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
{yangyang,xli,zxc,lam}@cs.utexas.edu

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 upon a secure group communications model [7]. 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 [20, 21, 2, 4, 5] 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 $O(\log N)$ [20, 21], where N is group size.

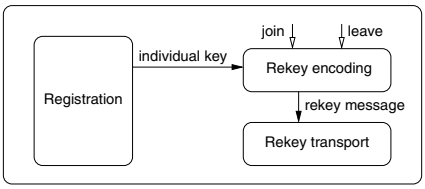

Figure 1: Functional components of a key management service

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 [20, 21]; 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

*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.

SIGCOMM'01, August 27-31, 2001, San Diego, California, USA..

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 [7, 23]. 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 [23].

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, 12, 17, 21]. 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 [7] 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 [6] can be used for rekey transport, and that prior analyses [10, 13, 19, 8, 14] 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 [9, 16], 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.

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. Our conclusion is in Section 5.

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 [20, 21].

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.

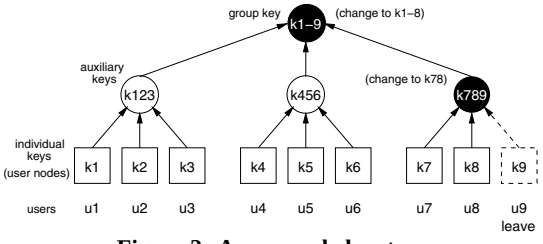

Figure 2: An example key tree

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 [21], 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 [18], which we have also independently derived at the same time using a different proof [24], 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 [26]. 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.

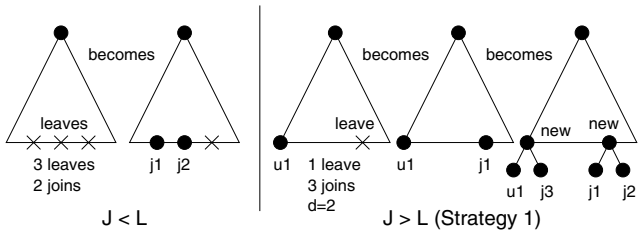

Figure 3: Example of marking algorithm for $J \neq L$.

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 users. By replacing departed users with newly joined users, the algorithm reduces the number of encrypted keys [11]. 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 [11], 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 [26], 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 [26].

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 [11].) 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 [11]. 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, assuming $L = d^l$, where $L \leq N/d$, we derive that the worst number of encrypted keys is:

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

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 [25]:

- $J = L$:

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

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

- $J > L$:

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

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

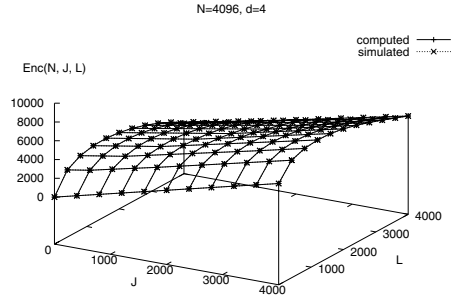

Figure 4: $Enc(N, J, L)$ by analysis and simulation

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, $Enc(N, J, L)$ grows linearly. This is expected because in our marking algorithm, joins replace leaves and thus the rekey subtree grows linearly with the number of joins. For a fixed J, $Enc(N, J, L)$ first increases (because more leaves means more keys to be changed), then decreases (because now some keys can be pruned from the rekey subtree).

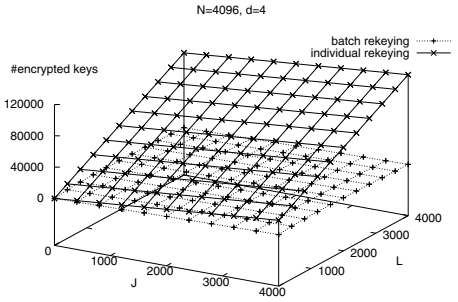

Figure 5: Batch vs. individual rekeying

Next, using the analytical expressions above, we consider the performance gains of batch rekeying when the average number of encrypted keys is used as performance metric. Figure 5 compares batch rekeying and individual rekeying for a wide range of J and L values. From Figure 5, we observe that the difference between batch and individual rekeying can be large. For $J = 400$, $L = 400$, batch rekeying generates 2547 encrypted keys, while individual rekeying generates 12000 encrypted keys, which is about 4 times larger; for $J = 0$, $L = 400$, batch generates 2146 encrypted keys, and individual generates 9600 encrypted keys; for $J = 400$, $L = 0$, batch generates 583 encrypted keys, and individual generates 2400 encrypted keys. The difference becomes even larger when J and L become larger.

3. PROVIDING RELIABLE REKEY TRANSPORT

A rekey subtree generated by the rekey encoding component is sent to the rekey transport component for delivery. We investigate two issues for rekey transport:

1. What are the special characteristics of the rekey transport workload?

2. Given the workload, how to provide reliability to the rekey packets, and what is the performance?

3.1 Rekey transport workload

For an encoding algorithm based on key trees, we know that each user only needs to receive the encrypted keys that are on the path from its individual key to the new group key. To avoid the overhead of unicasting individually to each user its encrypted keys, however, the key server partitions the users into a small number of subgroups (we consider one subgroup using one multicast channel in this paper), combines the encrypted keys for the subgroup of users into a rekey message, which may be partitioned into several rekey packets if the keys cannot fit into one packet, and multicasts the rekey message to all of the users in the subgroup. Instead of receiving all of the packets in a rekey message, however, each user only needs to receive those packets that contain its encrypted keys. As a result, the rekey packets that each user needs will depend on how encrypted keys are assigned into rekey packets. Therefore, our investigation of rekey transport workload is to address the following questions: 1) How to assign the encrypted keys in a rekey subtree of a subgroup of users into packets? 2) Given the assignment algorithm, how many packets a user needs to receive? What is the average? What is the variance? and 3) How many packets are there in a rekey message?

3.1.1 Key assignment algorithms

To improve the performance of rekey transport protocol, it is desired that a key assignment algorithm reduces the number of packets z_r that a user r needs to receive. Moreover, the overhead of rekey transport also depends on the users with the largest numbers of packets to receive. Thus, it is desired that a key assignment algorithm also reduces the variance of $\{z_r\}$. Given these requirements, we consider three key assignment algorithms: Breadth First Assignment (BFA), Depth First Assignment (DFA), and Recursive BFA (R-BFA). The common characteristic of these three key assignment algorithms is that they do not duplicate encrypted keys, that is, each encrypted key is assigned into only one packet. In [26], we have also proposed and investigated a different algorithm called User-oriented Assignment (UKA). The advantage of the UKA algorithm is that it assigns all of the encrypted keys for a user into the same packet, and therefore each user needs to receive only one packet, that is, $z_r = 1$ for all receivers. The disadvantage of this algorithm, however, is that some encrypted keys are duplicated into several packets, and such duplications can dominate bandwidth overhead, especially when MTU is small or when receiver loss rates are low.

For BFA and DFA, the key server traverses a rekey subtree using either breadth first or depth first order, and assigns sequentially the

encrypted keys into packets. By horizontally scanning a rekey subtree, BFA collects keys from different users in a round-robin manner. This "fairness" for each user reduces the variance of $\{z_r\}$. On the other hand, BFA spreads the keys of a user into multiple packets, and increases the average of $\{z_r\}$. By vertically tracing along a path, DFA first collects the keys for one user, and then goes to the next user. Thus, we expect that the average of $\{z_r\}$ is smaller for DFA. However, since the shared encrypted keys are assigned to the users processed earlier, such bias causes larger variance of $\{z_r\}$.

To gain the benefits of both BFA and DFA, we consider R-BFA. Figure 6 shows our R-BFA algorithm. This algorithm starts by calling R-BFA($root$), where $root$ is the root node of a rekey subtree.

```
Algorithm R-BFA (node_id)
▷ node_id uniquely identifies a node in rekey subtree.
▷ pkt is a global variable, denoting a rekey packet.
▷ family(i) is the set containing i and its immediate children.
1. Q ← create a local FIFO queue
2. put node_id into Q
3. while (Q is not empty)
       i ← pop the head element head(Q)
       if pkt has the space to contain family(node_id)
       then  put all the children of i into Q sequentially
             put all the encrypted keys of family(i) into pkt
       else  pkt ← generate a new rekey packet
             call R-BFA (i)
             while (Q is not empty)
                 i ← pop head(Q)
                 call R-BFA (i)
```

Figure 6: Recursive BFA (R-BFA) algorithm

To better understand R-BFA, we compare its bahavior with that of BFA. When there is still space in the current packet, R-BFA behaves just like BFA, and thus has performance in terms of variance similar to that of BFA. However, when the current packet is full and a new packet is created, instead of continuing horizontally scanning on the global rekey subtree (as BFA will do), R-BFA does BFA within a local subtree. Thus, R-BFA puts more related keys together and reduces the average value of $\{z_r\}$ compared with that of BFA. Figure 7 illustrates the basic idea of R-BFA.

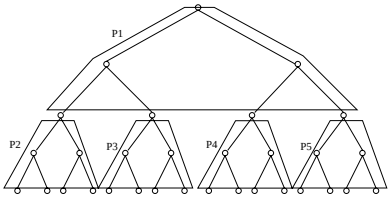

Figure 7: An illustration of the R-BFA algorithm

3.1.2 Comparison of assignment algorithms

We next verify that compared with BFA and DFA, R-BFA performs well in terms of both the average and the variance of $\{z_r\}$.

First, we consider the average value of $\{z_r\}$. Figure 8 plots the results of BFA, DFA and R-BFA for rekey subtrees with 2048 users. From this figure, we observe that over a wide range of J and L values, the average of $\{z_r\}$ for R-BFA and DFA is smaller than that of BFA. Since both R-BFA and DFA put related keys together, they achieve similar performance. We also observe from this figure that for R-BFA and DFA, on the average, users only need to

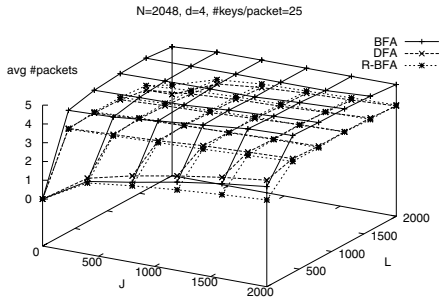

Figure 8: Average of $\{z_r\}$ for different J and L values

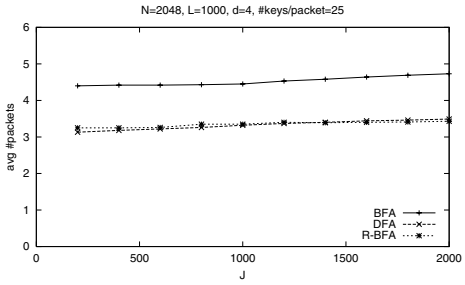

Figure 9: Average of $\{z_r\}$ for $L = 1000$

receive 3 to 4 packets even when J and L have been varied over a wide range. For clarity, Figure 9 shows the results for $L = 1000$.

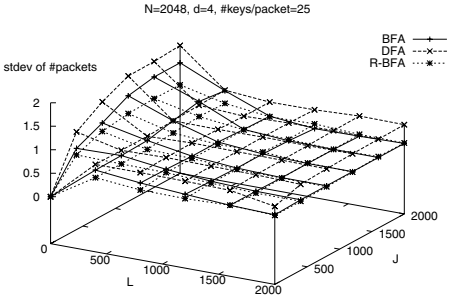

Figure 10: Variance of $\{z_r\}$ for different J and L values

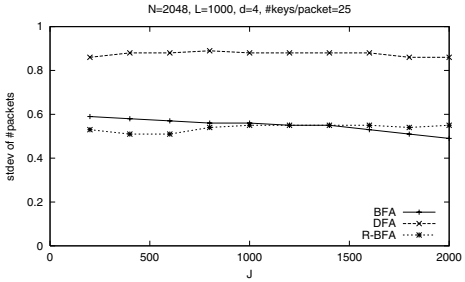

Figure 11: Variance of $\{z_r\}$ for $L = 1000$

Next, we consider the variance of $\{z_r\}$. From Figure 10, we observe that DFA has large variance while the variances of R-BFA and BFA are smaller. This is expected because we know that R-BFA and BFA treat users more fairly, and therefore will have smaller variances. For clarity, Figure 11 shows the results for $L = 1000$.

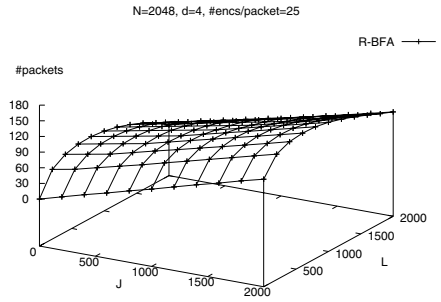

Figure 12: Average rekey message size

For comparison purpose, Figure 12 shows the average number of packets in one rekey message over a wide range of J and L values for a rekey subtree with 2048 users. Comparing Figure 12 with Figure 8, we observe that a user only needs to receive a small fraction of the packets in a rekey message. We refer to the property that each user only needs to receive a small fraction of the packets in a rekey message as the *sparseness* property.

3.2 Reliable rekey transport protocol

Given the rekey workload generated by a rekey subtree, we next investigate rekey transport protocol. Following the convention of reliable multicast, we also refer to a group user as a receiver.

To determine the rekey transport protocol, we need to first consider its properties. Although many reliable multicast protocols have been proposed and analyzed in the past few years [6], we cannot consider rekey transport as a conventional reliable multicast because of its specific properties. Besides its sparseness workload, we observe that rekey transport also has the following requirements:

- Eventual reliability. By eventual reliability, we mean that a receiver should be able to receive all of its encrypted keys. This requirement comes from the inter-dependencies discussed in Section 2.
- Soft real-time requirement. By soft real-time requirement, we mean that the transport of a rekey message is finished with a high probability before the start of the next rekey interval. The objective of the soft real-time requirement is to alleviate the out-of-sync problems.

We address the eventual reliability requirement by allowing receivers to send *re-synchronization* requests [23] when they cannot recover a rekey message in time. To provide soft real-time rekey transport, we use proactive FEC which can reduce recovery latency. In [9, 16], the authors have shown that round-based proactive FEC approaches can reduce delivery latency. Furthermore, it has been shown in [13] that a hybrid approach combining FEC and ARQ can significantly reduce the bandwidth requirements of a large reliable multicast session. In [14], the authors compare the benefits of combining local recovery with an FEC/ARQ hybrid technique and conclude that for many multicast scenarios, such a combination offers little improvement over an FEC/ARQ hybrid technique without local recovery. Given above, our performance evaluation of rekey transport is based on a simple round-based proactive FEC protocol.

One potential scalability problem of a reliable multicast protocol is the *feedback implosion* problem, and mechanisms such as tree-based feedback aggregation or NACK avoidance can be used to reduce feedback traffic [6]. Furthermore, for a proactive FEC based reliable multicast protocol, Rubenstein, Kurose and Towsley have observed that the number of NACK packets can be reduced by increasing proactivity factor [16]. In [26], we have proposed and investigated an adaptive proactive FEC algorithm, and our evaluations show that the number of NACK packets to the key server can be controlled by adjusting the proactivity factor at the beginning of each rekey interval. Given these results, in this paper, we assume that each receiver unicasts its feedback packets directly to the key server.

send k original and $\lceil k(\rho-1) \rceil$ FEC packets
at the end of each round:
  collect a_{max} as the largest a_r
at the beginning of next round:
  generate a_{max} FEC packets, and multicast to all receivers

Figure 13: Key server protocol

First, consider the key server protocol as specified in Figure 13. At the beginning of a rekey interval, the key server first runs a key assignment algorithm to assign the keys in a rekey subtree into k packets. Following the convention, we refer to the k packets in a rekey message as a *block* and k as the block size. After generating the k original packets, the key server makes each of the k packets individually verifiable by flow signing [22], and generates $\lceil (\rho-1)k \rceil$ proactive FEC packets, where $\rho \geq 1$ is the proactivity factor at the current rekey interval. To generate the FEC packets, the key server uses the Reed Solomon codes [15] when block size k is small and uses the Tornado codes [3] when k is large. The advantage of Reed Solomon code is that it allows a user to recover the k original packets from any k distinct packets, and therefore we base our analysis on Reed Solomon code. The Tornado codes, which may require a slightly higher number of packets to recover all of the original packets, also have advantages because they have smaller encoding time, and they may also allow a user to recover its packets without recovering all of the original packets. After multicasting the original and proactive FEC packets, the key server waits for the duration of a round, which is the largest round-trip time of all receivers, and collects feedbacks from the receivers. The feedback of each receiver is a NACK packet containing the number of packets a_r that it needs in order recover its packets. At the beginning of the next round, the key server calculates a_{max}, the largest of all a_r, generates a_{max} new FEC repair packets, and multicasts the repair packets to receivers. This process continues until the end of this rekey interval. We notice that it is possible that the key server may still receive NACK packets from some receivers at the end of a rekey interval. In this case, the key server considers the NACK packets as re-synchronization requests and sends re-synchronization packets to these users via reliable unicast. Since the design objective of keygem is to make sure that this scenario will rarely happen, we do not discuss this scenario in this paper. For a strategy in which the key server targets to multicast for only one or two rounds instead of until the end of a rekey interval, please see [26].

Next, consider the receiver protocol as specified in Figure 14. Assume the encrypted keys of a receiver r are assigned into z_r packets. Then, during the first round, the receiver checks whether it has received the z_r packets. If the receiver has received the z_r packets or is able to recover the z_r packets using FEC because it has received at least k distinct packets, the receiver extracts its encrypted keys and does not participate in the following rounds. Otherwise, the receiver will need to participate in the following rounds to receive a total of at least k distinct packets to recover the k orig-

```
for the first round:
    if receive its specific z_r or at least k distinct packets
      done
    else at the end of this round
      set a_r = k − (the number of distinct packets received)
      report a_r to key server
for each of the following rounds:
    if receive at least k distinct packets
      recover the encrypted keys
      done
    else at the end of this round
      set a_r = k − (the number of distinct packets received)
      report a_r to the key server
```

Figure 14: Receiver protocol I (with sparseness property)

inal packets, including its z_r packets. For this case, the receiver sets $a_r = k - y$, where y is the number of distinct packets that it has received, and reports a_r to the key server. For comparison purpose, we have also shown in Figure 15 a receiver protocol in which a receiver needs to receive all of the k packets in a block, that is, a workload that does not have the sparseness property. To distinguish these two protocols, we call the first protocol, which considers sparseness property, as Protocol I, and the second protocol, which can be seen as a conventional reliable multicast protocol, as Protocol II.

```
if receive at least k distinct packets
    construct the k original packets
    done
else at the end of each round
    set a_r = k − (the number of distinct packets received)
    report a_r to the key server
```

Figure 15: Receiver protocol II (conventional reliable multicast)

3.3 Performance of reliable rekey transport

Given the workload of rekey transport and the rekey transport protocol, we analyze in this section the performance of reliable rekey transport. To determine the guidelines for system design, we are interested in two performance metrics. Our first metric is bandwidth overhead, which we define as the ratio of k'/k, where k' is the total number of packets that the key server sends for a block of packets, including the repair packets to provide reliability, and k is the block size. Our second metric is latency, which we define as the number of rounds to deliver a rekey message to all receivers.

3.3.1 Analysis assumptions

For performance analysis, instead of considering a specific key assignment algorithm, we consider general key assignment algorithms and assume the distribution of $\{z_r\}$ as an input to the analysis. Furthermore, for all numerical results in this section, we assume that the number of receivers N is equal to 2048, and that all receivers have a z_r value of 6. For a balanced key tree of degree 4 with 2048 users, we know that 6 is the maximum number of encrypted keys that a user needs to receive. Thus, the value of z_r for a user r will be less than or equal to 6. As a result, the reported numerical results are upper bounds for the results of any specific key assignment algorithm.

As for the N receivers, we assume that n_h of them have a high packet loss rate of p_h, and the other $n_l = N - n_h$ have a low loss rate of p_l. For simplicity of analysis, we assume that receivers have independent losses, and that the losses of different packets are independent (for simulation results of correlated losses, please see [26]). For numerical results, we use the default values of $n_h = 615$, $p_h = 20\%$, $n_l = 1433$, and $p_l = 2\%$.

3.3.2 Conversion from protocol I to protocol II

Since a receiver does not need to receive all of the packets in a rekey message, we observe that previous analyses [19, 10, 13, 8, 14] of reliable multicast cannot be directly applied. Our key observation, however, is that we can convert the analysis of rekey transport workload into the analysis of conventional reliable multicast workload. In particular, we can convert one protocol instance where receivers run Protocol I to another protocol instance where receivers run Protocol II. Here, by a protocol instance, we mean a session with a given number of receivers running a given protocol.

Consider the following condition $C0$ at the end of the first round:

> C0: A receiver r has received its specific z_r packets, and the total number of packets it has received is less than k.

We observe that if we remove the receivers who satisfy condition $C0$ at the end of the first round, we have converted the analysis of an instance of Protocol I to an instance of Protocol II with reduced number of receivers, and therefore we can reuse the results of previous analyses for conventional reliable multicast. To be more specific, let n_h and n_l denote the numbers of high loss and low loss receivers in an instance of Protocol I. Let $N_{n_h}^{high}$ and $N_{n_l}^{low}$ denote the random variables of the numbers of high loss and low loss receivers who do not satisfy condition $C0$ at the end of the first round, where $0 \le N_{n_h}^{high} \le n_h$, and $0 \le N_{n_l}^{low} \le n_l$. Thus, our analysis of an instance of receiver Protocol I with n_h high loss receivers and n_l low loss receivers has been converted to the analysis of an instance of Protocol II with $N_{n_h}^{high}$ high loss and $N_{n_l}^{low}$ low loss receivers.

Let $Q_{n_h,n_l}(h,l)$ denote $Pr\{N_{n_h}^{high} = h, N_{n_l}^{low} = l\}$. Since we assume the losses of the receivers are independent, we have that

$$Q_{n_h,n_l}(h,l) = Q_{n_h}^{high}(h) \cdot Q_{n_l}^{low}(l)$$

where $Q_{n_h}^{high}(h)$ denotes $Pr\{N_{n_h}^{high} = h\}$, that is, the probability of h of the n_h high loss receivers do not satisfy $C0$, and $Q_{n_l}^{low}(l)$ denotes $Pr\{N_{n_l}^{low} = l\}$.

The remaining issues are to derive $Q_{n_h}^{high}(h)$, where $0 \le h \le n_h$, $Q_{n_l}^{low}(l)$, where $0 \le l \le n_l$, $E[N_{n_h}^{high}]$, and $E[N_{n_l}^{low}]$. The derivation details are shown in [25].

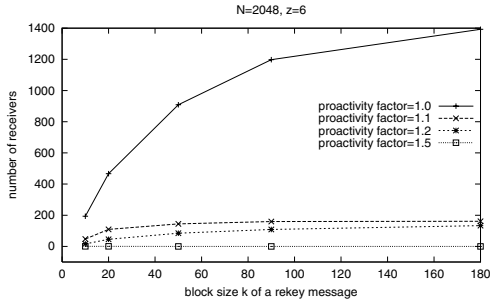

Figure 16: Expected number of receivers satisfying $C0$

To see the benefit of the sparseness rekey workload, Figure 16 shows the number of receivers satisfying $C0$ at the end of the first round. Since this number represents the reduction of the number of receivers when we convert from Protocol I to Protocol II, it reflects the savings of the sparseness rekey workload. We observe from this figure that when message block size k is large, and when proactivity factor ρ is small, the performance of a rekey multicast is equal to the performance of a conventional reliable multicast with a much smaller number of receivers.

3.3.3 Bandwidth overhead

We analyze in this section the bandwidth overhead of rekey transport. Given n_h high loss and n_l low loss receivers, we let $O(n_h, n_l)$ denote the random variable of bandwidth overhead when receivers run Protocol I. Let $E[O(n_h, n_l)]$ denote the mean value of this random variable. Given h high loss and l low loss receivers, we let $O^{II}(h, l)$ denote the random variable of bandwidth overhead when receivers run Protocol II. Let $E[O^{II}(h, l)]$ denote the mean of this random variable. Given the conversion from Protocol I to Protocol II, we have that:

$$E[O(n_h, n_l)] = \sum_{h,l} Q_{n_h}^{high}(h) \cdot Q_{n_l}^{low}(l) E[O^{II}(h, l)] \quad (2)$$

For given h and l, we can derive $O^{II}(h, l)$ by considering only Protocol II, and the detailed derivation of $E[O^{II}(h, l)]$ is shown in [25].

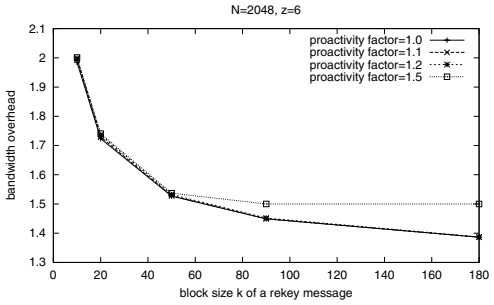

Figure 17: Overhead $E[O(n_h, n_l)]$

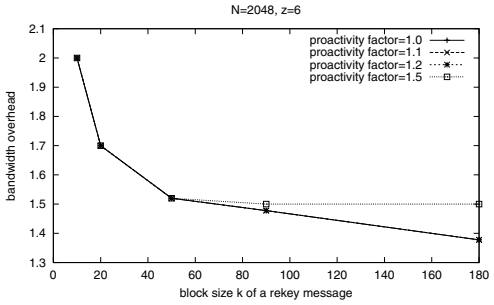

Figure 18: Overhead by *ns* simulation

Figure 17 shows our analytical results of rekey transport bandwidth overhead as functions of the block size k of a rekey message and proactivity factor ρ. To validate our analysis, Figure 18 shows simulation results using the *ns* simulator. Comparing both figures, we observe that our analytical results match with simulation results very well over a wide range of message block size and proactivity factor. We observe from Figure 17 that even with the sparseness property, the bandwidth overhead of reliable rekey transport is still high. Even for a rekey message with a large block size k, which has better transport bandwidth efficiency, to reliably transport the rekey message, the key server still needs to send a large amount of repair packets. For a smaller rekey message, the overhead is even higher. For example, for a rekey message with block size 20, the key server needs to send about $14\ (= (1.7 - 1) \cdot 20)$ repair packets.

3.3.4 Rekey transport latency

We measure the latency of rekey transport by the number of rounds to deliver a rekey message to all receivers. It is intuitive that rekey transport latency will also depend on the block size k of a rekey message and proactivity factor ρ.

Figure 19 shows the simulation results for the number of rounds to transport rekey messages with different message block size k at different proactivity factor ρ. We make the following observations. First, we observe that at a large proactivity factor ρ, the number of rounds to transport a rekey message with a large block size k can be smaller than that of a smaller rekey message. This is somehow counter intuitive because we expect the number of rounds to transport a large rekey message should always be larger than that of a smaller rekey message. To explain this result, we notice that for a rekey message with a large block size k, when proactivity factor ρ is large, the probability that a receiver will receive at least k out of the total $\lceil k\rho \rceil$ packets becomes higher; therefore, rekey transport latency reduces. On the other hand, if proactivity factor ρ is small, the number of rounds to transport a large rekey message is larger than that of a smaller rekey message.

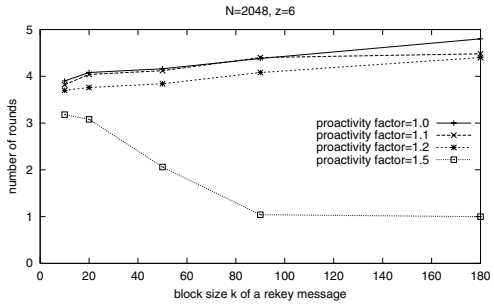

Figure 19: Rekey transport latency by *ns* simulation

Let $R(n_h, n_l)$ denote the random variable of the number of rounds to rekey n_h high loss and n_l low loss receivers when receivers run Protocol I. Let $R^{II}(h, l)$ denote the random variable of the number of rounds to transport k packets to h high loss and l low loss receivers when receivers run Protocol II. Similar to Equation (2), we have

$$E[R(n_h, n_l)] = \sum_{h,l} Q_{n_h}^{high}(h) Q_{n_l}^{low}(l) E[R^{II}(h, l)] \quad (3)$$

Therefore, we again convert the analysis from Protocol I to Protocol II. However, an exact calculation of the number of rounds to transport a rekey message requires complicated calculations involving modeling of transition states. Therefore, we derive an upper bound on $E[R^{II}(h, l)]$. The derivation of the upper bound is shown in [25].

3.3.5 How to determine proactivity factor ρ?

In our previous investigations of bandwidth overhead and rekey transport latency, we have considered the impacts of both the block size k of a rekey message and proactivity factor ρ. Given a rekey subtree and a key assignment algorithm, we know that block size k is determined. The proactivity factor ρ, however, is a protocol

parameter of a rekey transport protocol. We next discuss how to determine ρ.

To determine ρ, we observe that the key server can reduce rekey transport latency and the number of receiver feedbacks by increasing proactivity factor ρ. When ρ is large, the key server will send more proactive repair packets in the first round; therefore, more receivers will receive their packets in the first round, less receivers will send feedback packets to the key server, and the key server will send less repair packets in the following rounds. From Figure 20, for example, we observe that for a rekey message with block size 20, when the key server increases ρ from 1 to 1.7, rekey transport latency is reduced from 5 to about 1. Therefore, the key server can reduce rekey transport latency by increasing ρ. However, we also notice that the key server may set ρ to be too large and therefore increase bandwidth overhead. From Figure 20, for example, we observe that if the key server sets ρ to be higher than 1.7, then bandwidth overhead is dominated by proactivity factor and increases linearly with ρ while rekey transport latency stays flat.

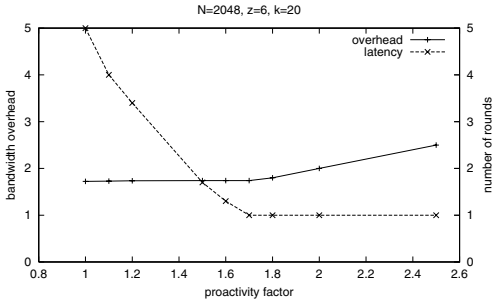

Figure 20: Overhead and latency as function of ρ

Given the above observations, we know that the key server should choose ρ such that rekey transport latency is close to 1 while the bandwidth overhead curve still stays flat. For example, in Figure 20, a good choice of ρ will be 1.7. In real implementation, however, the key server does not know the loss properties of the receivers (for example, independent loss assumption tends to overestimate the amount of redundancy needed when losses are shared [13]), and the block size of a rekey message may vary at different rounds. Thus, the key server should dynamically adjust ρ at each round. For example, in one type of strategy, the key server can adjust ρ using stochastic or AIMD (additive-increase-multiplicative-decrease) control so that rekey transport latency is close to a reference value, say 1 to 2 rounds. In another type of strategy, which we proposed and investigated in [26], the key server adjusts ρ in a way such that the number of receivers sending feedbacks is close to a small value, say 5% of the receivers. For our following performance analysis, we determine ρ by choosing the largest proactivity factor that still gives the lowest bandwidth overhead.

4. TRADEOFFS OF BANDWIDTH OVERHEAD AND REKEY INTERVAL

In Section 2, given J join and L leave requests in a rekey interval for a group with N users, we have investigated the marking algorithm to generate a rekey subtree. Given a rekey subtree, in Section 3, we have investigated rekey transport, and evaluated bandwidth overhead. Combining the results of Section 2 and Section 3, given J, L, and N, we can derive bandwidth overhead. Given group size N and user behaviors, we also know that J and L will be a function of rekey interval T. Thus, rekey interval T serves as a system design parameter that a group key management system can use to control bandwidth overhead. Furthermore, given user behaviors and system constraints, it is possible that a group key management cannot find a suitable T for a given group size N. Under this scenario, the group key management system needs to partition users into several groups to reduce group size.

In the remainder of this section, we first discuss a simple membership model. Then we discuss system performance metrics and the tradeoffs between bandwidth requirements and rekey interval T. Finally, we discuss several system constraints and an algorithm to determine rekey interval T as well as the number of users N that a single key server can support.

4.1 Membership dynamics

To quantify the numbers of joins and leaves arriving in a rekey interval T, we need to specify the arrival rates of joins and leaves. These arrival rates, which we call membership dynamics, depend on application and user behaviors. The only investigation about membership dynamics that we are aware of was by Almeroth and Ammar [1]. They showed that user join and leave behaviors in an audio multicast session follow exponential distributions.

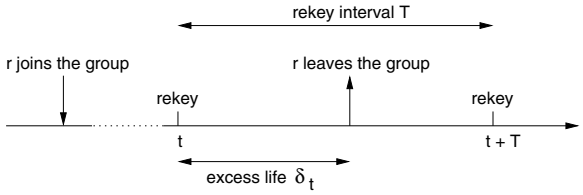

Figure 21: Illustration of excess life δ_t

To model the number of leave requests in a rekey interval T for a group with N users, we specify the distribution of the time each user spends in the group. Let $F(y)$ denote the cumulative distribution function of the time a user stays in the group. Let δ_t denote the remaining time a user will stay in the group, given that it is in the group at time t, which is the start time of a rekey interval. We call δ_t the excess life of a user at time t. Figure 21 illustrates the concept of excess life. Let m denote the mean value of the time that a user stays in the group. When the system is in steady state, from renewal theory, we have

$$p_L(T) = Pr\{\delta_t \leq T\} = \frac{1}{m}\int_0^T (1 - F(y))dy$$

where $p_L(T)$ is the probability that a user will leave the group before the end of the rekey interval. Therefore, for a group with N users, the expected number of leave requests $L(N,T)$ in a time period T will be $L(T,N) = N \cdot p_L(T)$. Here, we write $L(T,N)$ to indicate that the number of leaves will be a function of both T and N. In particular, we assume the amount of time that each receiver spends in a group is exponentially distributed with mean value of m. Denoting $\lambda_L = 1/m$, we have $L(T,N) = N(1 - e^{-\lambda_L T})$.

To model the number of join requests in a rekey interval T, we can assume user's arrivals are Poisson with a rate of λ_J. Therefore, our overall membership dynamics can be modeled as an $M/G/\infty$ system. For evaluation purpose, we assume that the group is in steady state, that is, $J = L$.

4.2 System metrics and tradeoffs

The two types of entities that participate in a group key management system are the key server and receivers. Accordingly, the four potential bottleneck resources are the CPU processing demand on the key server or a receiver and the bandwidth requirement of the key server or a receiver. Since CPU power keeps increasing, and

our evaluations show that in most cases CPU demands are not the limiting factors, we concentrate our efforts on the bandwidth requirements. To determine the rekey interval, another performance metric we consider is rekey transport latency since it gives a lower bound on the rekey interval.

We formally specify the following performance metrics:

- Key server outgoing bandwidth $B_{ks}(N,T)$. Let $S_{ks}(N,T)$ denote the total bytes that a key server multicasts to the N users in order to reliably transmit a rekey message. Since J and L are functions of N and T, we rewrite $Enc(N,J,L)$ as $Enc(N,T)$. Let B_p, B_{FEC}, and B_u denote the packet size of an original rekey packet, an FEC packet, and a re-synchronization packet, respectively. Let M denote the number of encrypted keys per packet, and let O_D denote packet duplication overhead of a key assignment algorithm (i.e. the number of packets generated by a key assignment algorithm divided by the number of packets generated by a key assignment algorithm without duplicate assignment). Let $E[O_b]$ denote the mean value of multicast bandwidth overhead as we defined in Section 3.3. Let $E[U]$ denote the mean number of re-synchronization packets that the key server needs to transmit. We have

$$B_{ks}(N,T) = \frac{S_{ks}(N,T) + B_u \cdot E[U]}{T} \tag{4}$$

where

$$\begin{aligned} S_{ks}(N,T) &= O_D \cdot \lceil \frac{Enc(N,T)}{M} \rceil \\ &\quad \cdot (B_p + (E[O_b]-1)B_{FEC}) \end{aligned}$$

- Receiver incoming bandwidth $B_r(N,T)$. Let $S_r(N,T)$ denote the total bytes that a receiver receives from multicast. Let p_r denote the packet loss rate of receiver r. We know that $S_r(N,T) = S_{ks}(N,T)(1-p_r)$. Assuming the probability that a receiver needs re-synchronization is small, we have

$$B_r(N,T) = \frac{S_r(N,T)}{T} \tag{5}$$

- Rekey transport delay $D(N,T)$. Let $R(N,T)$ denote the number of rounds for the key server to transmit a rekey message to the N users. Let D_R denote the largest round trip time from the key server to receivers. We have

$$D(N,T) = R(N,T) \cdot D_R \tag{6}$$

Our first observation is that $B_{ks}(N,T)$, $B_r(N,T)$, and $D(N,T)$ are all increasing functions of N, that is, bandwidth requirements and rekey latency increase as we increase group size. We also observe that p_r is generally not very high; therefore, $B_{ks}(N,T)$ and $B_r(N,T)$ are close to each other. Since a key server is likely to have much larger bandwidth than receivers, we plot overall bandwidth requirement for $B_r(N,T)$ only.

Figure 22 plots $B_r(N,T)$ as functions of N and T. The upper figure assumes that each receiver stays in the group for 3 minutes, and the lower figure assumes that each receiver stays in the group for 1 hour. We observe from both figures that $B_r(N,T)$ is a decreasing function of T. We also observe clearly from these curves the tradeoffs between bandwidth requirements and access control effectiveness. To determine a suitable rekey interval, a balance between performance requirements and access control effectiveness has to be achieved.

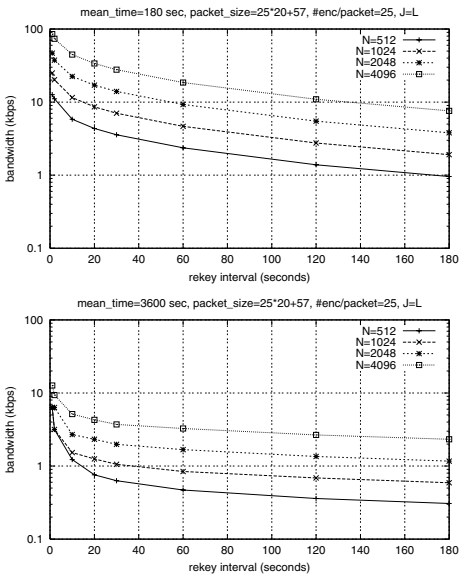

Figure 22: Bandwidth requirement vs. rekey interval

4.3 System constraints and algorithm

To decide the rekey interval and the maximum number of users a key server can support, we identify the following four potential system constraints.

1. $B_{ks}(N,T) \leq B_{ks}^{max}$. Here B_{ks}^{max} is a system specified bandwidth limit for the key server. For example, B_{ks}^{max} can be 10% of the key server's outgoing bandwidth. This constraint specifies a lower bound on T.

2. $B_r(N,T) \leq B_r^{max}$. Similar to B_{ks}^{max}, B_r^{max} is a receiver bandwidth limit. This constraint specifies another lower bound on T.

3. $D(N,T) \leq T$. This is to ensure that a rekey transport can finish before the start of the next rekey interval. We notice that with this constraint, the number of receivers sending re-synchronization requests can be greatly reduced. Therefore, this constraint specifies the third lower bound on T.

4. $T \leq T_{max}$. T_{max} is a constant determined either by business model or by application security requirements, and it specifies an upper bound on T. For example, one possible specification can be that the number of departed users that still have the group key is less than 5% of the total users. For our membership model, it means that $1 - e^{-\lambda_L T} \leq 0.05$; therefore, we can set T_{max} to be $-ln(0.95)/\lambda_L$ to satisfy this constraint.

Given the above constraints, we choose $T = T_{max}$ to minimize bandwidth requirement, if the three lower bounds are smaller than T_{max}. However, it is possible that no rekey interval can satisfy all four constraints. In that case, we need to determine the maximum group size that one key server can support, and partition users into smaller groups. An algorithm to determine the maximum group size one key server can support is shown in Figure 23. For how to partition users into smaller groups according to their behaviors and two architectures to extend a centralized key server to distributed key servers, one of which is based on Kronos [17], please see [25].

$$N_1 = \max\{n \mid B_{ks}(n, T_{max}) \leq B_{ks}^{max}\}$$
$$N_2 = \max\{n \mid B_r(n, T_{max}) \leq B_{ks}^{max}\}$$
$$N_3 = \max\{n \mid D(n, T_{max}) \leq T_{max}\}$$
$$N = \min\{N_1, N_2, N_3\}$$

Figure 23: Algorithm to determine N

5. CONCLUSION

In this paper, we have investigated the scalability issues of reliable group rekeying, and provided a performance analysis of keygem. Instead of rekeying after each join or leave, we use periodic batch rekeying to improve scalability and alleviate out-of-sync problems. Our analyses show that batch rekeying can achieve large performance gains. As for rekey transport, our investigations show that rekey transport has an eventual reliability and a soft real-time requirement, and that the rekey workload has the sparseness property. We then present a reliable rekey transport protocol based upon the use of proactive FEC. We show that reliable rekey transport in our design can be analyzed by converting it to conventional reliable multicast. We have also showed the tradeoffs between bandwidth requirements and rekey interval. Considering four system constraints, we provide some guidelines for choosing an appropriate rekey interval and determining the maximum number of users a key server can support. Our future work includes investigations of dynamic partitioning of group users, more detailed trace based experimental evaluations, and investigations of FEC encoding schemes that work better for a workload with the sparseness property. The investigation of FEC encoding schemes for sparseness workload is especially interesting because it can also apply to other applications such as stock quote delivery.

6. ACKNOWLEDGMENTS

We thank Ellen Zegura for her constructive comments in shepherding the final revision of this paper. We also thank Min S. Kim, Dong-Young Lee, Yanbin Liu, and Peiyu Wang for their assistance.

7. REFERENCES

[1] K. Almeroth and M. Ammar. Collection and modeling of the join/leave behavior of multicast group members in the mbone. In *Proceedings of High Performance Distributed Computing Focus Workshop (HPDC '96)*, Syracuse, New York, USA, August 1996. link: http://imj.ucsb.edu/publications.html.

[2] D. Balenson, D. McGrew, and A. Sherman. *Key Management for Large Dynamic Groups: One-way Function Trees and Amortized Initialization*, *INTERNET-DRAFT*, 1999.

[3] J. W. Byers, M. Luby, M. Mitzenmacher, , and A. Rege. A digital fountain approach to reliable distribution of bulk data. In *Proceedings of ACM SIGCOMM '98*, Vancouver, B.C., Sept. 1998.

[4] I. Chang, R. Engel, D. Kandlur, D. Pendarakis, and D. Saha. Key management for secure Internet multicast using boolean function minimization techniques. In *Proceedings of IEEE INFOCOM '99*, volume 2, Mar. 1999.

[5] H. Harney and E. Harder. *Logical Key Hierarchy Protocol*, *INTERNET-DRAFT*, Mar. 1999.

[6] I. R. T. F. (IRTF). Reliable Multicast Research Group. http://www.nard.net/ tmont/rm-links.html.

[7] I. R. T. F. (IRTF). The secure multicast research group (SMuG). http://www.ipmulticast.com/community/smug/.

[8] S. K. Kasera, J. Kurose, and D. Towsley. A comparison of server-based and receiver-based local recovery approaches for scalable reliable multicast. In *Proceedings of IEEE INFOCOM '98*, San Francisco, CA, Mar. 1998.

[9] R. Kermode. Scoped hybrid automatic repeat request with forward error correction (SHARQFEC). In *Proceedings of ACM SIGCOMM '98*, Sept. 1998.

[10] B. Levine and J. Garcia-Luna-Aceves. A comparison of known classes of reliable multicast protocols. In *Proceedings of IEEE ICNP '96*, Columbus, OH, Oct. 1996.

[11] X. S. Li, Y. R. Yang, M. G. Gouda, and S. S. Lam. Batch rekeying for secure group communications. In *Proceedings of Tenth International World Wide Web Conference (WWW10)*, Hong Kong, China, May 2001.

[12] M. J. Moyer, J. R. Rao, and P. Rohatgi. *Maintaining Balanced Key Trees for Secure Multicast*, *INTERNET-DRAFT*, June 1999.

[13] J. Nonnenmacher, E. Biersack, and D. Towsley. Parity-based loss recovery for reliable multicast transmission. In *Proceedings of ACM SIGCOMM '97*, Sept. 1997.

[14] J. Nonnenmacher, M. Lacher, M. Jung, E. Biersack, and G. Carle. How bad is reliable multicast without local recovery? In *Proceedings of IEEE INFOCOM '98*, San Francisco, CA, Mar. 1998.

[15] L. Rizzo. Effective erasure codes for reliable computer communication protocols. *Computer Communication Review*, Apr. 1997.

[16] D. Rubenstein, J. Kurose, and D. Towsley. Real-time reliable multicast using proactive forward error correction. In *Proceedings of NOSSDAV '98*, July 1998.

[17] S. Setia, S. Koussih, S. Jajodia, and E. Harder. Kronos: A scalable group re-keying approach for secure multicast. In *Proceedings of IEEE Symposium on Security and Privacy*, Berkeley, CA, May 2000.

[18] J. Snoeyink, S. Suri, and G. Varghese. A lower bound for multicast key distribution. In *Proceedings of IEEE INFOCOM 2001*, Anchorage, Alaska, Apr. 2001.

[19] D. Towsley, J. Kurose, and S. Pingali. A comparison of sender-initiated reliable multicast and receiver-initiated reliable multicast protocols. *IEEE Journal on Selected Areas in Communications*, 15(3):398–406, 1997.

[20] D. Wallner, E. Harder, and R. Agee. *Key Management for Multicast: Issues and Architectures*, *INTERNET-DRAFT*, Sept. 1998.

[21] C. K. Wong, M. G. Gouda, and S. S. Lam. Secure group communications using key graphs. In *Proceedings of ACM SIGCOMM '98*, Sept. 1998.

[22] C. K. Wong and S. S. Lam. Digital signatures for flows and multicasts. *IEEE/ACM Transactions on Networking*, Aug. 1999.

[23] C. K. Wong and S. S. Lam. Keystone: a group key management system. In *Proceedings of ICT 2000*, Acapulco, Mexico, May 2000.

[24] Y. R. Yang. A secure group key management communication lower bound. Technical Report TR–00–24, The University of Texas at Austin, July, Revised September 2000.

[25] Y. R. Yang, X. S. Li, X. B. Zhang, and S. S. Lam. Reliable group rekeying: A performance analysis. Technical Report TR–01–21, The University of Texas at Austin, June 2001.

[26] X. B. Zhang, S. S. Lam, D.-Y. Lee, and Y. R. Yang. Protocol design for scalable and reliable group rekeying. In *Proceedings of SPIE Conference on Scalability and Traffic Control in IP Networks*, Denver, CO, Aug. 2001.