Reliable Group Rekeying: A Performance Analysis *

Yang Richard Yang, X. Steve Li, X. Brian Zhang, Simon S. Lam
Department of Computer Sciences
The University of Texas at Austin
Austin, TX 78712–1188
{yangyang,xli,zxc,lam}@cs.utexas.edu

TR–01–21
June, 2001

Abstract

In secure group communications, users of a group share a common group key. A key server sends the group key to authorized new users as well as performs group rekeying for group users whenever the key changes. In this paper, we investigate scalability issues of reliable group rekeying, and provide a performance analysis of our group key management system (called keygem) based upon the use of key trees. Instead of rekeying after each join or leave, we use periodic batch rekeying to improve scalability and alleviate out-of-sync problems among rekey messages as well as between rekey and data messages. Our analyses show that batch rekeying can achieve large performance gains. We then investigate reliable multicast of rekey messages using proactive FEC. We observe that rekey transport has an eventual reliability and a soft real-time requirement, and that the rekey workload has a sparseness property, that is, each group user only needs to receive a small fraction of the packets that carry a rekey message sent by the key server. We also investigate tradeoffs between server and receiver bandwidth requirements versus group rekey interval, and show how to determine the maximum number of group users a key server can support.

1 Introduction

Many emerging network applications, such as pay-per-view distribution of digital media, restricted teleconferences, and pay-per-use multi-party games, are based

*Research sponsored in part by NSF grant no. ANI-9977267 and NSA INFOSEC University Research Program grant no. MDA904-98-C-A901. Experiments were performed on equipment procured with NSF grant no. CDA-9624082.

upon a secure group communications model [8]. In this model, to protect the privacy of group communications, a symmetric group key known only to group users and the key server is used for encrypting data traffic between group users. Access to the group key is controlled by a group key management system, which sends the group key to authorized new users as well as performs group rekeying whenever the group key changes. Specifically, a group key management system can implement two types of access control: *backward access control* and *forward access control*. If the system changes the group key after a new user joins, the new user will not be able to decrypt past group communications; this is called backward access control. Similarly, if the system rekeys after a current user leaves, or is expelled from the system, the departed user will not be able to access future group communications; this is called forward access control.

Implementing access control may have large performance overheads which limit system scalability. Backward access control can be implemented efficiently because a new group key can be distributed by encrypting it with the existing group key for existing group users. Forward access control is harder to implement. To send a new group key to all remaining group users after a user has departed, one approach is to encrypt the new group key with each remaining user's *individual key*, which is shared only between the user and the key management system. This straightforward approach, however, is not scalable because it requires the key management system to encrypt and send the new group key $N - 1$ times, where N is group size before the departure.

In the past few years, several approaches [21, 22, 2, 4, 6] have been proposed to implement scalable forward access control. For example, the *key tree* approach, which uses a hierarchy of keys to facilitate group rekeying, reduces group rekeying complexity to $O(\log N)$ [21, 22], where N is group size.

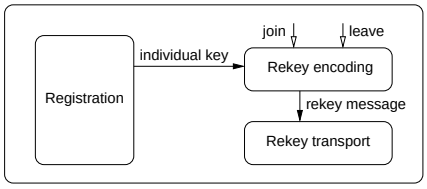

Figure 1: Functional components of a key management service

Figure 1 shows the functional components of an architecture for group key management system. The registration component authenticates users and distributes to each user its individual key. Authenticated users send their join and leave requests to the rekey encoding component. The rekey encoding component, which manages the keys in the system, validates the requests by checking whether they are encrypted by individual keys, and generates rekey messages, which are sent

to the rekey transport component for delivery. Previous studies have focused primarily on the rekey encoding component, particularly the processing time required by the rekey encoding component in a key server [21, 22]; the problem of reliable transport of group rekey messages has not been addressed in the literature. To make a group key management system scalable, however, the design of each of the three components needs to be scalable. Therefore, the objective of our study is to investigate scalability issues of all three components, including the evaluation of batch rekeying algorithms to improve scalability for a large and dynamic group, the characterization of rekey transport workload, the design of a reliable rekey transport protocol, and an overall performance analysis of our system, called keygem.

First, consider the registration component. For a group key management system to grant or deny a join or leave request, the identity of the user sending the request needs to be authenticated. Thus, each user needs to first register with the system by authenticating itself to the system and receive its individual key. Registration using an authentication protocol, however, can have large overheads, and a key server becomes a bottleneck when user registration rate is high. To improve the scalability of the registration component, the key server in keygem can offload its its registration workload to trusted registrars [8, 24]. Machines running registrars can be added or removed dynamically. Moreover, different registrars can use different authentication protocols to authenticate different sets of users. Since we can offload the registration workload to registrars, we do not consider this workload in this paper. For the detailed operations to register a new user, please see a description of the keystone system [24].

Second, consider the rekey encoding component. We show that rekeying after each join or leave (called individual rekeying) for the key tree approach has two problems: inefficiency and out-of-sync problems among rekey messages as well as between rekey and data messages (see Section 2). Furthermore, when user join/leave rate is high, the delay needed to reliably multicast a rekey message may be too large to implement individual rekeying. In keygem, we improve rekey encoding efficiency and alleviate the out-of-sync problems by rekeying periodically for a batch of join/leave requests. The idea of batch rekeying has been proposed before [4, 13, 18, 22]. However, for batch rekeying based on a key tree, no explicit algorithm has been presented and its performance has not been analyzed. In this paper, we present the specification of a batch rekeying algorithm, analyze its performance, and evaluate the benefits of batch rekeying. Our evaluation shows that batch rekeying not only can reduce the number of expensive signing operations, it also can reduce substantially bandwidth requirements at server and receivers. In other words, batch processing can improve system scalability for a highly dynamic group.

Third, consider the rekey transport component. Reliable transport of rekey messages has not received much attention in previous work. Although the idea of using FEC to improve the reliability of rekey transport has been discussed in the SMuG community [8] and in our keystone system, there is no protocol detail and its performance is not analyzed. The common assumption is that one of the reliable multicast protocols [7] can be used for rekey transport, and that prior analyses [11, 14, 20, 9, 15] of these reliable multicast protocols still apply. In this paper, we observe that rekey transport has its own special properties. First, we observe that rekey transport has an eventual reliability and a soft real-time requirement because of the inter-dependencies among rekey messages as well as between rekey and data messages. Second, we observe that rekey transport workload has a *sparseness* property, that is, while a key server sends a rekey message as a large block of packets, each receiver only needs to receive a small fraction of the packets. For our rekey transport protocol, which is based upon the use of proactive FEC [10, 17], we show that reliable rekey multicast can be analyzed by converting it to conventional reliable multicast, which does not have the sparseness property. Using this approach, we have investigated key server bandwidth overhead, number of rounds needed to transport the workload of a rekey operation, and how to determine the proactivity factor for FEC.

Fourth, consider the rekey encoding and the rekey transport components together. Based on a simple membership model, we show that group rekeying interval serves as a design parameter that allows tradeoffs between rekeying overheads, group access delay, and the degree of forward access control vulnerability. Considering four system constraints, we investigate how to choose an appropriate rekey interval and determine the maximum number of users that a key server can support.

To further improve the scalability and reliability of keygem, we allow keygem to extend a centralized key server to distributed key servers. Our performance analysis shows that partitioning users into active and inactive groups can further improve system scalability. In particular, we present two distributed architectures with one architecture suitable for applications with both security and reliability requirements on the transferred application data, such as reliable secure software transfer, and another architecture suitable for applications with only security requirement on the transferred application data, such as secure multimedia applications.

The balance of the paper is organized as follows. In Section 2, we investigate scalability issues of the rekey encoding component and evaluate periodic batch rekeying. In Section 3, we address the issues of reliable rekey transport, including rekey workload characterization and performance analysis of rekey transport. In Section 4, we integrate the results of Section 2 and Section 3 to consider overall system performance and study tradeoffs between bandwidth overhead and rekey

interval. Extensions to multiple key servers are presented in Section 5. Our conclusion is in Section 6.

2 Improving Rekey Encoding Scalability

Having been authenticated by a registrar, a user can then send a join request to the key server. The key server will also receive leave requests from existing users. The rekey encoding component processes these requests to prepare rekey messages. Before discussing the issues of individual rekeying, we first briefly review the key tree idea [21, 22].

2.1 Key tree

A key tree is a directed tree in which each node represents a key. The root of the key tree is the *group key*, which is shared by all users, and a leaf node is a user's *individual key*, which is shared only between the user and the key server. Since each node represents a key, we call a node in the key tree a key node. For key nodes representing the individual keys of users, we also refer to them as user nodes. A trusted key server manages the key tree, and a user u is given key k if and only if there is a directed path from its individual key to key k in the key tree. Consider a group with 9 users. An example key tree is shown in Figure 2. In this group, user u_9 is given three keys: k_9, k_{789}, and k_{1-9}. Key k_9 is the user's individual key, key k_{1-9} is the group key, and k_{789} is an auxiliary key shared by u_7, u_8, and u_9.

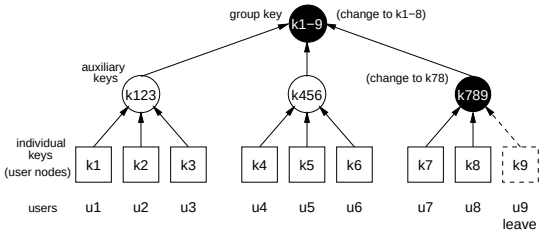

Figure 2: An example key tree

Suppose u_9 leaves the group. The key server will then need to change the keys that u_9 knows: change k_{1-9} to k_{1-8}, and change k_{789} to k_{78}. To distribute the changed keys to the remaining users using *group-oriented* rekeying strategy [22], the key server constructs the following *rekey message* by traversing the key tree bottom-up: ($\{k_{78}\}_{k_7}$, $\{k_{78}\}_{k_8}$, $\{k_{1-8}\}_{k_{123}}$, $\{k_{1-8}\}_{k_{456}}$, $\{k_{1-8}\}_{k_{78}}$). Here $\{k'\}_k$ denotes key k' encrypted by key k, and is referred to as an *encrypted key* or an

encryption. Upon receiving this message, a user extracts the encrypted keys that it needs. For example, u_7 only needs $\{k_{1-8}\}_{k_{78}}$ and $\{k_{78}\}_{k_7}$.

2.2 Issues of individual rekeying

Although individual rekeying introduces no extra delay to process user requests, it has two issues.

First, if we rekey after each join or leave, it is hard to control the synchronization that will arise because of the inter-dependencies among rekey messages as well as between rekey and data messages. When synchronization is not achieved, we will have *out-of-sync* problems. Consider an encryption $\{k\}_{k'}$ in a rekey message. A user must receive k' in order to decrypt the encryption. However, k' may be distributed in a previous rekey message, and if the previous rekey message has not arrived, the user will not be able to recover the new key. Also, consider a group key distributed in a rekey message to a user. If data messages are encrypted using the group key and the group key has not arrived, the user will not be able to decrypt the data messages. As a result of these out-of-sync problems, if rekey message delivery delay is high and join/leave requests happen frequently, a user may need to keep all of the old group keys, and buffer a large amount of rekey and data messages that it cannot decrypt yet.

Second, individual rekeying can be inefficient. For authentication purpose, each rekey message needs to be digitally signed to prove that it originates from the key server, and we know that signing operation can have large computation or bandwidth overheads. Moreover, as Snoeyink, Suri and Varghese observed in [19], which we have also independently derived at the same time using a different proof [25], we know that when a key server rekeys after each request and when forward access control is required, $\Omega(\log N)$ is a lower bound on the amortized number of encrypted keys. Thus, the key tree approach has already achieved the complexity of this lower bound, and we cannot further improve the performance of rekey encoding if we rekey after each request. To overcome this limit and reduce the number of signing operations, we need to consider batch rekeying.

2.3 Periodic batch rekeying

Periodic batch rekeying, which collects requests during a rekey interval and rekeys them in a batch, can alleviate the out-of-sync problems and improve efficiency. To alleviate the out-of-sync problems, periodic batch rekeying delays the usage of a new group key until the next rekey interval, and rekey transport can guarantee with a high probability that the rekey message has been delivered before the next interval (see Section 4). As for performance, an obvious performance gain of batch

processing J join and L leave requests is that it reduces the number of signing operations from $J + L$ to 1. Moreover, the number of encrypted keys generated by batch rekeying can be less than the sum of those generated by individual rekeying. Consider Figure 2. Suppose both u_8 and u_9 send leave requests. If the key server rekeys individually, it will need to update the group key twice, and at each time, the new group key needs to be encrypted by k_{123}. However, if the two requests are rekeyed in a batch, the key server only needs to update the group key once.

Periodic batch rekeying improves performance at the expense of delayed group access control, because a new user has to wait longer to be accepted by the group and a departed (or expelled) user can stay within the group longer. Thus, we observe that group rekeying interval serves as a design parameter that allows tradeoffs between rekeying overheads, group access delay, and the degree of forward access control vulnerability.

To accommodate different application requirements and make tradeoffs between performance and group access control, keygem can operate in three batch modes: 1) periodic batch rekeying, in which the key server processes both join and leave requests periodically in a batch; 2) periodic batch leave rekeying, in which the key server processes each join request immediately to reduce the delay for a new user to access group communications, but processes leave requests in a batch; and 3) periodic batch join rekeying, in which the key server processes each leave request immediately to reduce the exposure to users who have departed, but processes join requests in a batch. We will investigate the tradeoffs further in Section 4.

2.4 Batch rekeying algorithms

In periodic batch rekeying mode, the key server maintains a key tree that is slightly different from the key tree described in Section 2.1 to facilitate a key identification strategy that we proposed in [27]. In particular, we add null nodes that represent empty key nodes to a key tree so that the key server can always maintain a complete and balanced key tree. To identify each node in the key tree, the key server assigns integer IDs to tree nodes in breadth first search order, with the ID of the tree root as 0.

At the end of each rekey interval, the key server collects J join and L leave requests and executes the following marking algorithm to update the key tree and generate a *rekey subtree*. The objectives of the marking algorithm are to 1) reduce the number of encrypted keys; 2) maintain the balance of the updated key tree; and 3) make it efficient for users to identify the encrypted keys that they need.

The marking algorithm first updates the key tree. If $J \leq L$, the key server replaces J of the departed users that have the smallest IDs with the J newly joined

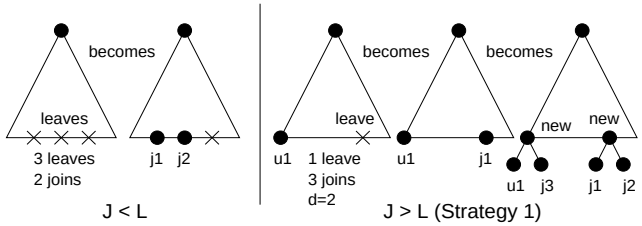

Figure 3: Example of marking algorithm for $J \neq L$.

users. By replacing departed users with newly joined users, the algorithm reduces the number of encrypted keys [12]. When $J < L$, we notice that some of the departed users will not be replaced. For these user nodes, the key server changes them to null nodes (see the left figure of Figure 3 for an example). If all of the children of a node are null nodes, the key server changes the node to null node as well. On the other hand, if $J > L$, the key server first replaces the L departed users with L of the newly joined users. However, the key server still needs to insert the remaining $J - L$ new users. For insertion, three strategies have been investigated to achieve different tradeoffs among the aforementioned three objectives:

- Strategy 1. In this strategy, to add the remaining $J - L$ new users, the key server first splits the L replaced nodes to add the remaining new users. If splitting the newly replaced nodes is still not enough to add all of the remaining new users (i.e. $J > d \cdot L$), the key server splits the leaf nodes from left to right and adds new users (see the right figure of Figure 3 for an example). The advantage of this approach is that it reduces the number of encrypted keys because it first splits the replaced user nodes. The disadvantage is that if the user nodes of some users are changed, the key server will need to provide new IDs individually to these users in addition to newly joined users. We notice that such notification will increase key server bandwidth overhead.

- Strategy 2. This strategy, which we proposed and investigated in [12], achieves a smaller number of encrypted keys than that of Strategy 1. With this strategy, the key server creates a tree with new users at its leaf nodes and grafts the tree under a departed user node with the smallest height. This strategy, however, does not keep the key tree as balanced as Strategy 1. On the other hand, with this strategy, the ID of at most one remaining user is modified; therefore, the key server only needs to provide new IDs to at most one remaining user in addition to newly joined users.

- Strategy 3. This strategy, which we proposed and investigated in [27], was designed to make it efficient for remaining users to identify the encrypted keys that they need. With this strategy, the key server first replaces the null nodes that have IDs between $d \cdot m + 1$ and $d \cdot m + d$ with newly joined users, where m is the ID of the last node in the key tree that is neither a user node nor a null node. If there are still extra joins, starting with the user node with ID $m + 1$, the key server splits a user node to add d children, moves the content of the user node to its left-most child, and adds $d - 1$ new user nodes. The key server repeats this process until all new users are added to the key tree. A disadvantage of this strategy is that it generates a slightly larger number of encrypted keys. The advantage of this strategy, however, is that if the key server multicasts m, the ID of the last node that is neither a user node nor a key node, in a rekey message, each remaining user will be able to independently derive the ID of its user node even if the structure of the key tree has been modified. For an explanation of how each user, whose ID has changed, determines its new ID, please see [27].

Comparing the three strategies to process the $J > L$ case, our evaluation shows that the difference in terms of the size of rekey subtree is small. Therefore, we report analytical results below for Strategy 3 only.

After updating the key tree, the key server makes a copy of the key tree, and marks the states of key nodes in the duplicated key tree. The nodes are marked with one of the following four states: *Unchanged*, *Join*, *Leave*, and *Replace*.

We first mark the states of user nodes: 1) A user node is marked *Unchanged* unless it is changed by the following rules. 2) A user node of a departed user is marked *Leave* if the node is not replaced; otherwise, it is marked *Replace*. 3) A user node is marked *Join* if it is a replacement for a null node or it is split from a previous user node.

We then mark the states of other key nodes: 1) If all the children of a key node are marked *Leave*, we mark it *Leave* and remove all of its children. 2) Otherwise, if all of its children are marked *Unchanged*, we mark it *Unchanged*, and remove all of its children. 3) Otherwise, if all of its children are marked *Unchanged* or *Join*, we mark it as *Join*, create a *virtual* node, which contains the old key of the key node, and use it to replace all of its *Unchanged* children. 4) Otherwise, if the node has at least one *Leave* or *Replace* child, we mark it as *Replace*.

We call the pruned subtree *rekey subtree*, and we observe that each edge in the rekey subtree corresponds to an encryption: parent node encrypted by child node. The detail of how to traverse a rekey subtree to generate a rekey message will be investigated in Section 3.1.

The running complexity of our marking algorithm is $O((J + L) \log N)$. Our

benchmark shows that on a Sun Ultra Sparc I with 167MHz CPU, the marking algorithm takes less than 4.5 ms for $N = 1024$, and less than 10 ms for $N = 4096$. On the other hand, according to our benchmark, the running time of a batch rekeying algorithm based on boolean function minimization [4] can take tens of seconds at similar group sizes.

2.5 Worst scenario analysis

We analyze the worst scenario and average scenario performance of batch rekeying based upon Strategy 3. (An analysis of batch rekeying based upon Strategy 2 was presented in [12].) The metric we use is the number of encrypted keys. In this subsection, we will show that even if we consider the worst number of encrypted keys to rekey L leave requests, assuming no joins in a batch, batch rekeying can still have large benefit. From our previous discussion, we know that it is because of forward access control that makes rekey encoding difficult; therefore quantifying the benefit of batch rekeying under this scenario can be instructive. For results on worst case performance of other cases, we refer the interested reader to [12]. We present the average performance in next section.

Consider a balanced tree with degree d and height h. We know that there are $N = d^h$ leaf nodes. Suppose L of the users leave. We observe that the worst scenario happens when the departed users are evenly distributed on the tree leaf nodes, and therefore, the number of overlapped encryptions is the minimum.

Without delving into the detail of analysis (see Appendix A.1), assuming $L = d^l$, where $L \leq N/d$, we derive that the worst number of encrypted keys is:

$$Enc_{worst}(N, L) = Ld \log_d \frac{N}{L} + \frac{L - d}{d - 1} \tag{1}$$

On the other hand, in individual rekeying, a single departed user costs $d \log_d N$. Suppose the L requests are processed individually, then there will be about a total of $Ld \log_d N$ encrypted keys. Comparing with Equation (1), we observe that the difference is $Ld \log_d L$. When L is large, the benefit of batch rekeying can be substantial. When $L \geq N/d$, more edges in the rekey subtree will be pruned, and the savings become even larger.

2.6 Average scenario analysis

Let $Enc(N, J, L)$ denote the average number of encrypted keys when J join and L leave requests are processed for an N user key tree. To simplify the analysis, we assume that the key tree is balanced at the beginning of a batch, and we let $h = \log_d N$ denote the height of the key tree. Also, we assume that the departed

users are uniformly distributed over the tree leaf nodes. The scenario that users have different leave probabilities can be utilized to further improve performance, for example, by using a Huffman type of tree to minimize the number of encrypted keys. However, such exploration and analysis are beyond the scope of this paper.

Since our batch rekeying algorithm depends on the relationship between J and L, our analytical results also depend on the relationship between J and L. By considering the number of times that a key node belongs to a rekey subtree, we derive the following analytical expressions for the average number of encrypted keys (see Appendix A.2):

- $J = L$:

$$Enc_1(N, J, L) = d \sum_{l=0}^{h-1} d^l (1 - \frac{C_{N-N_0}^J}{C_N^J})$$

  where $N_0 = N/d^l$.

- $J < L$:

$$Enc_2(N, J, L) = Enc_1(N, L, L) - (L - J) - \sum_{l=0}^{h-1} \sum_{i=0}^{d^l - 1} (\sum_{k=J}^{L-N_0} \frac{C_{N_1}^k C_{N_2}^{L-k-N_0}}{C_N^L})$$

  where $N_0 = N/d^l$, $N_1 = i \cdot N_0$, $N_2 = N - (i+1)N_0$.

- $J > L$:

$$\begin{aligned} Enc_3(N, J, L) = {} & \lceil \tfrac{d \cdot (J-L)}{d-1} \rceil + \\ & \sum_{l=0}^{h-1} \sum_{i=0}^{d^l - 1} (d(1 - \frac{C_{N-N_0}^J}{C_N^J}) + \\ & \frac{C_{N-N_0}^J}{C_N^J} \cdot \mathbf{1}(J - L - dN_1) \cdot \\ & \min\{d, \lceil \tfrac{J-L-dN_1}{N/d^{l+1}} \rceil + 1\}) \end{aligned}$$

  where $N_0 = N/d^l$, $N_1 = i \cdot N_0$, $\mathbf{1}(x) = 1$ if $x > 0$; otherwise, $\mathbf{1}(x) = 0$.

Next, we plot our analytical results. Figure 4 shows the values of $Enc(N, J, L)$ for $N = 4096$ and a wide range of J and L values. We have plotted both simulation results (controlled by achieving a confidence interval of 5%) and our analytical results; our analytical results match simulations well and they are indistinguishable in the figure. From Figure 4, we observe that for a fixed L, $Enc(N, J, L)$