as possible.

5.3.1 Difficulty in Fixed Length Batch Rekeying

The batch rekeying with a variable interval is more suitable to the network than that with the fixed interval, because the batch rekeying with a variable interval leads to a steady rekey traffic and the cost of rekeying. Keeping this point in mind, we shall apply the variable batch rekey interval for rekeying.

5.3.2 Proposed Approach

It has two threshold interval levels; the lower threshold called as $B_{\text{MIN}}$, which means the minimum interval at which rekeying can occur. The higher threshold called as $B_{\text{MAX}}$, which means the maximum batch rekey interval. The exact batch rekey interval will be called as the batch rekeying interval ($B_{\text{RI}}$), and in the range of, $B_{\text{MIN}} \leq B_{\text{RI}} \leq B_{\text{MAX}}$.

Based on the multicast application & its required security level, we can choose the threshold limits for $B_{\text{MIN}}$ and $B_{\text{MAX}}$. Its operation is as follows: the current batch rekey interval was chosen based on the following condition. The algorithm will wait for the minimum batch interval $B_{\text{MIN}}$ to occur. After reaching the time interval $B_{\text{MIN}}$, the algorithm checks whether
J >= D condition is achieved or not. If the condition is satisfied then BI_{MIN} will be considered as the Batch rekey interval BRI. If the condition does not occur, then for each join or depart request the system will continuously check if the J>=D condition is achieved or not. And if it happens, that particular current time will be taken as the current BRI. But sometimes the condition J >= D will not occur for a long period of time and it will reach the maximum batch rekey limit BI_{MAX}. Then BI_{MAX} will be considered as the current BRI. Thus, we try to avoid the condition J < D as much as possible, and the performance of the algorithm improved further. After finding the BRI the group controller will apply the batch balanced algorithm. The above possibilities will be explained with various cases in the next section.

5.3.2.1 BRI Calculation

In Figure 5.4, various cases for calculating the batch rekeying interval are depicted and each case explained below. In Case (i) the group will receive the join requests and depart requests until it reaches the BI_{MIN}. When it reaches BI_{MIN} it will compare the total number of join and depart requests. If it found that J > D or J =D then the BI_{MIN} is considered as a batch rekey interval and the batch balanced algorithm applied.

In case (ii) and (iii), the group will receive the join and depart requests upto BI_{MIN} and check whether J >= D condition occurs or not and found that it does not happen. So it continuously receives join and depart requests and checks for the same condition. If it happened then that particular moment will be considered as the current BRI and the batch balanced algorithm applied. In the case of (iv) also, the above conditions are applied. But the condition J >= D will never occur within the threshold limits. So the BI_{MAX} is considered as the current batch rekey interval BRI and the batch balanced algorithm applied. To determine the batch interval we should
consider two factors: the average delay of users’ request response and the batch traffic which is determined by the number of users’ request in the batch interval. Also, while choosing the threshold limits $B_{I_{\text{MIN}}}$ and $B_{I_{\text{MAX}}}$ care should be taken, so that we can preserve the forward and backward secrecy optimally.

* Figure 5.4 Various possibilities for calculating BRI

\* Point at which $J = D$ occur
5.4 PERFORMANCE EVALUATION

In this section, we study the performance of our proposed algorithms and compare them with that of the existing Algorithms. We consider four performance metrics:

- Rekeying cost,
- Update cost,
- Minimum and maximum height in the key tree, and
- Key storage.

5.4.1 Simulation Setup

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.

5.4.2 Rekeying Cost

The rekeying cost denotes the total number of rekey messages that need to be sent to all the authorized group members, for them to learn the new group key. A higher rekeying cost means that more bandwidth is needed for the transmission.
5.4.2.1 Batch join rekeying costs

We have performed some simulations to compare the performance of both our Merging Algorithms with the existing work for batch join requests. For our simulations, we used a balanced binary key tree of 256 members with a height of 8. The number of joining members varies from 1 to 250.

Figure 5.5 Batch Join Rekeying Costs

In Figure 5.5, we can see that the 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.

5.4.2.2 Batch balanced algorithm rekeying cost

We have built a simulator for the rekeying cost of our Revised Batch Balanced 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. Figures 5.6 and 5.7 show the computed and simulated best and worst rekeying costs for a binary key tree. For the best case, the rekeying 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.
Figure 5.6  Best Rekeying Cost for k=2

Figure 5.7  Worst Rekeying Cost for k=2
If the departing members are randomly selected, then we obtain the mean rekeying costs that lie between the theoretical best and worst cases. 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 to compare their performance with that of our Batch Balanced Algorithm. All the simulators first construct a balanced key tree and then randomly pick the 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. 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 half 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.

**5.4.3 Update Cost**

The update cost denotes the total number of update messages that need to be sent to all the affected members after the key tree has been reorganized, for them to identify the keys that they need. Of the four existing algorithms, only Marking Algorithm 2 does not need to distribute the 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
the Merging Algorithms need to send out one update message to inform the affected members of the newly created node.

For the Batch Balanced Algorithm, there are some overheads 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. Figure 5.8 shows the total update messages that need to be sent to the remaining group members, including the siblings of the departing members, 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.

![Batch Balanced Algorithm Update Messages (k=2)](image)

Figure 5.8 Update Message for the Batch Balanced Algorithm (k=2)
If we assume that a key is 128 bits long and the node ID is 20 bits (that is, up to 220 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. Already we show how we can reduce the number of update messages needed by the group members for some rekey events.

5.4.4 Minimum and Maximum Height

The minimum and maximum height of the tree affect the members’ key storage and, thus, the number of decryptions needed by each member and may even increase the rekeying costs.

5.4.4.1 Height considering only member join

Figure 5.9 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 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 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. In the case of Marking Algorithm 1, the maximum height increases significantly as the number of joining members increases, because all 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.

Figure 5.9  Maximum height of the key tree

Figure 5.10 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. 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 high rekeying costs, as shown in Figure 5.5. On the other hand, if we can choose appropriately between both our Merging Algorithms depending on the number of joining members, we can create a balanced key tree without extra costs.

![Batch Join Performance Evaluation](image)

**Figure 5.10 Maximum Difference in Height**

### 5.4.4.2 Height considering member join and leave

Figures 5.11 and 5.12 show 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.
Figure 5.11 Minimum Height for Marking Algorithm 1

Figure 5.12 Maximum Height for Marking Algorithm 1
Figures 5.13 and 5.14 show 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 to the network costs.

Figure 5.13 Minimum Height for Marking Algorithm 2
Figures 5.14 and 5.16 show 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.

Figure 5.14 Maximum Height for Marking Algorithm 2

Figure 5.15 Minimum Height for Batch Balanced Algorithm
Figure 5.16 Maximum Height for Batch Balanced Algorithm

5.4.5 Key Storage

The key storage denotes the number of keys each member needs to store.

Table 5.1 Minimum and Maximum Key Storage for Batch Join Events

<table>
<thead>
<tr>
<th></th>
<th>Marking Algorithm 1</th>
<th>Marking Algorithm 2</th>
<th>Merging Algorithm 1</th>
<th>Merging Algorithm 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Min key storage</td>
<td>$\lceil \log_b(N) \rceil$</td>
<td>$\lceil \log_b(N+J) \rceil$</td>
<td>$\lceil \log_b(N+J) \rceil$</td>
<td>$\lceil \log_b(N+J) \rceil$</td>
</tr>
<tr>
<td>Max key storage</td>
<td>$\lceil \log_b(N+\log_b(J+1)) \rceil$</td>
<td>$\lceil \log_b(N+J) \rceil$</td>
<td>$\lceil \log_b(N+J) \rceil$</td>
<td>$\lceil \log_b(N+J) \rceil$</td>
</tr>
</tbody>
</table>

Table 5.1 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 storage inefficiency as in Marking Algorithm 1, but it comes at the expense of large rekeying costs, as shown in Figure 5.5. 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.

**Table 5.2 Min and Max Key Storage for Batch Join and/or Depart Events**

<table>
<thead>
<tr>
<th></th>
<th>Marking Algorithm 1</th>
<th>Marking Algorithm 2</th>
<th>Batch Balanced Algorithm</th>
</tr>
</thead>
<tbody>
<tr>
<td>Min key storage</td>
<td>$\left\lfloor \log_2(N) \right\rfloor$</td>
<td>$\left\lfloor \log_2(N_{\text{max}}) \right\rfloor$</td>
<td>$\left\lfloor \log_2(N + J - D) \right\rfloor$</td>
</tr>
<tr>
<td>Max key storage</td>
<td>$\left\lfloor \log_2(N) \right\rfloor + \left\lfloor \log_2(J - D + 1) \right\rfloor$</td>
<td>$\left\lfloor \log_2(N_{\text{max}}) \right\rfloor$</td>
<td>$\left\lfloor \log_2(N + J - D) \right\rfloor$</td>
</tr>
</tbody>
</table>

Table 5.2 shows the minimum and maximum key storage for the three algorithms. Since a full calculation of the key storage 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, $N_{\text{MAX}}$, 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.

5.5 **OPTIMIZATION**

From the above simulations, we observe that the Batch Balanced Algorithm has identical rekeying costs compared to those of the existing algorithms when the number of joining members and 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: \( D \leq J \leq (D - D_{\text{min}}) + kD_{\text{min}} \) where \( D_{\text{min}} \) 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 \( [(D - D_{\text{min}}) + (kD_{\text{min}})] \), then we replace all \( (D - D_{\text{min}}) \) departs at the maximum height with \( (D - D_{\text{min}}) \) joins. The remaining joining members are split across the \( D_{\text{min}} \) nodes.

Figure 5.17 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. But the rekeying costs still remain the same as the earlier one. 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.

![Optimized Batch Balanced Algorithm Update Messages](image-url)

**Figure 5.17 Update Message for the Optimized Batch Balanced Algorithm**
5.6 SUMMARY

In this chapter, we have presented the revised variable length batch rekeying algorithm along with the batch balanced algorithm. This algorithm tries to minimize the difference in height in the key tree without adding extra network costs. However, 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 the key tree and the rekeying costs, and therefore, lead to either an unbalanced key tree or high rekeying costs. Our proposed Algorithm provides a good compromise compared to the existing algorithms, producing a balanced key tree with low rekeying costs. As for other events, our Batch Balanced Algorithm outperforms the 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 tries to avoid the condition $J < D$ as much as possible and provide an optimal solution in terms of rekeying cost, and update messages. However, if the departing members are spread evenly across the key tree, then the highest rekeying cost happens at around $N/k$. 