Protocol Design for Scalable and Reliable Group Rekeying [*]

X. Brian Zhang, Simon S. Lam, Dong-Young Lee, and Y. Richard Yang

Department of Computer Sciences
The University of Texas at Austin
Austin, TX 78712-1188
{zxc, lam, dylee, yangyang}@cs.utexas.edu

TR-02-29 June 2002
Revised, November 2002

Abstract

We present the design and specification of a scalable and reliable protocol for group rekeying together with performance evaluation results. The protocol is based upon the use of key trees for secure groups and periodic batch rekeying. At the beginning of each rekey interval, the key server sends a rekey message to all users consisting of encrypted new keys (encryptions, in short) carried in a sequence of packets. We present a scheme for identifying keys, encryptions, and users, and a key assignment algorithm that ensures that the encryptions needed by a user are in the same packet. Our protocol provides reliable delivery of new keys to all users eventually. It also attempts to deliver new keys to all users with a high probability by the end of the rekey interval. For each rekey message, the protocol runs in two steps: a multicast step followed by a unicast step. Proactive FEC multicast is used to reduce delivery latency. Our experiments show that a small FEC block size can be used to reduce encoding time at the server without increasing server bandwidth overhead. Early transition to unicast, after at most two multicast rounds, further reduces the worst-case delivery latency as well as user bandwidth requirement. The key server adaptively adjusts the proactivity factor based upon past feedback information; our experiments show that the number of NACKs after a multicast round can be effectively controlled around a target number. Throughout the protocol design, we strive to minimize processing and bandwidth requirements for both the key server and users.

keywords: group key management, reliable multicast, secure multicast, proactive FEC

1 Introduction

Many emerging Internet applications, such as pay-per-view distribution of digital media, restricted teleconferences, multi-party games, and virtual private networks will benefit from using a secure group communications model [10]. In this model, members of a group share a symmetric key, called *group key*, which is known only to group users and the key server. The group key can be used for encrypting data traffic between group members or restricting access to resources intended for group members only. The group key is distributed by a group key management system, which changes the group key from time to time (called group rekeying). It is desirable that the group key changes after a new user has joined (so that the new user will not be able to decrypt past group

[*]Research sponsored in part by NSF grant no. ANI-9977267 and NSA INFOSEC University Research Program grant no. MDA904-98-C-A901.

communications) or an existing user has departed (so that the departed user will not be able to access future group communications).

A group key management system has three functional components: registration, key management, and rekey transport [25]. All three components can be implemented in a key server. However, to improve registration scalability, it is preferable to use one or more trusted registrars to offload user registration from the key server [25].

When a user wants to join a group, the user and registration component mutually authenticate each other using a protocol such as SSL [6]. If authenticated and accepted into the group, the new user receives its ID and a symmetric key, called the user's *individual key*, which it shares only with the key server. Authenticated users send join and leave requests to the key management component, which validates the requests by checking whether they are encrypted by individual keys. The key management component also generates rekey messages, which are sent to the rekey transport component for delivery to all users in the group. To build a scalable group key management system, it is important to improve the efficiency of the key management and rekey transport components.

We first consider the key management component, which has been the primary focus of prior work [23, 24, 8, 4, 1, 26]. In this paper, we follow the *key tree* approach [23, 24], which uses a hierarchy of keys to facilitate group rekeying, reducing the processing time complexity of each leave request from $O(N)$ to $O(\log_d(N))$, where N is group size and d the key tree degree. Rekeying after every join or leave request, however, can still incur a large server processing overhead. Thus we proposed to further reduce processing overhead by using periodic rekeying [21, 14, 26], such that the key server processes the join and leave requests during a rekey interval as a batch, and sends out just one rekey message per rekey interval to users. Batch rekeying reduces the number of computationally expensive signing operations. It also reduces substantially bandwidth requirements of the key server and users.

We next consider the rekey transport component. Reliable delivery of rekey messages has not had much attention in prior work. In our prototype system, Keystone [25], we designed and implemented a basic protocol that uses proactive forward error correction (FEC) to improve the reliability of multicast rekey transport. We also investigated the performance issues of rekey transport [26] and observed that although many reliable multicast protocols have been proposed and studied in recent years [9, 18, 5, 22, 13, 16, 11, 17], rekey transport differs from conventional reliable multicast problems in a number of ways. In particular, rekey transport has the following requirements:

- Reliability requirement. It is required that every user will receive all of its (encrypted) new keys, no matter how large the group size. This requirement arises because the key server uses some keys for one rekey interval to encrypt new keys for the next rekey interval. Each user however does not have to receive the entire rekey message because it needs only a very small subset of all the new keys.

- Soft real-time requirement. It is required that the delivery of new keys to all users be finished with a high probability before the start of the next rekey interval. This requirement arises because a user needs to buffer encrypted data and keys before the arrival of encrypting keys, and we would like to limit the buffer size.

- Scalability requirement. The processing and bandwidth requirements of the key server and each user should increase as a function of group size at a low rate such that a single server is able to support a large group. [1]

The objective of this paper is to present in detail our rekey transport protocol as well as its performance. In particular, we have the following contributions. First, a new marking algorithm for batch rekeying is presented.

[1]To further increase system reliability as well as group size, we might consider the use of multiple servers, which is a topic beyond the scope of this paper.

Second, a key identification scheme, key assignment algorithm, and block ID estimation algorithm are presented and evaluated. Third, we show that a fairly small FEC block size can be used to reduce encoding time at the server without increasing server bandwidth overhead. Lastly, an adaptive algorithm to adjust the proactivity factor is proposed and evaluated. The algorithm is found to be effective in controlling the number of NACKs and reducing delivery latency. (Another adaptive algorithm with further refinements is presented in a recent technical report [28].)

Our server protocol for each rekey message consists of four phases: (i) generating a sequence of packets containing encrypted keys (called ENC packets), (ii) generating packets containing FEC redundant information (called $PARITY$ packets), (iii) multicast of ENC and $PARITY$ packets, and (iv) transition from multicast to unicast.

To achieve reliability, our protocol runs in two steps: a multicast step followed by a unicast step. During the multicast step, which typically lasts for just one or two rounds, almost all of the users will receive their new keys because each user only needs one specific packet (guaranteed by our key assignment algorithm) and proactive FEC is also used. Subsequently, for each user who cannot recover its new keys in the multicast step, the keys are sent to the user via unicast. Since each user only needs a small number of new keys, and there are few users remaining in the unicast step, our protocol achieves reliability with a small bandwidth overhead.

To meet the soft real-time requirement, proactive FEC in the multicast step is used to reduce delivery latency [12, 20]. When needed, early transition from multicast to unicast reduces worst-case delivery latency because the server does not need to wait for the maximum round-trip time (RTT) for all users before sending in the unicast step. By adaptively adjusting the time to switch to unicast, our protocol allows explicit tradeoff between key server bandwidth overhead and worst-cast delivery latency.

Towards a scalable design, we observe that the key factors are processing and bandwidth requirements at the key server and each user. To improve scalability, we use the following ideas: 1) To reduce the key server processing requirement, we partition a rekey message into blocks to reduce the size of each block and therefore reduce the key server's FEC encoding time; 2) To reduce each user's processing requirement, our key assignment algorithm assigns encrypted new keys such that each user needs only one packet. Thus, the vast majority of users do not need to recover their specific packets through FEC decoding; 3) To reduce key server bandwidth requirement, our protocol uses multicast to send new keys to users initially; 4) To reduce a user's bandwidth requirement, we use unicast for each user who cannot recover its new keys during the multicast step. This way, a small number of users in high-loss environments will not cause our protocol to perform multicast to all users.

The balance of this paper is organized as follows. In Section 2, we briefly review the ideas of key tree and periodic batch rekeying. In Section 3 we present our server and user protocols. In Section 4 we show how to construct a rekey message. The key identification scheme and key assignment algorithm are presented. Block partitioning and block ID estimation are presented and evaluated in Section 5. In section 6 we discuss how to adaptively adjust the proactivity factor to achieve low delivery latency with a small bandwidth overhead. In Section 7 we discuss when and how to unicast. Our conclusions are in Section 8.

2 Background

We review in this section the ideas of key tree [23, 24] and periodic batch rekeying [21, 14, 26] and present a new marking algorithm. The algorithm is used to update the key tree and generate workload for rekey transport.

2.1 Key tree

A key tree is a rooted tree with the group key as root. A key tree contains two types of nodes: *u-nodes* containing users' individual keys, and *k-nodes* containing the group key and auxiliary keys. A user is given the individual key contained in its u-node as well as the keys contained in the k-nodes on the path from its u-node to the root.

Consider a group with nine users. An example key tree is shown in Figure 1. In this group, user u_9 is given the three keys on its path to the root: k_9, k_{789}, and k_{1-9}. Key k_9 is the *individual key* of u_9, key k_{1-9} is the *group key* that is shared by all users, and k_{789} is an auxiliary key shared by u_7, u_8, and u_9.

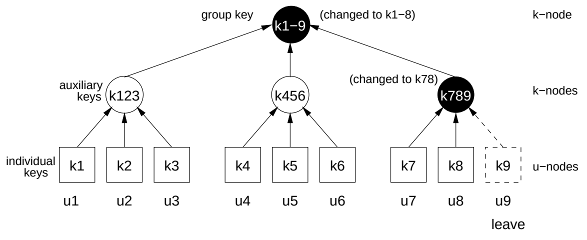


Figure 1: An example key tree

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 new keys to the remaining users using the group-oriented rekeying strategy [24], 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 *encryption*. Upon receiving a rekey message, a user extracts the encryptions that it needs. For example, u_7 only needs $\{k_{1-8}\}_{k_{78}}$ and $\{k_{78}\}_{k_7}$. In other words, a user does not need to receive all of the encryptions in a rekey message.

2.2 Periodic batch rekeying

Rekeying after every join or leave request, however, can be expensive. In periodic batch rekeying, the key server first collects J join and L leave requests during a rekey interval. At the end of the rekey interval, the key server runs a marking algorithm to update the key tree and construct a rekey subtree. The marking algorithm, presented in Appendix B. is different from the one presented in our previous papers [26, 14].

In the marking algorithm, the key server first modifies the key tree to satisfy the leave and join requests. The u-nodes for departed users are removed or replaced by u-nodes for newly joined users. If $J > L$, the key server will split nodes after the rightmost k-node at the highest level (with the root at level 0, the lowest) to accommodate the extra joins. After modifying the key tree, the key server changes the key in a k-node if the k-node is on the path from a changed u-node (either removed or newly joined node) to the root.

Next, the key server constructs a rekey subtree. A *rekey subtree* consists of all of the k-nodes whose keys have been updated in the key tree, the direct children of the updated k-nodes, and the edges connecting updated k-nodes with their direct children. Given a rekey subtree, the key server can then generate encryptions. In particular, for each edge in the rekey subtree, the key server uses the key in the child node to encrypt the key in the parent node.

3 Protocol Overview

In this section, we give an overview of the rekey transport protocol. An informal specification of the key server protocol is shown in Figure 2. Notation used in this paper is defined in Table 1.

First the key server constructs a rekey message as follows. At the beginning of a rekey interval, after the marking algorithm has generated encryptions, the key server runs the key assignment algorithm to assign the encryptions into ENC packets.[2] Our key assignment algorithm guarantees that each user needs only one ENC packet.

[2]An ENC packet is a protocol message generated in the application layer. But we will refer to it as a *packet* to conform to terminology in the literature.

symbol	*description*
d	degree of a key tree
J	number of join requests in a rekey interval
L	number of leave requests in a rekey interval
m	node ID in an expanded key tree
N	number of existing users
n_k	largest k-node ID
a	number of $PARITY$ packets requested by a user in its NACK
$amax[i]$	largest number of $PARITY$ packets requested by users for block i
$numNACK$	target number of NACKs for the first multicast round
h	number of proactive $PARITY$ packets for each block
k	block size; also denotes a key when it appears in $\{k'\}_k$
ρ	proactivity factor, defined as $(h+k)/k$
α	percentage of high loss rate receivers
p	loss rate
p_h	high loss rate
p_l	low loss rate
p_s	loss rate at the source link

Table 1: Notation

Next, the key server uses a Reed-Solomon Erasure (RSE) coder to generate FEC redundant information, called $PARITY$ packets. In particular, the key server partitions ENC packets into multiple blocks. Each block contains k ENC packets. We call k the block size. The key server generates h $PARITY$ packets for each block. We define the ratio of $(h+k)/k$ as *proactivity factor*, denoted by ρ.

Then the key server multicasts the ENC and $PARITY$ packets to all users. A user can recover its required encryptions in any one of the following three cases: 1) The user receives the specific ENC packet that contains all of the encryptions for the user. 2) The user receives at least k packets from the block that contains its specific ENC packet, and thus the user can recover the k original ENC packets. 3) The user receives a USR packet during a subsequent unicast phase. The USR packet contains all of the encryptions needed by the user.

After multicasting ENC and $PARITY$ packets to users, the server waits for the duration of a round, which is typically larger than the maximum round-trip time over all users, and collects NACKs from the users. Based on the NACKs, the key server adaptively adjusts the proactivity factor to control the number of NACKs for the next rekey message. Each NACK specifies the number of $PARITY$ packets that a user needs in order to have k packets to recover its block. In particular, the key server collects the largest number of $PARITY$ packets needed (denoted as $amax[i]$) for each block i. At the beginning of the next round, the key server generates $amax[i]$ new $PARITY$ packets for each block i, and multicasts the new $PARITY$ packets to the users. This process repeats until the conditions for switching to unicast are satisfied (see Section 7). Typically, unicast will start after one or two multicast rounds. During unicast, the key server sends USR packets to those users who have not recovered their required encryptions.

An informal specification of the user protocol is shown in Figure 3. In our protocol, a NACK-based feedback mechanism is used because the vast majority of users can receive or recover their required encryptions within a single round. In particular, during each round, a user checks whether it has received its specific ENC packet or can recover its block. If not, the user will report a, the number of $PARITY$ packets needed to recover its block, to the key server. By the property of Reed-Solomon encoding, a is equal to k minus the number of

```
1. use key assignment algorithm to construct $ENC$ packets
2. partition the sequence of $ENC$ packets into blocks
3. multicast $k$ $ENC$ packets and $h$ $PARITY$ packets for each block
4. when timeout
5.    do adaptively adjust proactivity factor
6.         if conditions for switching to unicast hold
7.            then unicast $USR$ packets to users who did not receive their required encryptions
8.            else  collect $amax[i]$ as the largest number of $PARITY$ packets needed for each block $i$
9.                  generate $amax[i]$ new $PARITY$ packets for each block $i$
10.                 multicast these $PARITY$ packets to all users at the beginning of next round
```

Figure 2: Basic protocol for key server

```
1. when timeout
2.    do if received its specific $ENC$ packet, or at least $k$ packets in the required block, or a $USR$ packet
3.          then retrieve required encryptions
4.          else   $a \leftarrow$ number of $PARITY$ packets needed for recovery
5.                 send $a$ by NACK to the key server
```

Figure 3: Basic protocol for a user

packets received in the block containing its specific ENC packet.

In summary, our protocol uses four types of packets: 1) ENC packet, which contains encryptions for a set of users; 2) $PARITY$ packet, which contains FEC redundant information produced by a RSE coder; 3) USR packet, which contains all of the encryptions for a specific user; 4) $NACK$ packet, which is feedback from a user to the key server. This type of packets reports the number of $PARITY$ packets needed for specific blocks.

Note that protocols given in Figure 2 and 3 only outline the behaviors of the key server and users. More detailed specifications of these protocols and packet formats are shown in Appendix A.

4 Construction of ENC Packets

After running the marking algorithm to generate the encryptions of a rekey message, the key server next runs a key assignment algorithm to assign the encryptions into ENC packets. To increase the probability for each user to receive its required encryptions within one round, our key assignment algorithm guarantees that all of the encryptions for a given user are assigned into a single ENC packet. For each user to identify its specific ENC packet and extract its encryptions from the ENC packet, the key server assigns a unique ID for each key, user, and encryption; such ID information is included in ENC packets.

Below, we first discuss how to assign an ID for each key, user, and encryption; then we define the format of an ENC packet. Lastly we present and evaluate our key assignment algorithm.

4.1 Key identification

To uniquely identify each key, the key server assigns an integer as the ID of each node on a key tree. In particular, the key server first expands the key tree to make it full and balanced by adding null nodes, which we refer to as *n-nodes*. As a result of the expansion, the key tree contains three types of nodes: u-nodes containing individual keys, k-nodes containing the group key and auxiliary keys, and n-nodes. Then the key server traverses the expanded key tree in a top-down and left-right order, and sequentially assigns an integer as a node's ID. The ID starts from 0 and increments by 1. For example, the root node has an ID of 0, and its leftmost child has an ID of 1. Figure 4 (left) illustrates the IDs of nodes in an expanded key tree with a tree degree of three.

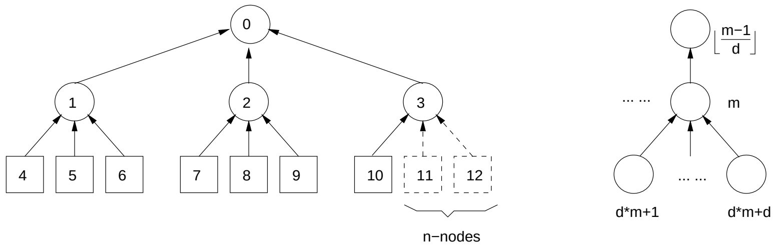


Figure 4: Illustration of key identification

Given the key identification strategy, we observe that the IDs of a node and its parent node have the following simple relationship: If a node has an ID of m, its parent node will have an ID of $\lfloor \frac{m-1}{d} \rfloor$, where d is the key tree degree. Figure 4 (right) illustrates the relationship.

To uniquely identify an encryption $\{k'\}_k$, we assign the ID of the encrypting key k as the ID of this encryption because the key in each node will be used at most once to encrypt another key. Since k' is the parent node of k, its ID can be easily derived given the ID of the encryption.

The ID of a user is, by definition, the ID of its individual key. Given the ID of an encryption and the ID of a user, by the simple relationship between a node and its parent node, a user can easily determine whether the encryption is encrypted by a key that is on the path from the user's u-node to the tree root.

When users join and leave, our marking algorithm may modify the structure of a key tree, and thus the IDs of some nodes will be changed. For a user to determine the up-to-date ID of its u-node, a straightforward approach is for the server to inform each user its new ID by sending a packet to the user. This approach, however, is obviously not scalable. By Lemma 1 and Theorem 1, we show that by knowing the maximum ID of the current k-nodes, each user can derive its new ID independently.

Lemma 1 *If the key server uses the marking algorithm in Appendix B for tree update, then in the updated key tree, the ID of any k-node is always less than the ID of any u-node.*

Theorem 1 *For any user, let m denote the user's ID before the key server runs the marking algorithm, and m' denote its ID after the key server finishes the marking algorithm. Let n_k denote the maximum k-node ID after the key server finishes the marking algorithm. Define function $f(x) = d^x m + \frac{1-d^x}{1-d}$ for integer $x \geq 0$, where d is the key tree degree. Then there exists one and only one integer $x' \geq 0$ such that $n_k < f(x') \leq d \cdot n_k + d$. And m' is equal to $f(x')$.*

A proof is shown in Appendix C. By Theorem 1, we know that a user can derive its current ID by knowing its old ID and the maximum ID of the current k-nodes.

4.2 Format of ENC packets

Given the results in subsection 4.1, we can now define the format of an ENC packet. As shown in the Figure 5, an ENC packet has eight fields, and contains both ID information and encryptions.

The ID information in an ENC packet allows a user to identify the packet, extract its required encryptions, and update its user ID (if changed). In particular, Fields 1 to 5 uniquely identify a packet. A flag bit in Field 2 specifies whether this packet is a duplicate; this field will be further explained in Section 5. Field 6

1. Type: ENC (3 bits)	2. Flag bit (1 bit)
3. Rekey message ID (12 bits)	4. Block ID (8 bits)
5. Sequence number within a block (8 bits)	6. $maxKID$ (16 bits)
7. $<frmID, toID>$ (32 bits)	8. A list of <encryption, ID> (variable length)
9. Padding (variable length)	

Figure 5: Format of an ENC packet

is the maximum ID of the current k-nodes. As we discussed in the previous subsection, each user can derive its current ID based upon this field and its old ID. Field 7 specifies that this ENC packet contains only the encryptions for users whose IDs are in the range of $<frmID, toID>$ inclusively.

Field 8 of an ENC packet contains a list of encryption and its ID pairs. After the encryption payload, an ENC packet may be padded by zero to have fixed length because FEC encoding requires fixed length packets. We observe that padding by zero will not cause any ambiguity because no encryption has an ID of zero.

4.3 User-oriented Key Assignment algorithm

Given the format of an ENC packet, we next discuss the details of our key assignment algorithm, which we refer to as the User-oriented Key Assignment (UKA) algorithm. UKA guarantees that all of the encryptions for a user are assigned into a single ENC packet.

Figure 6 (left) illustrates a particular run of the UKA algorithm in which seven ENC packets are generated. UKA first puts all of the user IDs into a list in increasing order. Then, a longest prefix of the list is extracted such that all of the encryptions needed by the users in this prefix will fill up an ENC packet. Repeatedly, UKA generates a sequence of ENC packets whose $<frmID, toID>$ intervals do not overlap. In particular, the algorithm guarantees that $toID$ of a previous ENC packet is less than the $frmID$ of the next packet. This property is useful for block ID estimation to be performed by a user.

4.4 Performance of UKA

UKA assigns all of the encryptions for a user into a single ENC packet, and thus significantly increases the probability for a user to receive its encryptions in a single round. Consequently, the number of NACKs sent to the key server is reduced.

This benefit, however, is achieved at an expense of sending duplicate encryptions. In a rekey subtree, users may share encryptions. For two users whose encryptions are assigned into two different ENC packets, their shared encryptions are duplicated in these two ENC packets; therefore, we expect that UKA would increase the bandwidth overhead at the key server.

We evaluate the performance of UKA in this subsection using simulations. In the simulations, we assume that at the beginning of a rekey interval the key tree is full and balanced with N u-nodes. During the rekey interval, J join and L leave requests are processed. We further assume that the leave requests are uniformly distributed over the u-nodes. We set the key tree degree d as 4 and the length of an ENC packet as 1028 bytes. In all of our experiments in this paper, each average value is computed based on at least 100 simulation runs.

We first investigate the size of a rekey message as a function of J and L for $N = 4096$, as shown in Figure 6 (middle). For a fixed L, we observe that the average number of ENC packets increases linearly with J. For a fixed J, we observe that as L increases, the number of ENC packets first increases (because more leaves imply more keys to be changed), and then decreases (because now some keys can be pruned from the rekey subtree).

Next we investigate the size of a rekey message as a function of N, as shown in Figure 6 (right). We observe that the average number of ENC packets in a rekey message increases linearly with N for three combinations

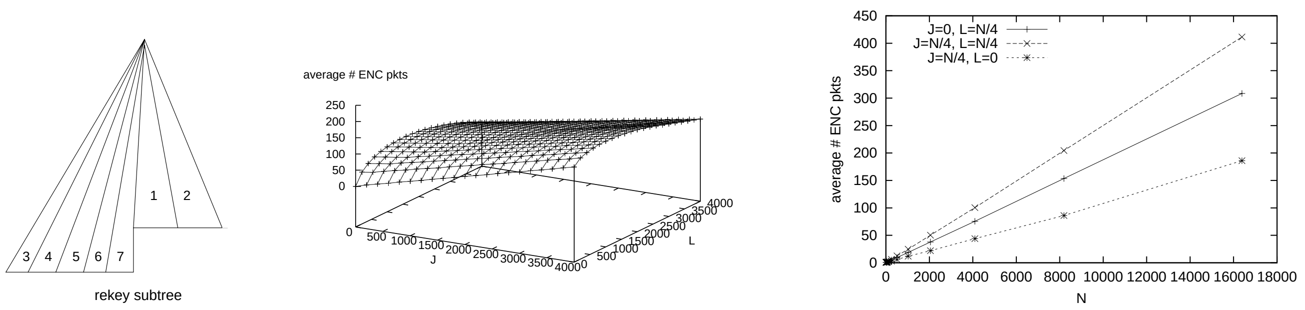


Figure 6: Illustration of UKA algorithm (left), average number of ENC packets as a function of J and L for $N = 4096$ (middle), and as a function of N (right)

of J and L values.

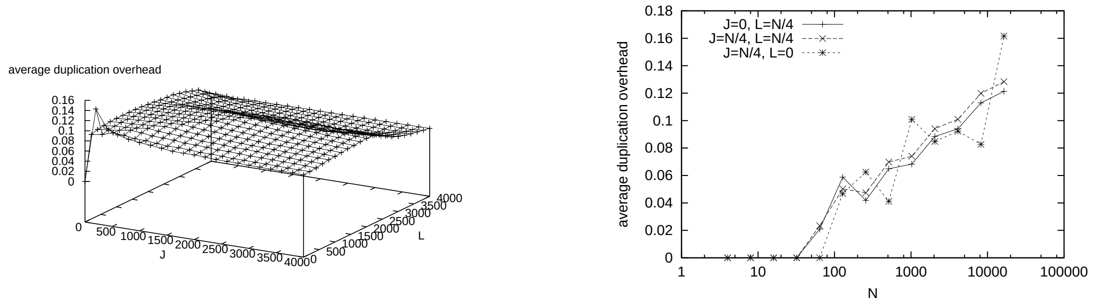


Figure 7: Average duplication overhead as a function of J and L for $N = 4096$ (left) and as a function of N(right)

With the UKA algorithm, some encryptions are duplicated in ENC packets. We define *duplication overhead* as the ratio of duplicated encryptions to the total number of encryptions in a rekey subtree. Figure 7 (left) shows the average duplication overhead as a function of J and L for $N = 4096$. First consider the case of a fixed L. We observe that the duplication overhead decreases from about 0.1 to 0.05 as we increase J. Next consider the case of a fixed J. We observe that the duplication overhead first increases and then decreases as we increase L.

Last, we plot in Figure 7 (right) the average duplication overhead as a function of N. We observe that for $J = 0, L = N/4$ or $J = L = N/4$, the average duplication overhead increases approximately linearly with $\log(N)$ for $N \geq 32$. This is because the rekey subtree is almost full and balanced for $J = 0, L = N/4$ or $J = L = N/4$, and thus the duplication overhead is directly related to the tree height $\log_d(N)$. We also observe that the duplication overhead is generally less than $\frac{\log_d(N)-1}{46}$, where 46 is the number of encryptions that can be carried in an ENC packet with a packet size of 1028 bytes. For $J = N/4, L = 0$, the rekey subtree is very sparse, and thus the graph of duplication overhead fluctuates around the graph of $J = L = N/4$.

5 Block Partitioning

After running UKA assignment algorithm to generate the ENC packets of a rekey message, the key server next generates $PARITY$ packets for the ENC packets using a Reed-Solomon Erasure (RSE) coder.

Although grouping all of the ENC packets into a single RSE block may reduce server bandwidth overhead, a large block size can significantly increase encoding and decoding time [19, 3, 16]. For example, using the RSE coder of Rizzo [19], the encoding time for one $PARITY$ packet is approximately a linear function of block size. Our evaluation shows that for a large group, the number of ENC packets generated in a rekey interval can be large. For example, for a group with 4096 users, when $J = L = N/4$, the key server can generate up to 128 ENC packets with a packet size of 1028 bytes. Given such a large number of ENC packets in a rekey interval, it is necessary to partition the ENC packets into multiple blocks in order to reduce the key server's encoding time.

Consider the ENC packets of a rekey message sequenced in order of generation by UKA algorithm. The packet sequence is partitioned into blocks of k packets, with the first k packets forming the first block, the next k packets forming the second block, and so on. Each block formed is assigned sequentially an integer-valued block ID. Each packet within a block is assigned a sequence number from 0 to k-1.

To form the last block, the key server may need to duplicate ENC packets in the last block until there are k packets. (Alternatively, the key server may distribute the required number of duplicates over several blocks.) We use a flag bit in each ENC packet to specify whether the packet is a duplicate, as shown in Figure 5. A duplicate ENC packet has the same contents in all fields as the original packet except for the sequence number field and the flag bit field. A new sequence number is assigned to each duplicate ENC packet because Reed-Solomon encoding needs to uniquely identify every packet in a block, duplicate or not. At a user, duplicate packets are used in Reed-Solomon decoding; however, they are not used for block ID estimation (see Appendix D).

5.1 Block ID estimation

One issue that arises from partitioning ENC packets into blocks is that if a user loses its specific ENC packet, the user will not be able to determine directly the block to which its ENC packet belongs. As a result, the user needs to estimate the block ID to which its specific ENC packet belongs. We present an algorithm for users to estimate its block ID in Appendix D. With this algorithm, the probability that a user cannot determine the precise value of its block ID is no more than p^2 in the worst case, where p is the loss rate observed by the user under the assumption of independent packet loss. When this happens, the user can still estimate a possible range of its block ID. It will then request $PARITY$ packets for every block within this range when it sends a NACK.

5.2 Packets sent in interleaving pattern

After forming the blocks of a rekey message, the key server generates $PARITY$ packets, and multicasts all ENC and $PARITY$ packets to users. One remaining issue is to determine an order in which the key server sends these packets. In our protocol, the key server sends packets of different blocks in an interleaving pattern. By interleaving packets from different blocks, two packets from the same block will be separated by a larger time interval, and thus are less likely to experience the same burst loss on a link. With interleaving, our evaluation shows that the bandwidth overhead at the key server can be reduced.

5.3 Choosing block size

Block partitioning is carried out for a given block size k. To determine the block size, we need to evaluate the impact of block size on two performance metrics.

The first performance metric is the key server's multicast bandwidth overhead, which is defined to be the ratio of v' to v, where v is the number of ENC packets in a rekey message, and v' is the total number of packets that the key server multicasts to enable recovery of specific ENC packets by all users. To evaluate the bandwidth overhead, we consider the impact of block size k for $\rho = 1$ only (that is, $h = 0$) in this section. The extra bandwidth overhead due to an adaptive ρ, for $\rho > 1$, will be evaluated in Section 6. Such extra bandwidth was found to be small (see Figure 18).

The second performance metric is overall FEC encoding time, which is the time that the key server spends to generate all of the $PARITY$ packets for a rekey message. Although block size k also has a direct impact on each user's FEC decoding time, the impact is small because in our protocol, the vast majority of users receive their specific ENC packets and thus do not perform any decoding.

We use simulations to evaluate the impact of block size. To support a large group size, we developed our own simulator for a model proposed and used by J. Nonnenmacher, et al. [17]. In this model, the key server connects to a backbone network via a source link, and each user connects to the backbone network via a receiver link. The backbone network is assumed to be loss-free. The source link has a fixed loss rate of p_s. A fraction α of the N users have a high loss rate of p_h, and the others have a low loss rate of p_l. For each given loss rate, p, we use a two-state continuous-time Markov chain [16] to simulate burst loss as follows: the average duration of a burst loss is $\frac{100}{p}$ msec, and the average duration of loss-free time between consecutive loss bursts is $\frac{100}{1-p}$ msec. [3] The default values in our simulations are as follows: $N = 4096$, $d = 4$, $J = L = N/4$, $\alpha = 20\%$, $p_h = 20\%$, $p_l = 2\%$, $p_s = 1\%$, and the key server's sending rate is 10 packets/second, and the length of an ENC packet is 1028 bytes. The same simulation topology and parameter values will also be used in experiments described in the following sections unless otherwise stated.

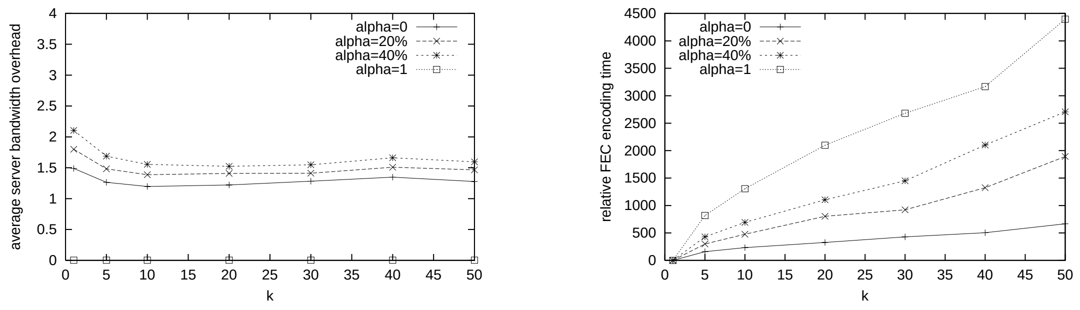


Figure 8: Average server bandwidth overhead (left) and relative overall FEC encoding time (right) as a function of block size

The impact of block size on the key server's bandwidth overhead is shown in Figure 8 (left). Observe that the key server's average bandwidth overhead is not sensitive to the block size k for $k \geq 5$.

We next consider the impact of block size k on the key server's overall FEC encoding time. If we use Rizzo's RSE coder [19], the encoding time for one $PARITY$ packet is approximately a linear function of k. Therefore, the encoding time of all $PARITY$ packets for a rekey message is approximately the product of the total number of $PARITY$ packets and the encoding time for one $PARITY$ packet. By the definition of the key server's bandwidth overhead, we know that the total number of $PARITY$ packets is proportional to the server bandwidth overhead that is almost flat as a function of k for $k \geq 5$. The relative overall encoding time (assuming k time units to generate one $PARITY$ packet for block size k) is shown in Figure 8 (right).

In summary, we found that, for $\rho = 1$, a small block size k can be chosen to enable fast FEC encoding at the server without incurring a large server bandwidth overhead. For experiments in the following sections, we choose $k = 10$ as the default value unless otherwise specified.

[3]The network topology and loss model are simplistic compared to the Internet. They are however needed for simulating a large group size (up to 16384). For simulation results from the use of ns and GT-ITM for a smaller group size, we refer the interested reader to our recent work [28].

6 Adaptive Proactive FEC Multicast

In the previous section, we discussed how to partition the ENC packets of a rekey message into blocks and generate $PARITY$ packets for each block. The discussion, however, assumes a given proactivity factor ρ. In this section, we investigate how to determine ρ.

Proactive FEC has been widely used to improve reliability and reduce delivery latency [20, 7, 16, 12, 2, 27, 15]. However, if the proactivity factor is too large, the key server may incur high bandwidth overhead. On the other hand, if the proactivity factor is too small, the users may have to depend on retransmissions to achieve reliability; thus, the benefit of reduced delivery latency diminishes. Furthermore, if we depend on proactive FEC to avoid feedback implosion and the proactivity factor is too small, many users may experience packet losses and the key server would be overwhelmed by NACKs.

The appropriate proactivity factor will depend on network status, in particular, factors such as network topology, loss rates of network links, number of users in a session, and number of sessions using proactive FEC. Such factors are unknown to the key server and may be changing during a session's life time. The objective of our next investigation, therefore, is to study how to adaptively adjust proactivity factor by observing its impact on the number of NACKs from users. With adaptive adjustment, we aim to achieve low delivery latency with small bandwidth overhead.

6.1 Impact of proactivity factor

To design an algorithm to adjust ρ, we need to first evaluate the impact of ρ on the number of NACKs, the delivery latency at users, and the bandwidth overhead at the key server.

We first evaluate the impact of ρ on the number of NACKs. Figure 9 (left) plots the average number of NACKs received at the key server at the end of the first round. Note that y-axis is in log scale. We observe that the average number of NACKs decreases exponentially as a function of ρ. (A similar observation was made in a previous study [20].)

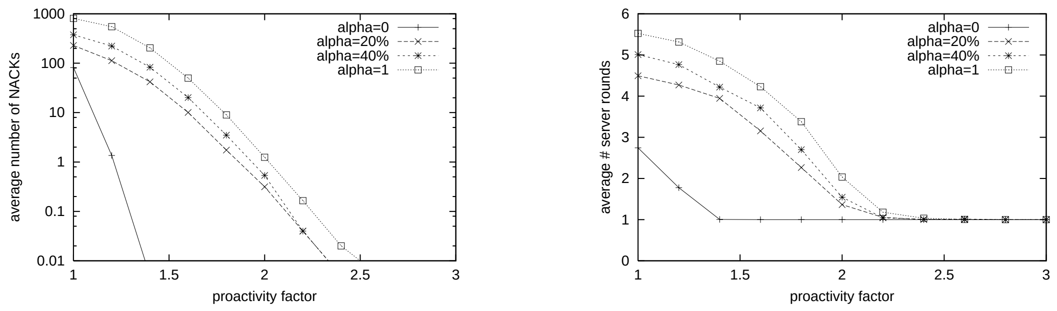


Figure 9: Average number of NACKs in the first round (left) and average number of rounds for all users to receive their encryptions (right) as a function of ρ

ρ	*Percentage of users who need*			
	1 *round*	2 *rounds*	3 *rounds*	≥ 4 *rounds*
1	94.414%	5.134%	0.389%	0.063%
1.2	97.256%	2.502%	0.196%	0.046%
1.6	99.888%	0.090%	0.018%	0.004%
2	99.992%	0.006%	0.001%	0.001%

Table 2: Percentage of users on average who need a given number of rounds to receive their encryptions

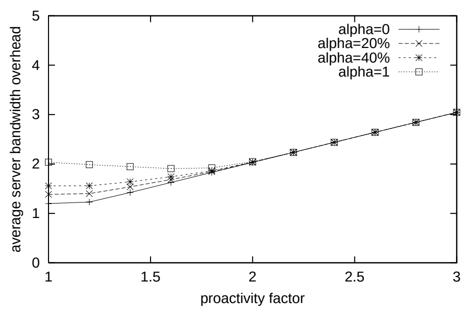


Figure 10: Average server bandwidth overhead as a function of ρ

We next evaluate the impact of ρ on delivery latency. Figure 10 plots the average number of rounds for all users to receive their encryptions. We observe that the average number of rounds decreases almost linearly as we increase ρ, until ρ is large enough and the curve levels off. Table 2 shows the percentage of users on average who need a given number of rounds to receive their encryptions. For $\rho = 1$, we observe that on average 94.41% of the users can receive their encryptions within a single round; for $\rho = 1.6$, the percentage value is increased to 99.89%; for $\rho = 2.0$, the percentage value is increased to 99.99%.

We next evaluate the impact of ρ on the average server bandwidth overhead, as shown in Figure 10. For ρ close to 1, the key server sends a small amount of proactive $PARITY$ packets during the first round, but it needs to send more reactive $PARITY$ packets in subsequent rounds to allow users to recover their packets. As a result, a small increase of ρ has little impact on the average server bandwidth overhead. When ρ becomes large, the bandwidth overhead during the first round dominates the overall bandwidth overhead, and the overall bandwidth overhead increases linearly with ρ.

In summary, we observe that an increase of ρ can have the following three effects: 1) It will significantly reduce the average number of NACKs. 2) It will reduce the worst-case delivery latency. 3) It will increase the key server's bandwidth overhead when ρ is larger than needed.

6.2 Adjustment of proactivity factor

We present in Figure 11 an algorithm to adaptively adjust ρ. The basic idea of the algorithm is to adjust ρ based on NACK information received for the current rekey message so that the key server will receive a target number of NACKs for the next rekey message. The key server runs this algorithm at the end of the first multicast round.

```
    Algorithm $AdjustRho(A)$
$\triangleright$ $A = \{a_i\}$: each item $a_i$ is the number of $PARITY$ packets requested by a user.
1. **if** $(size(A) > numNACK)$
2.    **then** sort $A$ such that $a_0 \geq a_1 \geq a_2, ...$
3.          $h \leftarrow h + a_{numNACK}$
4. **if** $(size(A) < numNACK)$
5.    **then** do $h \leftarrow \max\{0, h-1\}$ with probability $\max\{0, \frac{numNACK - 2 \cdot size(A)}{numNACK}\}$
6. $\rho \leftarrow (h+k)/k$
```

Figure 11: Algorithm to adaptively adjust proactivity factor

The input to algorithm $AdjustRho$ is a list A. Each item in A is the number of $PARITY$ packets requested by a user. If a user requests packets for a range of blocks, the key server records into A the number of $PARITY$

packets requested for the block that contains the user's specific ENC packet.

The algorithm works as follows. For each rekey message, at the end of the first round, the key server compares the number of NACKs it has received, which is equal to $sizeof(A)$, and the number of NACKs it targets (denoted by $numNACK$). The comparison results in two cases.

In the first case, the key server receives more NACKs than its target. For this case, the server selects the $(numNACK+1)^{th}$ largest item (denoted by $a_{numNACK}$) from A, and increases ρ so that $a_{numNACK}$ additional proactive $PARITY$ packets will be generated for each block of the next rekey message. To illustrate, suppose 10 users, u_i, $i = 0, ..., 9$, have sent NACKs for the current rekey message, and user u_i requests a_i $PARITY$ packets. For illustration purposes, we assume $a_0 \geq a_1 \geq ... \geq a_9$, and the target number of NACKs is 2, that is, $numNACK = 2$. Then according to our algorithm, for the next rekey message, the key server will send a_2 additional $PARITY$ packets so that users $\{u_2, u_3, ..., u_9\}$ have a higher probability to recover their ENC packets within a single round. This is because according to the current rekey message, if users $\{u_2, u_3, ..., u_9\}$ were to receive a_2 more $PARITY$ packets, they could have recovered their ENC packets within a single round.

In the second case, the key server receives less NACKs than its target. Although receiving less NACKs is better in terms of reducing delivery latency, the small number of NACKs received may mean that the current proactivity factor is too high, and thus may cause high bandwidth overhead. Therefore, the algorithm reduces ρ by one $PARITY$ packet with probability equal to $\frac{numNACK - 2 \cdot size(A)}{numNACK}$.

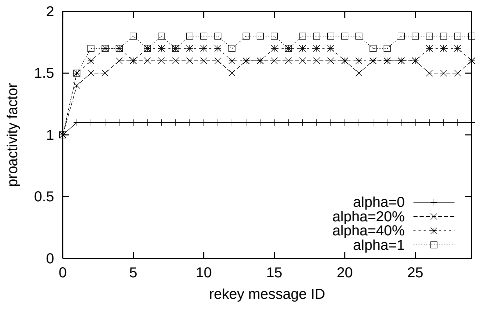
6.3 Performance evaluation

We use simulations to evaluate algorithm $AdjustRho$. In Section 6.3.1, we evaluate the performance of our algorithm in terms of controlling the number of NACKs. In Section 6.3.2, we investigate how to choose block size k for the adaptive ρ scenario. In Section 6.3.3, we evaluate the server bandwidth overhead for adaptive proactive FEC.

6.3.1 Controlling the number of NACKs

Before evaluating whether algorithm $AdjustRho$ can control the number of NACKs, we first investigate the stability of the algorithm. For the simulations in this section, we set the target number of NACKs ($numNACK$) at 20.

Figure 12 shows how ρ is adaptively adjusted when the key server sends a sequence of rekey messages. For initial $\rho = 1$ as shown in the left figure, we observe that it takes only two or three rekey messages for ρ to settle down to stable values. For initial $\rho = 2$ as shown in the right figure, we observe that ρ keeps decreasing until it reaches stable values. Comparing both figures, we note that the stable values of these two figures match each other very well.



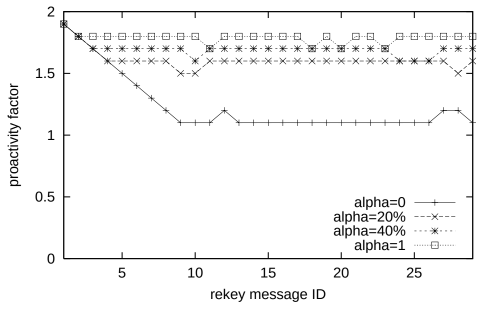
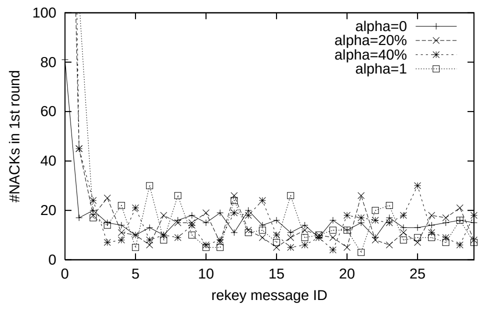


Figure 12: Traces of proactivity factor with initial $\rho = 1$ (left) and initial $\rho = 2$ (right)



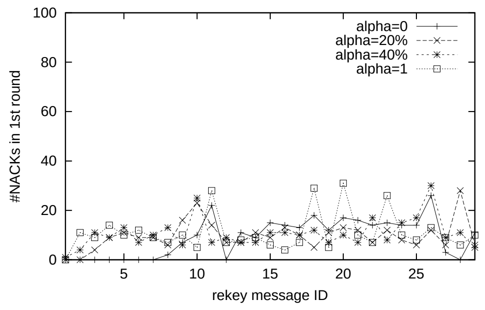
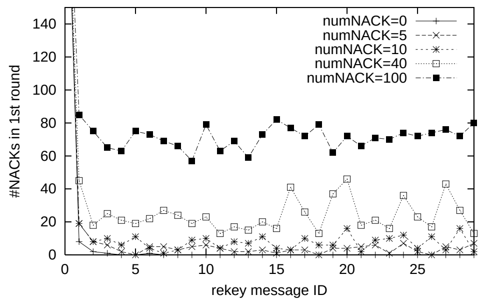


Figure 13: Traces of the number of NACKs received with initial $\rho = 1$ (left) and initial $\rho = 2$ (right)

Next we consider the number of NACKs received by the key server at the end of the first round for each rekey message, as shown in Figure 13. In the left figure where the initial ρ value is 1, the number of NACKs received stabilizes very quickly, and the stable values are generally less than 1.5 times of $numNACK$. The right figure shows the case for initial $\rho = 2$. We observe that the stable values of these two figures match very well.



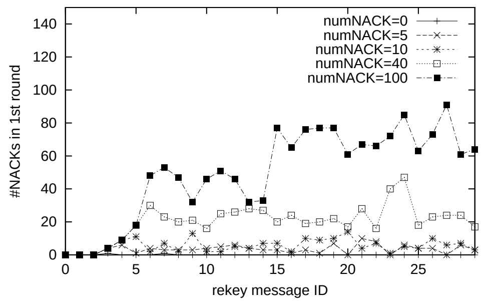


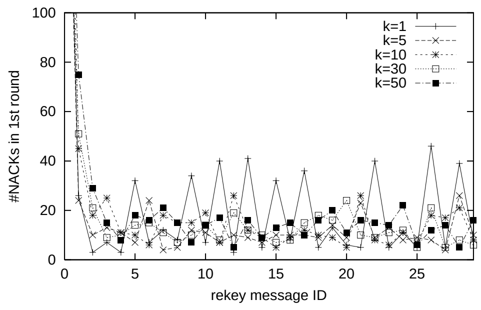
Figure 14: Traces of the number of NACKs for various $numNACK$ values with initial $\rho = 1$ (left) and initial $\rho = 2$ (right)

Next we evaluate whether algorithm $AdjustRho$ can control the number of NACKs for various values of $numNACK$. As shown in the left figure (initially ρ is 1) and right figure (initially ρ is 2) of Figure 14, the number of NACKs received at the key server fluctuates around each $numNACK$ target value specified. However, we do observe that the fluctuations become more significant for larger values of $numNACK$. Therefore, in choosing $numNACK$, we need to consider the potential impact of large fluctuations when $numNACK$ is large. In the experiments to be presented, we choose 20 to be a fixed default value of $numNACK$ unless otherwise specified.

6.3.2 Choosing block size

In Section 5.3, we have discussed how to choose block size k for $\rho = 1$. In this section, we reconsider this problem for a new scenario where ρ is adaptively adjusted. To determine the block size, we consider the following factors:

- Fluctuations in the number of NACKs received—From Figure 15, we observe that a very small block size may cause large fluctuations in the number of NACKs. For example, for $k = 1$ or 5, the number of NACKs received by the key server can reach as high as two times of $numNACK$.



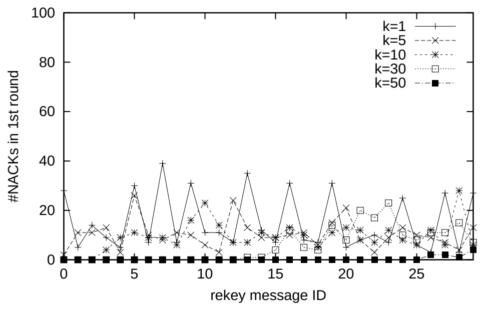
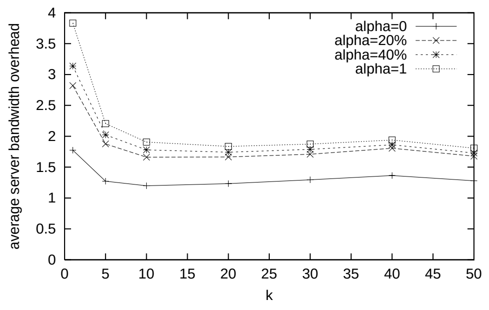


Figure 15: Traces of the number of NACKs for various k values with initial $\rho = 1$ (left) and initial $\rho = 2$ (right)



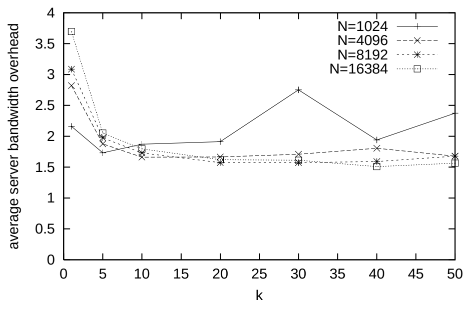
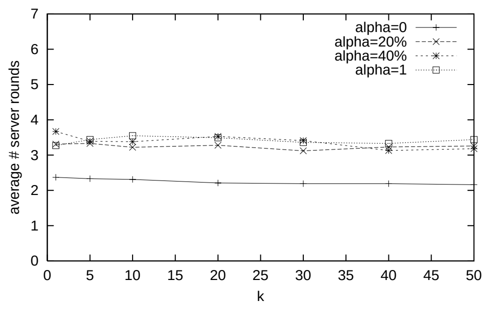


Figure 16: Average bandwidth overhead as a function of block size for various α values (left) and for various N values (right)



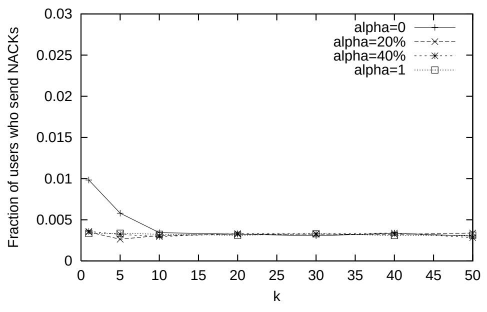


Figure 17: Average number of rounds for all users to receive their encryptions (left) and fraction of users on average who send NACKs in first round (right) as a function of k

- Server bandwidth overhead—First consider Figure 16 (left), which shows the average bandwidth overhead at the key server as a function of k when ρ is adaptively adjusted. [4] We observe that the average server bandwidth overhead is very high for $k = 1$; then it decreases and becomes flat as we increase k. This observation is almost the same as what we see for $\rho = 1$ in Figure 8 (left), except that the bandwidth overhead for $k = 1$ is much higher in the adaptive ρ scenario.

  Next consider Figure 16 (right), which shows the average server bandwidth overhead as a function of k for different group sizes when ρ is adaptively adjusted. The figure shows the same trend as the left figure. Note that the average server bandwidth overhead fluctuates a lot for $N = 1024$. This is because the rekey message contains only 32 ENC packets for $N = 1024$ and $J = L = N/4$. For a large k, the number of

[4]In the case of adaptive FEC, we measure average values based on at least 100 rekey messages after the key server has sent out 10 rekey messages with initial $\rho = 1$.

duplicated packets in the last block can significantly affect the server bandwidth overhead.

- Overall FEC encoding time to generate all $PARITY$ packets in a rekey message—According to our observations in Figure 16, we know that for $k \geq 5$, the server bandwidth overhead is not sensitive to block size. As a result, the overall FEC encoding time at the key server will increase approximately at a linear rate as block size k increases.
- Delivery latency—From Figure 17 (left), we observe that the average number of rounds for all users to receive their encryptions stays flat as we vary block size k; therefore, block size k does not have much impact on the delivery latency at users. To further validate this result, Figure 17 (right) plots the fraction of users on average who send NACKs in the first round. For $k \geq 10$, the curves for different values of α stay flat as block size k increases.

In summary, when ρ is adaptively adjusted, block size k should not be too small in order to avoid large fluctuations in the number of NACKs and a large key server bandwidth overhead. On the other hand, block size k should be small enough to reduce the key server's FEC encoding time. This confirms our previous conclusion for the case of $\rho = 1$.

6.3.3 Overhead of adaptive proactive FEC

From the previous section, we know that algorithm $AdjustRho$ can effectively control the number of NACKs and reduce delivery latency. However, compared with an approach that does not send any proactive $PARITY$ packets at all during the first round (i.e. $\rho = 1$) and only generates reactive $PARITY$ packets during the subsequent rounds, the adaptive proactive FEC scheme may incur extra bandwidth overhead at the key server. We investigate this issue in this subsection.

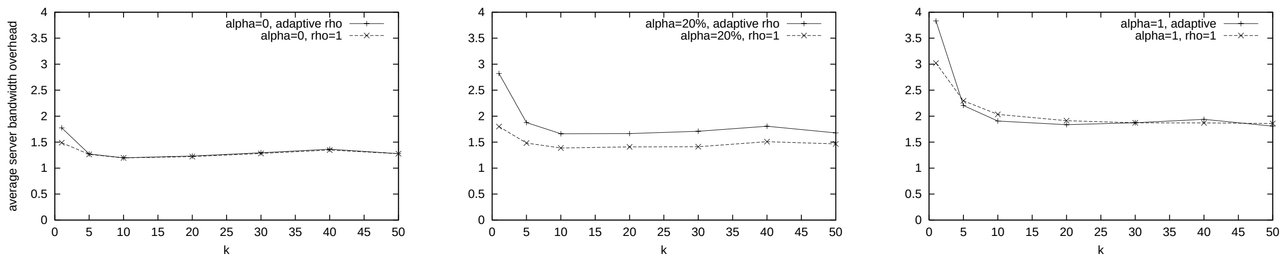


Figure 18: Extra server bandwidth caused by adaptive ρ for different α

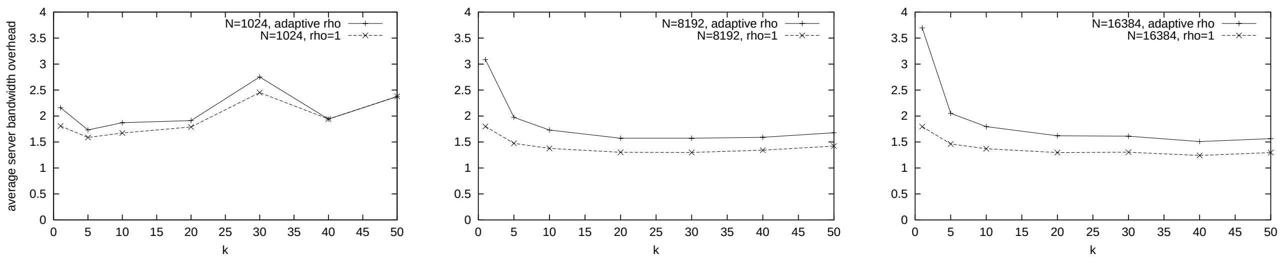


Figure 19: Extra server bandwidth caused by adaptive ρ for different N

We first evaluate the extra bandwidth overhead at the key server caused by proactive FEC. Figure 18 shows the results for three loss environments. We observe that compared with the approach where all $PARITY$ packets are generated reactively, our adaptive proactive scheme causes very little extra server bandwidth overhead in a homogeneous low loss environment (i.e. $\alpha = 0$). For $\alpha = 1$, shown on the right of Figure 18, our scheme can even save a little bandwidth. This is because for $\rho = 1$, the key server takes more rounds for all users to recover their encryptions in the reactive scheme than in the adaptive proactive scheme. Therefore, it is possible that the total number of $PARITY$ packets generated during the rounds for $\rho = 1$ is larger than that of the adaptive proactive scheme. In a heterogeneous environment, such as $\alpha = 20\%$ shown in the middle of Figure 18, the extra bandwidth overhead generated by adaptive ρ is less than 0.3 for $k > 5$.

We next consider the server bandwidth overhead for various group sizes. From Figure 19, we observe that the extra bandwidth overhead incurred by adaptive ρ increases with N. The extra bandwidth overhead, however, is still less than 0.42 even for $N = 16384$ when $k > 5$.

7 Speedup with unicast

Rekeying transport has a soft real-time requirement, that is, it is desired that all users receive their new keys before the end of the rekey interval. To meet this requirement, we have proposed in the previous section to adaptively adjust $numNACK$ and ρ during the multicast phase to reduce the number of users missing deadline. However, these approaches may not guarantee that all users can meet deadline because $numNACK$ does not directly affect delivery latency at users. In fact, from Table 2 we have observed that there may exist a few NACKs after two rounds when $numNACK$ is 20.

To further increase the number of users meeting deadline, the key server will switch to unicast after one or two multicast rounds. Unicast can reduce delivery latency compared to multicast because the duration of a multicast round is typically larger than the maximum round-trip time over all users.

To use unicast, we need to solve two problems. First, we need to determine when to switch to unicast so that unicast will not cause a large bandwidth overhead at the key server. Second, we need to determine how to send the unicast packets so that each user who still needs recovery can receive its USR packet with high probability.

7.1 When to switch to unicast

One issue of early unicast is its possible high bandwidth overhead at the key server. However, in our protocol, unicast will not cause large bandwidth overhead at the key server for the following two reasons. First, the size of a USR packet sent during unicast is much smaller than the size of an ENC or $PARITY$ packet. In our protocol, a USR packet contains only the encryptions for a specific user, and its packet size is at most $(3 + 20 \cdot l)$ bytes, where l is the height of the key tree. On the other hand, the size of an ENC or $PARITY$ packet is typically more than one kilobyte long. Second, our protocol guarantees that only a few users need unicast if $numNACK$ is small enough. In fact, our evaluations show that for $numNACK = 20$, $N = 4096$, and initial $\rho = 1$, roughly 5 or less users need recovery after two multicast rounds when the system becomes stable.

Our conditions for switching to unicast are as follows. Our protocol switches to unicast after one or two multicast rounds. We suggest two multicast rounds for a large rekey interval and one multicast round for a small rekey interval. Even for a large interval, the time to switch to unicast can be earlier if the total length of the USR packets is no more than that of $PARITY$ packets needed for the next multicast round.

7.2 Unicast protocol

Let R denote the set of users who cannot recover their new keys in the multicast step. For each user i in set R, the key server can estimate the loss rate of user i while it receives the current rekey message. Let p_i denote the current loss rate of user i, and we have:

$$p_i = \frac{(k+h) - (k - a_i)}{k+h} = \frac{h + a_i}{k+h}$$

where h is the number of proactive $PARITY$ packets multicasted for each block, and a_i is the number of $PARITY$ packets requested by user i in the latest $NACK$. We observe that p_i reflects the current loss rate of user i based on the fact that i received only $k - a_i$ packets out of $k + h$ packets.

Given loss rate p_i, if the key server unicasts c copies of USR packets to user i, then the probability for user i to receive at least one USR packet is $1 - {p_i}^c$. Therefore, if we want to guarantee that user i can receive its USR packet with a probability larger than or equal to $\mathcal{P}$, the key server should unicast at least $\lceil log_{p_i}(1 - \mathcal{P}) \rceil$ copies of USR packets to i.

However, it is still possible that user i will lose all of the USR packets. In this case, user i will send another NACK to the key server. The key server will first update the value of p_i, and then unicast a new sequence of USR packets to user i. This process repeats until user i receives one USR packet.

8 Conclusion

The objective of this paper is to present in detail our rekey transport protocol as well as its performance. Our server protocol for each rekey message consists of four phases: (i) generating a sequence of ENC packets containing encrypted keys, (ii) generating $PARITY$ packets, (iii) multicast of ENC and $PARITY$ packets, and (iv) transition from multicast to unicast.

In the first phase, after running the marking algorithm to generate encryptions for a rekey message, the key server constructs ENC packets. The major problem in this phase is to allow a user to identify its required encryptions. To solve the problem, first we assign a unique integer ID to each key, user, and encryption. Second, our key assignment algorithm guarantees that each user needs only one ENC packet. By including a small amount of ID information in ENC packets, each user can easily identify its specific ENC packet and extract the encryptions it needs.

In the second phase, the key server uses a RSE coder to generate $PARITY$ packets for ENC packets. The major problem in this phase is to determine the block size for FEC encoding. This is because a large block size can significantly increase FEC encoding and decoding time. Our performance results show that a small block size can be chosen to provide fast FEC encoding without increasing bandwidth overhead. We also present an algorithm for a user to estimate its block ID if it has not received its specific ENC packet.

In the third phase, the key server multicasts both ENC and $PARITY$ packets to all users. This proactive FEC multicast can effectively reduce delivery latency of users; however, a large proactivity factor may increase the server bandwidth overhead. Therefore, the major problem in this phase is how to achieve low delivery latency with small bandwidth overhead. In our protocol, the key server adaptively adjusts the proactivity factor based on past feedback. Our experiments show that the number of NACKs can be effectively controlled around a target number, thus achieving low delivery latency, while the extra bandwidth overhead incurred is small.

In the fourth phase, the key server switches to unicast to reduce the worst-case delivery latency. The problems in this phase are (1) to determine when to switch to unicast such that unicast will not cause large server bandwidth overhead, and (2) how to do unicast to provide small delivery latency. We let the key server switch to unicast after one or two multicast rounds (depending upon deadline). To guarantee that each user who still needs recovery can receive its USR packet with a high probability, the key server estimates the user's loss rate, and sends multiple copies of the user's USR packet during unicast.

In summary, we have the following contributions. First, a new marking algorithm for batch rekeying is presented. Second, a key identification scheme, key assignment algorithm, and block ID estimation algorithm are presented and evaluated. Third, we show that a fairly small FEC block size can be used to reduce encoding time at the server without increasing server bandwidth overhead. Lastly, an adaptive algorithm to adjust the proactivity factor is proposed and evaluated. The algorithm is found to be effective in controlling the number of NACKs and reducing delivery latency. Another adaptive algorithm with further refinements is presented in a recent technical report [28].

ACKNOWLEDGMENTS

We would like to thank Mr. Min S. Kim, Jia Liu, and the anonymous reviewers for their constructive comments.

References

[1] David Balenson, David McGrew, and Alan Sherman. Key management for large dynamic groups: One-way function trees and amortized initialization, INTERNET-DRAFT, 1999.

[2] Jean-Chrysostome Bolot, Sacha Fosse-Parisis, and Don Towsley. Adaptive FEC-based error control for Internet Telephony. In *Proceedings of IEEE INFOCOM '99*, March 1999.

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

[4] Isabella Chang, Robert Engel, Dilip Kandlur, Dimitrios Pendarakis, and Debanjan Saha. Key management for secure Internet multicast using boolean function minimization techniques. In *Proceedings of IEEE INFOCOM '99*, volume 2, March 1999.

[5] S. Floyd, V. Jacobson, C. Liu, S. McCanne, and L. Zhang. A reliable multicast framework for light-weight sessions and application level framing. *IEEE/ACM Transactions on Networking*, 5(6):784–803, 1997.

[6] Alan O. Freier, Philip Karlton, and Paul C. Kocher. The SSL protocol version 3.0. Work in progress, IETF Internet-Draft, March 1996.

[7] M. Handley, S. Floyd, B. Whetten, R. Kermode, L. Vicisano, and M. Luby. The reliable multicast design space for bulk data transfer, RFC 2887, August 2001.

[8] Hugh Harney and Eric Harder. Logical key hierarchy protocol, INTERNET-DRAFT, March 1999.

[9] Internet Research Task Force (IRTF). Reliable Multicast Research Group. http://www.nard.net/ tmont/rm-links.html.

[10] Internet Research Task Force (IRTF). The secure multicast research group (SMuG). http://www.ipmulticast.com/community/smug/.

[11] Sneha K. Kasera, Jim Kurose, and Don 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, March 1998.

[12] Roger G. Kermode. Scoped Hybrid Automatic Repeat reQuest with Forward Error Correction (SHARQFEC). In *Proceedings of ACM SIGCOMM '98*, September 1998.

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

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

[15] Philip K. McKinley and Arun P. Mani. An experimental study of adaptive forward error correction for wireless collaborative computing. In *Proceedings of the 2001 IEEE Symposium on Applications and the Internet (SAINT-2001)*, San Diego, CA, January 2001.

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

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

[18] S. Paul, K. Sabnani, and D. Kristol. Multicast transport protocols for high speed networks. In *Proceedings of IEEE ICNP '94*, Boston, MA, October 1994.

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

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

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

[22] 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.

[23] D. Wallner, E. Harder, and Ryan Agee. Key management for multicast: Issues and architectures, RFC 2627, June 1999.

[24] Chung Kei Wong, Mohamed G. Gouda, and Simon S. Lam. Secure group communications using key graphs. In *Proceedings of ACM SIGCOMM '98*, September 1998.

[25] Chung Kei Wong and Simon S. Lam. Keystone: a group key management system. In *Proceedings of ICT 2000*, Acapulco, Mexico, May 2000.

[26] Y. Richard Yang, X. Steve Li, X. Brian Zhang, and Simon S. Lam. Reliable group rekeying: A performance analysis. In *Proceedings of ACM SIGCOMM 2001*, San Diegao, CA, August 2001.

[27] Jaehee Yoon, Azer Bestavros, and Ibrahim Matta. Adaptive reliable multicast. In *International Conference on Communications (ICC)*, New Orleans, LA, June 2000.

[28] X. Brian Zhang, Simon S. Lam, and Dong-Young Lee. Group rekeying with limited unicast recovery. Technical Report TR–02-36, The University of Texas at Austin, July 2002.

A Protocol Specification

A.1 Packet formats

Figures 5, 20, 21 and 22 define the formats of ENC, $PARITY$, USR, and $NACK$ packets, respectively. In a USR packet, the encryption IDs are optional if we arrange the encryptions in increasing order of ID.

1. Type: $PARITY$ (3 bits)	2. Reserved (1 bit)
3. Rekey message ID (12 bits)	4. Block ID (8 bits)
5. Sequence number within a block (8 bits)	6. FEC parity information for Fields 6 to 9 of ENC packets

Figure 20: Format of a $PARITY$ packet

1. Type: USR (3 bits)	2. Reserved (1 bit)
3. Rekey message ID (12 bits)	4. New user ID (16 bits)
5. A list of <encryption, ID> (variable length)	

Figure 21: Format of a USR packet

A.2 Specification of server and user protocols

The protocol for the key server is shown in Figure 23. And the protocol for a user is shown in Figure 24. In both protocols, we consider only one rekey message.

B Marking algorithm

In periodic batch rekeying, the key server collects J join and L leave requests during a rekey interval. At the end of the interval, the server runs the following marking algorithm to update the key tree and construct a rekey subtree. The marking algorithm is different from the one presented in our previous papers [26, 14]. The n-node and ID information are introduced in Section 4.

To update the key tree, the marking algorithm performs the following operations:

1. If $J = L$, replace all u-nodes who have left by the u-nodes of newly joined users.

2. If $J < L$, choose J u-nodes who have smallest IDs among the L departed u-nodes, and replace those J u-nodes with joins . Change the remaining $L - J$ u-nodes to n-nodes. If all of the children of a k-node are n-nodes, change the k-node to n-node. Repeat this operation iteratively on all k-nodes.

3. If $J > L$, first replace the u-nodes who have left by joins, then replace the n-nodes with ID between $n_k + 1$ and $d \cdot n_k + d$ (inclusive) in order of from low to high, where n_k is the maximum ID of current k-nodes. If there are still extra joins after this, keep splitting the node whose ID is equal to $n_k + 1$, and updating n_k. The split node becomes its leftmost child.

4. If any n-node has a descendant u-node, change the n-node to k-node.

1. Type: $NACK$ (3 bits)	2. Reserved (1 bit)
3. Rekey message ID (12 bits)	4. User ID (16 bits)
5. A list of <number of $PARITY$ packets requested, block ID> (variable length)	

Figure 22: Format of a $NACK$ packet

```
1. status ← MULTICAST
2. for each block do multicast $k$ $ENC$ packets and $h$ $PARITY$ packets
3. $R$ ← empty set      ▷ $R$ is the set of users who send NACKs
4. $A$ ← empty list     ▷ $A$ contains NACK information
5. for each block ID i do $amax[i] \leftarrow 0$
6. start timer
7. when receiving a NACK ($m$, a list of $< a, i >$)
8. ▷ $m$: the ID of the user who sends the NACK
9. ▷ $< a, i >$: the user requests $a$ $PARITY$ packets for block $i$
10.  do if (status = MULTICAST)
11.        then
12.             $R \leftarrow R + \{m\}$
13.             $i_m$ ← ID of the block to which the user $m$ belongs to
14.             $a_m$ ← number of $PARITY$ packets that the user requests for block $i_m$
15.             append $a_m$ to A
16.             $amax[i_m] \leftarrow \max\{amax[i_m], a_m\}$
17.        else send $USR$ packets to $m$
18.when timeout
19.  do if (it is the first round)
20.        then $UpdateRho(A)$
21.     if (conditions for switching to unicast hold)
22.        then status ← UNICAST
23.             switch to unicast
24.        else  if ($R$ is not empty)
25.                then for each block $i$
26.                       do multicast $amax[i]$ new $PARITY$ packets
27.                          $amax[i] \leftarrow 0$
28.                     start timer
```

Figure 23: Key server protocol for one rekey message

To construct the rekey subtree, the marking algorithm first copies the current key tree as the initial rekey subtree. Then the marking algorithm labels the nodes in the rekey subtree. We have four label: “Unchanged”, “Join”, “Leave”, and “Replace”:

1. First label all of the n-nodes as Leave.
2. Then label the u-nodes. Label a u-node who has departed and then joined (as another user) as Replace, a newly joined u-node as Join, and other u-nodes as Unchanged.
3. Next label the k-nodes: 1) If all the children of a key node are labeled Leave, label it as Leave, and remove all of its children from the rekey subtree. 2) Otherwise, if all of its children are Unchanged, label it as Unchanged, and remove all of its children. 3) Otherwise, if all of its children are Unchanged or Join, label it as Join. 4) Otherwise, if the node has at least one Leave or Replace child, label it as Replace.

We call the remaining subtree *rekey subtree*. Each edge in the rekey subtree corresponds to an encryption. The key server traverses the rekey subtree and uses the key assignment algorithm to assign encryptions into packets.

```
1.  for each block ID $i$ do $counter[i] \leftarrow 0$
2.  start timer
3.  when receiving a packet $pkt$
4.    do if ($pkt$ is a $USR$ packet)
5.         then $m \leftarrow$ the new ID contained in $pkt$
6.              retrieve encryptions from the packet and cancel timer
7.         else if ($pkt$ is an $ENC$ packet)
8.                then $m \leftarrow$ new ID computed
9.                     if ($pkt.frmID \leq m \leq pkt.toID$)
10.                      then retrieve required encryptions from the packet, and cancel timer
11.                      else if ($pkt$ is not a duplicate)
12.                             then $EstimateBlkID(m, high, low, pkt)$
13.                           $pkt.blkID \leftarrow$ block ID contained in $pkt$
14.                           increase $counter[pkt.blkID]$ by 1
15. when timeout
16.   do if ($high = low$) and ($counter[high] \geq k$)
17.        then decode the block, and retrieve required encryptions
18.        else for each block ID $i \in [low, high]$
19.               do if ($counter[i] \geq k$)
20.                    then decode the block
21.                         if (required $ENC$ packet is in the block)
22.                           then retrieve required encryptions, and quit
23.                    else put <$k - counter[i]$, $i$> into a $NACK$ packet
24.             send the $NACK$ packet to the key server, and start timer
```

Figure 24: User protocol for one rekey message

C Proofs of Lemma and Theorem

C.1 Proof of Lemma 1

1. Initially the key tree is empty. After collecting some join requests, the key server will construct a key tree that satisfies the property stated in this lemma at the end of the first rekey interval.

2. The property holds when the key server processes J join and L leave requests during any rekey interval because:

   (a) The property holds for $J \leq L$ because joined u-nodes replace departed u-nodes in our marking algorithm. Note that the algorithm does not change the IDs of the remaining u-nodes.

   (b) For $J > L$, newly joined u-nodes first replace departed u-nodes or the n-nodes whose IDs are larger than n_k, where n_k is the maximum ID of current k-nodes . These replacements make the property hold. Then the marking algorithm splits the node with ID $n_k + 1$. Therefore, the property holds after splitting.

C.2 Proof of Theorem 1

1. There exists an integer $x' \geq 0$ such that $n_k < f(x') \leq d \cdot n_k + d$, because:

(a) From the marking algorithm, we know that the u-node m needs to change its ID only when it splits. If no splitting happens, then $m' = m = f(0)$. Otherwise, after splitting, the u-node becomes its leftmost descendant. Then there exists an integer $x' > 0$ such that $m' = f(x')$. By Lemma 1, $n_k < m'$ since m' is a u-node.

(b) Since the maximum ID of current k-nodes is n_k, the maximum ID of current u-nodes must be less than or equal to $d \cdot n_k + d$. Therefore $m' \leq d \cdot n_k + d$.

2. Suppose besides m', there exists another leftmost descendant (denoted by m'') of m that also satisfies the condition $n_k < m'' \leq d \cdot n_k + d$. Then we get a contradiction because:

   (a) By the assumption $n_k < m''$, m'' must be a u-node or n-node. Furthermore, m'' must be a n-node and be a descendant of m' since m' is a u-node.

   (b) Since m' is the ancestor of m'', n_k is the parent node of $d \cdot n_k + d$, and by the assumption $m'' \leq d \cdot n_k + d$, we have $m' \leq n_k$. This contradicts Lemma 1 since m' is a u-node.

3. From the proof above, we have $m' = f(x')$.

D Estimating Block ID

When we partition the ENC packets into multiple blocks, and if a user loses its specific ENC packet, the user will not be able to know directly the block to which its ENC packet belongs. We address this issue in this appendix.

The key observation is that a user can estimate the block ID to which its ENC packet belongs from the ID information contained in the received ENC packets. Assume a user has ID m, and its ENC packet is the j^{th} packet in block i. Let <i, j> denote the <block ID, sequence number within a block> pair. Whenever a user receives an ENC packet, it can refine its estimation of the block ID i. For example, if m is larger than $toID$ of a received packet, then i should be larger than or equal to the block ID of the received packet because the received packet must be generated earlier than the user's specific ENC packet. In this way, if the user can receive any one ENC packet in $S_l = \{$<$i-1$, $k-1$>, <i, 0>,..., <i, $j-1$>$\}$, and receive any one ENC packet in $S_u = \{$<i, $j+1$>, ..., <i, $k-1$>, <$i+1$, 0>$\}$, then it can determine the precise value of i even if $< i, j >$ is lost. Figure 25 illustrates the block ID estimation. The detailed algorithm to estimate block ID is shown in Figure 26.

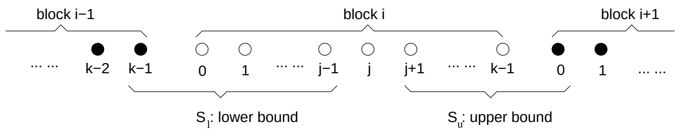


Figure 25: Illustration of block ID estimation

A user can determine the precise value of its required block ID with high probability. Only if all of the ENC packets in set $S_l + \{$<i,j>$\}$ are lost, or when all of the packets in set $S_u + \{$<i,j>$\}$ are lost, the user cannot determine the precise value of its required block ID. The probability of such failure, however, is as low as $p^{j+2} + p^{k-j+1} - p^{k+2}$, where p is the loss rate observed by the user when we assume independent losses among packets. In the worst case when $j = 0$ or $j = k-1$, the probability is about p^2. In case of failure, the user first estimates a possible range of the required block ID. Then during feedback, the user requests $PARITY$ packets for each block within the estimated block ID range. When the key server receives the NACK, it only considers the block to which the user's specific ENC packet belongs. (See the key server's protocol in Figure 23.)

```
    Algorithm $EstimateBlkID$ ($m$, $low$, $high$, $pkt$)
▷ $m$ is the ID of the user executing this procedure.
▷ $low$ is the current estimate of the lower bound of required block ID.
▷ $high$ is the current estimate of the upper bound of required block ID.
▷ $pkt$ is the $ENC$ packet received.
1.  if $(pkt.toID \leq m \leq pkt.frmID)$
2.     then $high \leftarrow pkt.blkID$
3.          $low \leftarrow pkt.blkID$
4.  if $(m > pkt.toID)$ and $(pkt.seqNo = k - 1)$
5.     then $low \leftarrow \max\{low, pkt.blkID + 1\}$
6.  if $(m > pkt.toID)$ and $(pkt.seqNo < k - 1)$
7.     then $low \leftarrow \max\{low, pkt.blkID\}$
8.  if $(m < pkt.frmID)$ and $(pkt.seqNo = 0)$
9.     then $high \leftarrow \min\{high, pkt.blkID - 1\}$
10. if $(m < pkt.frmID)$ and $(pkt.seqNo > 0)$
11.    then $high \leftarrow \min\{high, pkt.blkID\}$
12. if $(m > pkt.toID)$
13.    then $high \leftarrow \min\{high, pkt.blkID + \lceil \frac{d \cdot (pkt.maxKID+1) - pkt.toID - (k-1-pkt.seqNo)}{k} \rceil\}$
```

Figure 26: Estimating required block ID

$EstimateBlkID$ algorithm works as follows. Initially, a user sets the lower bound low as 0, and upper bound $high$ as infinity. The **if** statement of lines 12-13 in Figure 26 guarantees that eventually $high$ will not be infinity if the user receives any ENC packet. The reasoning is as follows. When the user receives an ENC packet pkt, the $maxKID$ field of the packet specifies the maximum ID of current k-nodes. Therefore, the maximum ID of current users cannot be larger than $d \cdot (pkt.maxKID + 1)$. In the worst case, one ENC packet contains encryptions for only one user, then there are at most $(d \cdot (pkt.maxKID + 1) - pkt.toID)$ ENC packets whose $frmID$ sub-field is larger than $pkt.toID$. Therefore, the maximum block ID cannot be larger than $pkt.blkID + \lceil \frac{d \cdot (pkt.maxKID+1) - pkt.toID - (k-1-pkt.seqNo)}{k} \rceil$.