Efficient Secure Multicast with Well-Populated Multicast Key Trees

M. Moharrum and R. Mukkamala
Department of Computer Science
Old Dominion University
Norfolk, Virginia 23529
{mohar_m, mukka}@cs.odu.edu

M. Eltoweissy
Department of Computer Science
Virginia Tech
Falls Church, VA 22043
toweissy@nvc.cs.vt.edu

Abstract
Secure group communications is the basis for many recent multimedia and web technologies. In order to maintain secure and efficient communications within a dynamic group, it is essential that the generation and management of group key(s) be secure and efficient with real-time response. Typically, a logical key hierarchy is used for distribution of group keys to users so that whenever users leave or join the group, new keys are generated and distributed using the key hierarchy. In this paper, we propose Well-Populated Multicast Key Tree (WPMKT), a new efficient technique to handle group dynamics in the key tree and maintain the tree balanced with minimal cost. In WPKT, sub-trees are swapped in a way that keeps the key tree balanced and well populated. At the same time, rekeying overhead due to reorganization is kept at a minimum. Another advantage of WPKT is that rebalancing has no effect on the internal key structure of the swapped sub-trees. Results from simulation studies show that under random user deletion, our approach achieves one order of magnitude in overhead less than existing approaches. Under clustered sequential user deletion, our approach achieves almost a linear growth with tree size under individual rebalancing. For periodic rebalancing, we achieved almost half the overhead introduced by other approaches.

1. Introduction
Secure group communications model is the basis for many recent multimedia and web technologies [3, 6, 11]. Secure group multicast represents the core component of many real-time applications that involve large groups. Applications such as pay-per-view, online gaming, command and control centers, and stock market price distribution need efficient management of group dynamics and security [7, 11]. One way to achieve security within group communications is to have a group key used by the sender to encrypt data, and by the receiver to decrypt. Such a key is known as Traffic Encryption Key (TEK). The group key can be used as well in different security services such as authentication and maintenance of message integrity among users [13].

Generation and management of group key(s) are essential to maintain the security of group communications. Secure group communications needs to satisfy three properties: Group Secrecy, Forward Confidentiality, and Backward Confidentiality. Group secrecy guarantees that non-group members have no access to the data sent to the group, even if they have the ability to sample the line. This type of requirement is usually achieved through encryption using the group key. Forward confidentiality guarantees that whenever a user leaves the group, they can no longer have access to the group communications. Backward confidentiality guarantees that whenever a new member joins the group, they have no access to past group communications. Both forward and backward confidentiality are usually achieved through changing the group key (Rekeying) whenever a member joins or leaves a group.

In this paper, we propose Well-Populated Multicast Key Trees (WPMKT), a new efficient technique to handle group dynamics in the key tree and maintain the tree balanced with minimal cost. Our approach depends on swapping sub-trees in a way that keeps the key tree balanced and well populated while keeping the rekeying operations due to reorganization at a minimum. WPMKT does not affect the internal key structure of sub-trees used for swapping. Results from simulation studies show that under random user deletion, our approach achieves one order of magnitude in overhead less than existing...
approaches, for example approaches in [4, 5, 9]. Under clustered sequential user deletion, our approach achieved almost a linear growth with tree size under individual rebalancing. For periodic rebalancing, we achieved almost half the overhead introduced by other approaches.

The paper is organized as follows. Section 2 briefly reviews current algorithms for maintaining the key tree balanced. Section 3 introduces WPMKT, our new approach to tree maintenance. Section 4 presents a performance study of our new approach compared to previous schemes and different factors that affect its performance. Finally, section 5 concludes and discusses some possible extensions and future work.

2. Related work

A key tree is a special case of a general solution called key graph [14]. The tree is a logical structure that maintains both group key(s), or TEKs, and administrative keys at different levels. In this section, we will briefly discuss some background on key tree maintenance.

Keeping the key tree balanced is very important for maintaining the logarithmic order of the rekeying procedure. There are different approaches to tree maintenance that preserve the balance under join conditions. Fewer solutions exist for maintaining the balance under massive leave conditions. One simple solution to model the LKH tree is to use a binary search tree based on the node ID [7, 16]. Another important constraint on the key tree is the allowance or disallowance of mixing. We consider two types of nodes in the key tree: internal nodes that carry only administrative key, and external nodes that hold user personal keys as well as all the administrative keys along the path to the root. In a non-mixing key tree, an internal node cannot have descendant nodes of different types (internal and external) [5]. In a mixing tree, an internal node can have descendants of mixed types as in [9].

Another solution is to handle the key tree as a B+ tree (B+-LKH) [4, 5]. This will include handling splits and joins along the path that may go up to the root in the worst case. The main advantage of such solution is that B+ tree is more efficient in larger groups. B+ tree provides a smaller height and hence a fewer number of key change per join or leave. On the other hand, maintaining a B+ tree include too much overhead for handling individual joins and leaves which include node splitting and/or merge with key updates and distribution.

Moyer et al introduced a separate balancing procedure in [9] that includes virtual removal of the deepest user node in the tree, and then rejoining that user at the shallowest node. This step is repeated until the tree becomes balanced. The balance condition (represented as the height difference between the deepest and the shallowest node compared to a threshold value) may be checked after every deletion, and the rebalancing procedure is invoked. Another option is to check the balance and invoke the rebalancing procedure periodically. The main problem with this solution is the overhead of changing all the keys along the two paths, which costs in average 2logdN. Another problem is that this solution handles balance-violating nodes one at a time, which repeat that overhead number of times equal to the number of such nodes.

3. Well-Populated Multicast Key-Tree (WPMKT)

In this section, we present our approach for balancing a multicast key-tree. We start by noticing the following two points about MKTs. Unlike search trees used in data management (e.g., binary search trees, AVLs), multicast-key trees (MKT) need not maintain the order of the keys. Thus, tree balancing should be less expensive in the case of multicast-key trees.

Moving the keys (or a sub-tree of leaves) around the tree is a common way to achieve tree balancing in MKTs. The cost of such movements is regeneration of internal keys and/or multicasting of keys.

Our approach for balancing MKTs depends on two procedures, tree compaction and tree rebalancing as follows:

3.1 Tree compaction

This procedure uses the concept of promotion. Promotion of a sub-tree (or a leaf node) is defined as the process of moving it up along its path to the root. For example, consider the sub-tree T3 in Figure 1.a. When it is moved up along its path to the root, i.e., promoted, the tree appears as in Figure 1a. Similarly, when both sub-trees T1 and T2 are promoted in Figure 1.c, the tree becomes as in Figure 1d. The procedure for tree compaction is as follows:
While there are intermediate nodes with number of children ≤ 1
{If there are no child nodes, remove the intermediate node from the tree;
Otherwise replace the intermediate node with its child sub-tree;}

This procedure is also illustrated in Figure 1, where tree in (a) is compacted to (b), and tree in (c) is compacted to (d). It is should be noted that tree compaction gets the leaf nodes closer to the root and hence attempts to reduce the tree height. Clearly, a compacted tree may be unbalanced. In such a case, we apply the tree rebalancing procedure.

It should also be noted that since compaction only moves a sub-tree up along its path towards the root, no rekey operations are involved. For example, when sub-tree T3 is moved one step up towards the root in Figure 1 (a and b), due to the removal of I2, nodes in T3 need not know any new keys. They simply need to discard the key for I2. Thus there is no rekeying cost due to tree compaction.

Figure 1. Illustration of tree compaction

3.2 Tree rebalancing

Maintaining balanced trees has been a classical interesting problem to many researchers. Several types of balanced trees have been around in the literature [2, 10]. Among them are AVL trees, Symmetric Binary trees, Weight-balanced trees, balanced trees, k-neighbor trees, and general balanced trees [1, 2, 10]. The main difference between these classical trees and our MKT is that all the tree types above handle in general sort trees of degree d. This partial order of the tree nodes places a restriction on some tree transformations. Each transformation must result in a tree that satisfies the ordering constraints. In addition, the whole set of balancing transformations must result in a tree that satisfies the balance condition. A different balance condition is associated with each of the above tree types. For more details on different types of balanced trees, we direct the interested user to [10].

A general balanced tree, GB-Tree, is a tree of degree d, which satisfies the following condition at the root node only: 
\[ h \leq c \log d N + b \]
where \( h \) is the tree height, \( c \geq 1 \), \( N \) is the number of leaf-nodes, and \( b \) is some constant. The main idea of the general-balanced tree is that it is considered balanced as long as it satisfies the logarithmic height in terms of the number of leaf-nodes. Many other balanced trees can be considered as special cases of the GB-Tree with different values of the constants \( b \) and \( c \). Rebalancing a GB-Tree is usually done using a technique known as partial rebuilding [1]. Partial rebuilding techniques locate the shallowest node that violates the balancing constraint, and reorganize the sub tree rooted at that node. Clearly, in sort tree, reorganizing a sub-tree takes the order constraint in consideration. Partial rebuilding can be done in \( O(\log d N) \) [1].

Prior to describing our rebalancing algorithm, we describe a few terms used in the algorithm.

**Height of a tree (sub-tree) \( (h) \):** The number of links between the root of the tree (sub-tree) and the farthest leaf node in the tree (sub-tree). For example, in Figure 2a height of the sub-tree rooted at I2 is 1, while the height of the sub-tree rooted at I3 is 2. Similarly, height of the tree (R) is 4.

**Depth of a node \( (d) \):** The number of links on the path between the node and the root of the tree. For example, in Figure 2a the depth of I2 is 1, while the depth of I3 is 2.

**Level of a node \( (l) \):** Height of the tree – depth of the node. (Level of the root of a tree is the height of the tree. \( l = h - d \). For example, in Figure 2a, the level of I2 is 3 \((4-1)\), while the level of I3 is 2 \((4-2)\).

**Weight of a tree (sub-tree) \( (W) \):** The number of descendant leaf nodes (i.e., external nodes) to that tree (sub-tree). For example, in Figure 2a, the weight of sub-tree rooted at I2 is 2 while the one rooted at I3 is 4. The weight of the tree (R) is 8.
Degree of unbalance of a sub-tree ($\beta$): $l_v - c \log_d W_v - b$ where $v$ is the sub-tree, and $b$ and $c$ are constants defined in the tree balancing condition. For example, in Figure 2a, suppose $b=0$, $c=1$, and $d=2$. Then the degree of unbalance of the sub-tree rooted at $I_2$ is 2 (3-1) and the one rooted at $I_3$ is 0 (2-2). The degree of unbalance of $R$ is 1 (4-3). Thus, the tree in Figure 2a is unbalanced but the sub-tree at $I_3$ is balanced.

Well-populated sub-tree: It is a sub-tree with $\beta_v \leq 0$. In other words, a well-populated sub-tree is one that satisfies the tree balance condition. How well the sub-tree meets the balancing condition is denoted by $-\beta_v$. Clearly, the sub-tree rooted at $I_3$ is well populated.

Ill-populated sub-tree: It is a sub-tree with $\beta_v > 0$. In other words, an ill-populated sub-tree is one that violates the tree balance condition. How much short the sub-tree falls in meeting the balancing condition is denoted by $\beta_v$. The tree rooted at $R$ is ill-populated and falls short by a degree of 1. (Intuitively, this means that the height of the tree is one larger than the optimal height for the given number of 8 leaves.) The sub-tree rooted at $I_2$ falls short by a degree of 2.

The primary goal of the rebalancing algorithm is to reduce the degree of unbalance of the tree. This is achieved by swapping the highest-level well-populated sub-tree along the longest path of the tree with the most ill-populated sub-tree. The condition that the well-populated sub-tree must be at a higher level than the ill-populated tree ensures that the degree of unbalance of the ill-populated tree is decreased due to the swapping to a lower level. Similarly, the condition on the weights ensures that the higher weighted sub-tree moves closer to the root and the lighter one moves away from the root. As a result of each swap operation, the resulting tree (including the sub-trees) is more balanced than the tree prior to the swap. Our approach swaps entire sub-trees, which are some times as large as one level less than the whole key tree. An advantage of subtree swapping is the preservation of the key structure of each swapped sub-tree. This might represent significant overhead reduction for future group activities specially in case of clustering related users at the same sub-tree. The algorithm is described in Figure 3.

As an illustration of the rebalancing algorithm,

\begin{verbatim}
While (the tree is ill-populated)
{
Step 1: Traverse along the longest paths from the root to:
{} Identify all well-populated sub-trees;
{} Order the identified sub-trees in the decreasing order of their height;
{} Let $WT_i$, $i=1,2,...,m$ be such sub-trees;
}
Step 2: For all sub-trees not on the longest path:
{} Identify all ill-populated sub-trees;
{} Order the identified sub-trees in the decreasing order of their degree of unbalance ($\beta$);
{} Let $IT_j$, $j=1,2,...,n$ be such sub-trees;
Step 3: Find the least possible $i$ and $j$ such that
{} $(level(WT_i) < level(IT_j)) \& (weight(WT_i) > weight(IT_j));$
{} If such a pair $(i,j)$ exists then
{} swap($WT_i$, $IT_j$);
{}else break;
}
\end{verbatim}

Figure 2. Illustration of tree rebalancing algorithm

Figure 3. Tree rebalancing algorithm
consider the MKT in Figure 2a. Assuming that \( b=0 \) and \( c=1 \), and with \( d=2 \) (binary tree), the degree of unbalance of R is \( 4\log_2 8 \) or 1. Thus R is unbalance d. While traversing along the longest paths (R--U1, R--U2, R--U3, R--U4), we find that sub-tree rooted at I3 is the one with the highest height of 2. Its weight is 4. Among the ill-populated sub-trees, we find that the sub-tree rooted at I2 is the one with the highest degree of unbalance \( \beta = 3 - \log_2 2 = 2 \). Its level is 3 and weight is 2. Since both the level condition and the weight condition are satisfied, we can swap them. The resulting tree is shown in Figure 2b. The new tree is balanced at root R and hence we terminate the algorithm.

To see how the above algorithm converges to a more balanced tree, we consider an ill-populated tree T to start with. Step 1 and 2 involve no change on the tree, we just try to identify two suitable sub-trees for swapping. In step 3, if failing to find such two suitable sub-trees, the algorithm terminates and we end up with the same tree T. In case two sub-trees are found and swapped, we end up with a new tree T’ with shorter longest path carrying more weight. The length of any path is less than or equal to the new longest path. Clearly, T’ has more balanced weight distribution than T. To show the convergence, we assume we have k suitable sub-trees in step 2. After step 3, we end up with k-1 suitable sub-trees for the next iteration. As we continue iterations, less suitable sub-trees are found till we end up finding no more sub-trees that satisfy the condition in step 2, and hence, the algorithm terminates.

4. Performance Evaluation
In order to evaluate the performance of our approach compared to the approach presented in [9], we performed some simulation experiments for both random and clustered user leaves. The tree balance was either checked individually, i.e., after every user deletion, or periodically, i.e., after every rekeying interval. One way to organize users in the key tree is to place them randomly according to the order they join the group as well as available nodes. Another way is to maintain related users as clustered in the key tree by joining them either individually or as a group to the same sub-tree. We applied the same stream of leaves to all the key trees involved.

We used trees of different sizes varying between 4000 and 32000 user, and applied excessive user leaves of up to 45% of the tree population. We considered the following in evaluating the performance of our approach. (i) Trees resulting from our WPMKT (ii) Trees resulting from Moyer et al’s approach [9] (iii) Optimal trees with mixing, and (iv) Optimal trees without mixing. In order to make sure the resulting tree is balanced, we compared the average tree height of the resulting key tree to the average tree height of a complete tree having the same number of leaf-nodes. The result is shown in Figure 3. It can be seen from the figure that both approaches achieve almost the optimal tree height with and without mixing allowed. Note that the tree height grows linearly with the logarithmic scale of the size.

The average tree height is directly proportional to both the average number of key change per deletion. In a key tree of degree d, distributing a new or updated key at a certain node requires at most d multicast messages. Thus, the total number of key changes is directly proportional to the number of multicast messages. To evaluate the efficiency of different tree balancing approaches, the total number of key changes was used.

4.1. Performance Under Random User leaves
The first model we applied is random user deletion in which random 45% of the tree population leave the group. In order to compare the overhead, we measured the overhead due to rebalancing in terms of the total number of key changes. To determine whether the tree needs rebalancing or not, we used the same criteria used in [9]. The tree is considered unbalanced when the height difference between the deepest and shallowest nodes is greater than certain threshold (typically 2 levels). The tree compaction technique described in Section 3 keeps the number of times the tree needs rebalancing relatively small. This is due to promotion of the sibling nodes of the deleted user, which keeps the tree somewhat crowded. A similar technique is used in [9] to attach the sibling of the deleted node to a higher parent. However, our compaction is more generic and efficient.
Figure 4 shows the number of times the rebalancing procedure is invoked (due to tree unbalance) in both our approach and that in [9]. The figure shows that in our approach, the number of times the tree needs rebalancing is relatively smaller than it is in [9]. This might be due to the tree compaction procedure we apply after every deletion as described in Section 3. In addition, as the tree size grows, the number of times rebalance is invoked also increases.

In order to evaluate the efficiency of the rebalance procedure, we measured the overhead per single invocation of the rebalance procedure. Figure 6 shows this per-invocation overhead in terms of the number of key changes for both our approach and Moyer et al’s approach. The figure shows that the average number of key changes per rebalance invocation is less in our approach than it is in [9]. The results may be explained as follows. Suppose we assume that the tree is unbalanced and that there are m leaf nodes at the deepest level. In Moyer et al’s approach, those m nodes will be handled individually moving one node at a time to the shallowest point of the tree till the tree is balanced [9]. This moving will involve changing all the keys along the two paths: from the deepest node to the root, and from the shallowest node to the root, every time a leaf-node is moved. In our approach, this case requires fewer key changes for the following two reasons:

- The m nodes are handled one sub-tree at a time instead of one node at a time; and each time a sub-tree is moved (swapped) the keys are changed along two shorter paths.
- Each path starts from the parent of one sub-tree and continues up to the first common root between the two swapped sub-trees.

One rebalancing option is to do individual rebalancing. In individual rebalancing, the key server checks the tree balance every time a user is deleted, and applies the rebalancing procedure if needed. Another option for performance enhancement is periodic rebalancing, in which the tree balance is checked periodically instead of every deletion.

Figure 6 shows the overhead of rebalancing measured in terms of the number of key changes under individual rebalancing. Clearly, the total number of key changes is the product of number of key changes per rebalance invocation (Figure 5) multiplied by the number of rebalance invocations (Figure 6). Our approach achieves almost one order of magnitude lower in the overhead compared with [9]. This reduction of overhead can be explained by two reasons:

- The fewer number of times the tree needs rebalancing (Figure 5); and
- The fewer number of key changes per rebalance invocation (Figure 6).

In general, the total number of deletions may be divided into d intervals, each of which carries D/d users, where D is the total number of deleted users. The rebalancing check is invoked in case of reaching D/d deleted users, or the timer times out. When invoking the rebalance procedure, the tree balance is checked and rebalancing is done when necessary. Figure 7 shows the overhead of rebalancing under periodic rebalancing with D=45% of total population and d =10 intervals.
4.2. Effects of Massive leaves of adjacent users

Clustering is a common group dynamic trend. In many real-life applications, a group of users may join or leave a group all together, or within a short period of time. We base our model for clustering on the assumption that the system assigns adjacent locations in the key tree to users who belong to the same cluster. We assume certain techniques used by the system to analyze the inter-user relationships and assign such adjacent locations. However, the details of such techniques may be out of the scope of this paper. We refer to the event of a group of such users leaving the system together as clustered leaving. Clustered leaving might even happen to members who did not join together, and cannot be identified as a sub-group. An example of such an event is multicast of a sport even when certain team loses a game, all fans of that team may be interested in leaving the multicast group. A key tree management algorithm should be able to accommodate such dynamics without affecting the remaining users with large number of rekey messages. Our simulation experiment included leaving chunks of consecutive users chosen randomly. We selected the chunk size to be 1% of the tree population and the total number of leaves is still 45%. We compared our approach to Moyer et al’s approach [9] under individual and periodic balancing. Figure 8 shows the overhead of individual rebalancing.

From these figures, it may be observed that periodic check for rebalancing under group leaves is more efficient than individual rebalancing. This can be due to the avoidance of unnecessarily moving of nodes that may be shortly deleted. It can also be noticed from the figures that clustered user deletion requires more rebalancing cost than random user deletion in individual rebalancing. This can be explained by the extra distortion caused to the key tree by the deletion of a cluster of users who mostly belong to the same sub-tree. Under periodic rebalancing, clustered user deletion was shown to require less rebalancing cost than random user deletion with periodic rebalancing. This may be explained by the fact that deleting multiple clusters of users will remove complete sub-trees that will be pruned off the original tree. The compaction procedure is believed to promote the remaining sub-trees up the tree, which may result in a less unbalanced tree than the case of individual user deletion. Another explanation is that the user clustering may have an effect similar to the effect of periodic rebalancing. This means that the case of clustered user deletion with periodic rebalancing can be seen as periodic rebalancing with each interval having a set of local sub-intervals represented as user clusters. This extra splitting of intervals may maximize the effect of periodic rebalancing in reducing the rebalance overhead.
5. Conclusion
In this paper, we presented Well-Populated Multicast Key Tree (WPMKT), a new approach to maintaining a balanced multicast key tree. WPMKT is composed of two major procedures, tree compaction and swapping appropriate sub-trees to keep the key tree balanced with minimal overhead in terms of number of key changes. Tree compaction introduces zero overhead as it depends on promoting nodes up in the path to the root without affecting the key set known to the user at leaf-nodes. Swapping sub-trees depends on an approach similar to partial rebuilding techniques used in General balanced trees. In addition to low overhead, swapping preserves the internal key structure of involved sub-trees. Results from the simulation studies show that our approach achieves one order of magnitude less than other approaches under random user deletion. Under clustered sequential user deletion, our approach achieved almost linear growth with tree size under individual rebalancing. For periodic rebalancing we achieved almost half the overhead introduced by other approaches. The main difference with our approach is disallowing mixed nodes, which produces a key tree with one level more than the tree resulted from an approach that allow mixing.

References