# Protocol Design for Scalable and Reliable Group Rekeying

X. Brian Zhang, *Student Member, IEEE*, Simon S. Lam, *Fellow, IEEE*, Dong-Young Lee, and Y. Richard Yang, *Member, IEEE*

***Abstract*—We present the design and specification of a protocol for scalable and reliable 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 forward error correction (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.**

***Index Terms*—Adaptive FEC, group key management, proactive FEC, reliable multicast, secure multicast.**

## I. Introduction

MANY EMERGING Internet applications, such as pay-per-view distribution of digital media, restricted teleconferences, multiparty games, and virtual private networks will benefit from using a secure group communications model [9]. 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 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 [24]. 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 [24].

When a user wants to join a group, the user and registration component mutually authenticate each other using a protocol such as SSL. If authenticated and accepted into the group, the new user receives 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 [1], [4], [7], [22], [23], [25]. In this paper, we follow the *key tree* approach [22], [23], 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 the group size and $d$ is the key tree degree. Rekeying after every join or leave request, however, can still incur a large server processing overhead. Thus, we propose to further reduce processing overhead by using periodic rekeying [13], [20], [25], 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 [24], 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 [25] and observed that although many reliable multicast protocols have been proposed and studied in recent years

Manuscript received September 11, 2001; revised August 16, 2002; approved by IEEE/ACM TRANSACTIONS ON NETWORKING Editor S. Paul. This work was supported in part by NSF Grant ANI-9977267 and NSA INFOSEC University Research Program Grant MDA904-98-C-A901. An early version of this paper appeared in *Proceedings of the SPIE Conference on Scalability and Traffic Control in IP Networks*, Denver, CO, August 2001.

X. B. Zhang, S. S. Lam, and D.-Y. Lee are with the Department of Computer Sciences, University of Texas, Austin, TX 78712-0233 USA (e-mail: zxc@cs.utexas.edu; lam@cs.utexas.edu; dylee@cs.utexas.edu).

Y. R. Yang is with the Department of Computer Science, Yale University, New Haven, CT 06511 USA (e-mail: yry@cs.yale.edu).

Digital Object Identifier 10.1109/TNET.2003.820256

[5], [8], [10], [12], [15]–[17], [21], 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 should 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. 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. Finally, an adaptive algorithm to adjust the proactivity factor (see Table I for its definition) 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 [27] and [28].)

Our server protocol for each rekey message consists of four phases:

1) generating a sequence of packets containing encrypted keys (called ENC packets);
2) generating packets containing FEC redundant information (called PARITY packets);
3) multicast of ENC and PARITY packets;
4) 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 [11], [19].

[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.

TABLE I
NOTATION

| *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 |
| $N$ | number of existing users |
| $h$ | number of proactive $PARITY$ packets for each block |
| $k$ | block size; also denotes a key when it appears in $\{k'\}_k$ |
| $\rho$ | proactivity factor, defined as $(h+k)/k$ |
| $u^*$ | target number of NACKs for the first multicast round |
| $\alpha$ | percentage of high loss rate receivers |

Furthermore, 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.

Toward a scalable design, 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 each user has a high probability to receive its specific packet. As a result, 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 II, we briefly review the ideas of key tree and periodic batch rekeying. In Section III, we present our server and user protocols. In Section IV, 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 V. In Section VI, we discuss how to adaptively adjust the proactivity factor to achieve low delivery latency with a small bandwidth overhead. In Section VII, we discuss when and how to unicast. Our conclusions are given in Section VIII.

## II. Background

We review in this section the ideas of key tree [22], [23] and periodic batch rekeying [13], [20], [25] and present a new marking algorithm. The algorithm is used to update the key tree and generate workload for rekey transport.

### A. 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

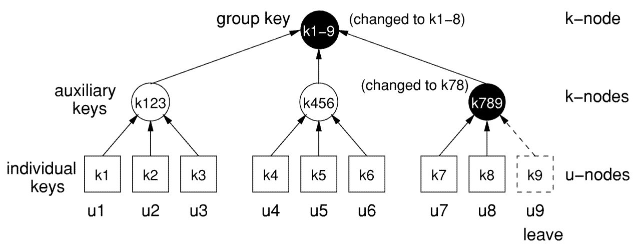


Fig. 1. Example key tree.

```
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 do
5.     adaptively adjust proactivity factor;
6.     if conditions for switching to unicast hold then
7.         unicast $USR$ packets to users who did not receive their required encryptions;
8.     else
9.         collect $amax[i]$ as the largest number of $PARITY$ packets needed for
           each block $i$;
10.        generate $amax[i]$ new $PARITY$ packets for each block $i$;
11.        multicast these $PARITY$ packets to all users at the beginning of next round;
```

Fig. 2. Basic protocol for key server.

from its u-node to the root. Consider a group with nine users. An example key tree is shown in Fig. 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 $u_9$.

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 [23], 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.

## B. 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 those in our previous papers [13], [25]. This new algorithm facilitates key identification (see Section IV-A).

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 (see Fig. 28). 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.

## III. 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 Fig. 2. Notation used in this paper is defined in Table I.

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.

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 the *proactivity factor*, denoted by $\rho$.

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 RTT 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 VII). 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.

[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.

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

Fig. 3. Basic protocol for a user.

An informal specification of the user protocol is shown in Fig. 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 packets received in the block containing its specific ENC packet.

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

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

## IV. 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; this 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. Finally, we present and evaluate our key assignment algorithm.

### A. 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. Fig. 4

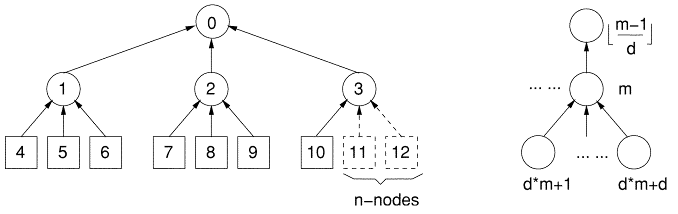


Fig. 4. Illustration of key identification.

(left) illustrates the IDs of nodes in an expanded key tree with a tree degree of three.

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 $\lfloor (m-1)/(d) \rfloor$, where $d$ is the key tree degree. Fig. 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 corresponding u-node, which contains 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 $f(x) = d^x m + (1 - d^x)/(1-d)$ 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.

### B. Format of ENC Packets

Given the results in the previous subsection, we can now define the format of an ENC packet. As shown in the Fig. 5, an ENC packet has nine fields, and contains both ID information and encryptions. Each number in parentheses of Fig. 5 is the suggested field length, in number of bits.

The ID information in an ENC packet allows a user to identify the packet, extract its required encryptions, and update its

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

Fig. 5. Format of an ENC packet.

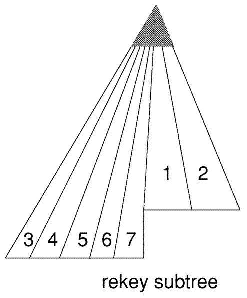


Fig. 6. Illustration of UKA algorithm.

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 V. Field 6 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 new IDs are in the range of $\langle \mathrm{frmID}, \mathrm{toID} \rangle$ 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.

### C. 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 algorithm (UKA). UKA guarantees that all of the encryptions for a user are assigned into a single ENC packet.

Fig. 6 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 $\langle \mathrm{frmID}, \mathrm{toID} \rangle$ intervals do not overlap. In particular, the algorithm guarantees that $\mathrm{toID}$ of a previous ENC packet is less than the $\mathrm{frmID}$ of the next packet. This property is useful for block ID estimation to be performed by a user (see Section V-A).

### D. 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.

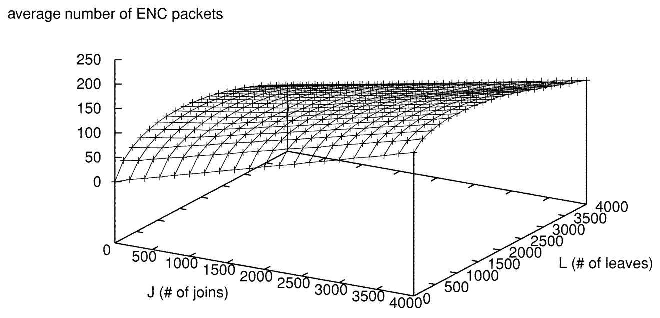


Fig. 7. Average number of ENC packets as a function of $J$ and $L$ for $N = 4096$.

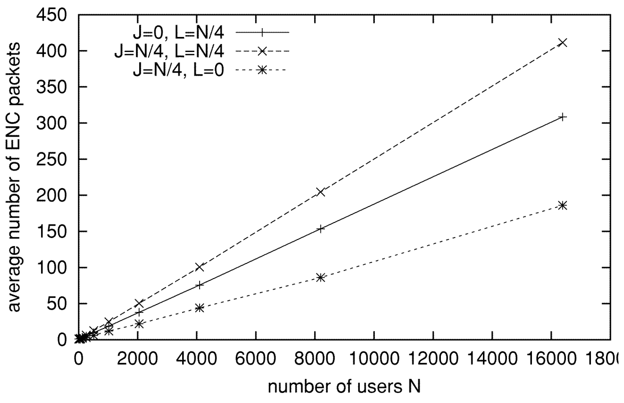


Fig. 8. Average number of ENC packets as a function of $N$.

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 Fig. 7. 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 Fig. 8. We observe that the average number of ENC packets in a rekey message increases linearly with $N$ for three combinations of $J$ and $L$ values.

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. Fig. 9 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

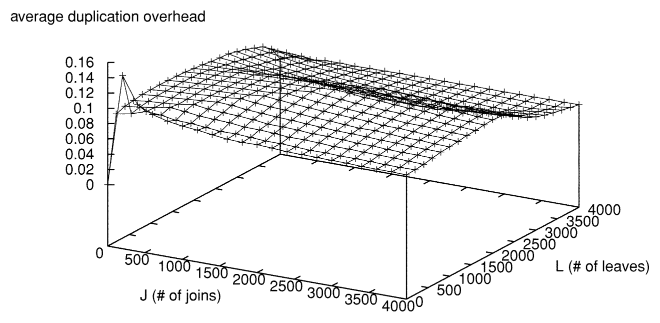


Fig. 9. Average duplication overhead as a function of $J$ and $L$ for $N = 4096$.

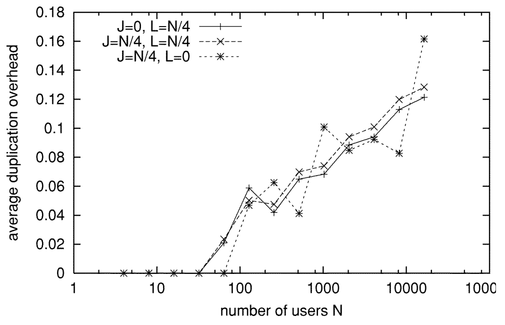


Fig. 10. Average duplication overhead as a function of $N$.

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 Fig. 10 the average duplication overhead as a function of $N$. We observe that for $J = 0$ and $L = N/4$, 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 $J = 0$ and $L = N/4$, 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 $(\log_d(N) - 1)/(46)$, where 46 is the number of encryptions that can be carried in an ENC packet with a packet size of 1028 bytes. For $J = N/4$ and $L = 0$, the rekey subtree is very sparse, and thus the graph of duplication overhead fluctuates around the graph of $J = L = N/4$.

## V. Block Partitioning

After running the UKA assignment algorithm to generate the ENC packets of a rekey message, the key server next generates PARITY packets for the ENC packets using an 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 [3], [15], [18]. For example, using the RSE coder of Rizzo [18], 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 until there are $k$ packets to fill the last block. (The key server may choose ENC packets from other blocks to duplicate, but all duplicates are used to fill the last block.) We use a flag bit in each ENC packet to specify whether the packet is a duplicate, as shown in Fig. 5. A duplicate ENC packet has the same contents in all fields as the original packet except for the $\langle \text{block ID}, \text{sequence number} \rangle$ and flag bit fields. A new $\langle \text{block ID}, \text{sequence number} \rangle$ pair is assigned to each duplicate ENC packet because Reed–Solomon encoding needs to uniquely identify every packet, duplicate or not.

### A. 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 needs to determine the block to which its ENC packet belongs. Then the user will try to recover this block through FEC decoding. We present an algorithm in Appendix D for users to estimate the ID of the block containing its specific ENC packet. 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.

### B. 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.

### C. 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 in terms of 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.[3]

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 in [16]. 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 $\alpha$ 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 [15] to simulate burst loss as follows. The average duration of a burst loss is $100/p$ ms, and the average duration of loss-free time between consecutive loss bursts is $100/(1-p)$ ms.[4] 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/s, and the rekey interval is 60 s, 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.

The impact of block size on the key server's bandwidth overhead is shown in Fig. 11. Here we set $\rho = 1$ (that is, $h = 0$).[5] 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 [18], 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. Also the encoding time for one PARITY packet is approximately a linear function of $k$. The relative overall encoding time (assuming $k$ time units to generate one PARITY packet for block size $k$) is shown in Fig. 12.

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.

## VI. Adaptive 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 $\rho$. In this section, we investigate how to determine $\rho$.

Proactive FEC has been widely used to improve reliability and reduce delivery latency [2], [6], [11], [14], [15], [19], [26]. 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.

### A. Impact of Proactivity Factor

Before designing an algorithm to adjust $\rho$, it is desirable to evaluate the impact of $\rho$ on the number of NACKs, the delivery latency at users, and the bandwidth overhead at the key server.

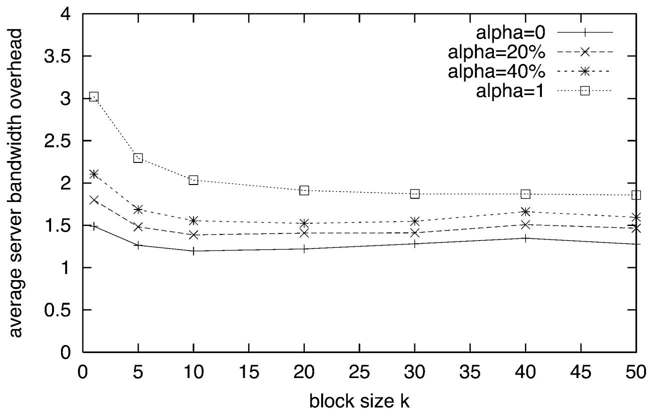


Fig. 11. Average server bandwidth overhead as a function of block size.

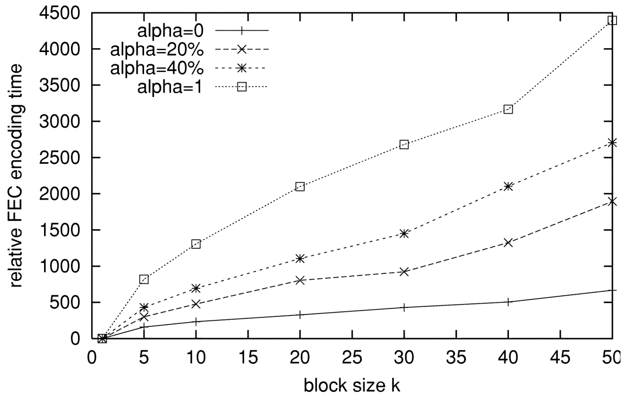


Fig. 12. Relative overall FEC encoding time as a function of block size.

[3]Note that no unicast recovery is involved while we evaluate the server bandwidth overhead. The key server will do multicast until all users receive or can recover their specific ENC packets.

[4]This network topology and loss model are simplistic compared to the Internet. They are however needed for simulating a large group size (up to 16 384). 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 [27], [28].

[5]The result for the case of adaptive $\rho$ is similar. See [29] and [30] for details.

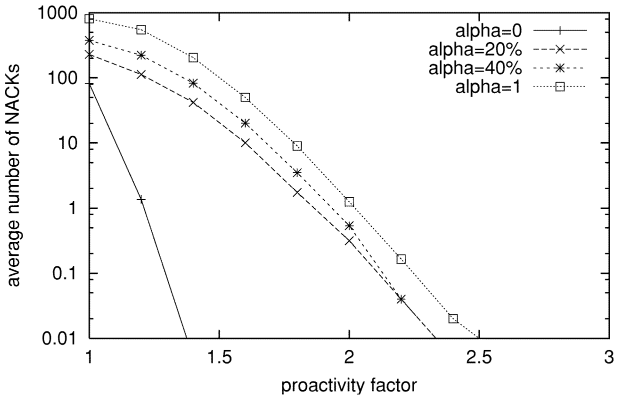


Fig. 13. Average number of NACKs in the first round as a function of $\rho$.

TABLE II
PERCENTAGE OF USERS ON AVERAGE WHO NEED A GIVEN NUMBER OF ROUNDS TO RECEIVE THEIR ENCRYPTIONS

| $\rho$ | *Percentage of users who need* | | | |
|---|---|---|---|---|
| | 1 *round* | 2 *rounds* | 3 *rounds* | $\geq 4$ *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% |

We first evaluate the impact of $\rho$ on the number of NACKs. Fig. 13 plots the average number of NACKs for the first round as a function of $\rho$. Note that the $y$ axis is in log scale. We observe that the average number of NACKs decreases exponentially as we increase $\rho$. (A similar observation was made in [19].)

We next evaluate the impact of $\rho$ on delivery latency. Table II 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 then evaluate the impact of $\rho$ on the average server bandwidth overhead, as shown in Fig. 14. For $\rho$ 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 $\rho$ has little impact on the average server bandwidth overhead. When $\rho$ becomes large, the bandwidth overhead during the first round dominates the overall bandwidth overhead, and the overall bandwidth overhead increases linearly with $\rho$.

In summary, we observe that an increase of $\rho$ can have the following three effects.

1) It will significantly reduce the average number of NACKs for the first multicast round.
2) It will reduce the worst-case delivery latency.
3) It will increase the key server's bandwidth overhead when $\rho$ is larger than needed.

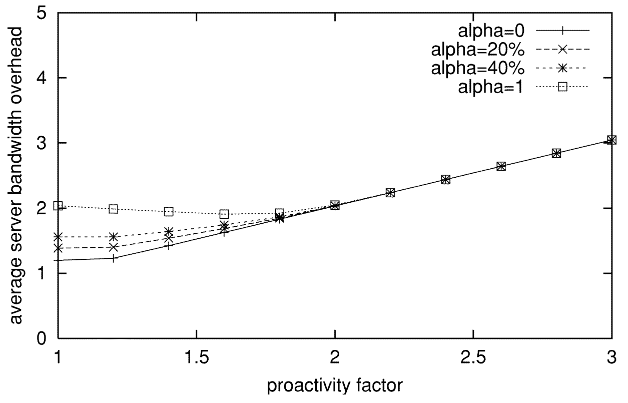


Fig. 14. Average server bandwidth overhead as a function of $\rho$.

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

Fig. 15. Algorithm to adaptively adjust proactivity factor.

## B. Adjustment of Proactivity Factor

We present in Fig. 15 an algorithm to adaptively adjust $\rho$. The basic idea of the algorithm is to adjust $\rho$ based on NACK information received for the current rekey message, such that a target number of NACKs are expected to be returned for the next rekey message. The key server runs this algorithm at the end of the first multicast round.

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

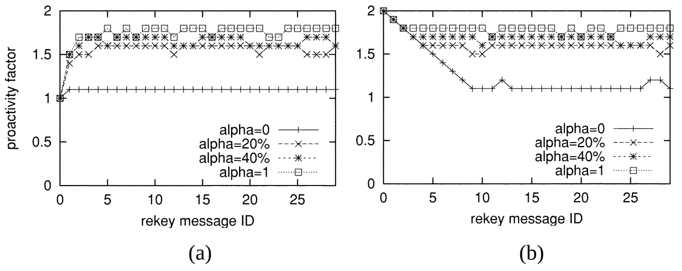


Fig. 16. Traces of proactivity factor with (a) initial $\rho = 1$ and (b) initial $\rho = 2$.

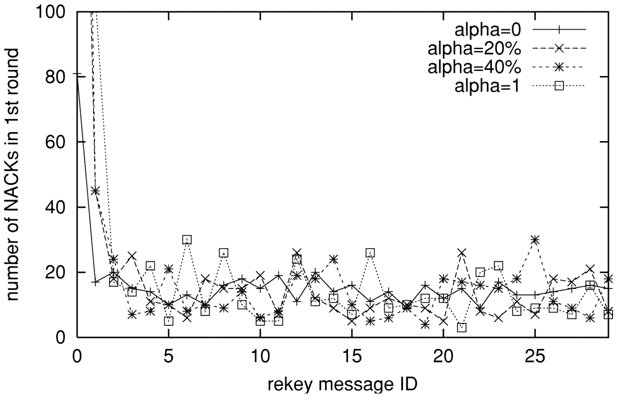


Fig. 17. Traces of the number of NACKs with initial $\rho = 1$.

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 $\rho$ by one PARITY packet with probability equal to $(u^* - 2 \cdot \text{size}(A))/(u^*)$.

## C. Performance Evaluation

We use simulations to evaluate algorithm *AdjustRho*. We will first investigate whether our protocol can effectively control the number of NACKs, and then evaluate the extra bandwidth overhead that it may incur. In the experiments to be presented, we choose 20 to be a default value of $u^*$ unless otherwise specified.

*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.

Fig. 16 shows how $\rho$ is adaptively adjusted when the key server sends a sequence of rekey messages. For initial $\rho = 1$ as shown in Fig. 16(a), we observe that it takes only two or three rekey messages for $\rho$ to settle down to stable values. For initial $\rho = 2$ as shown in Fig. 16(b), we observe that $\rho$ 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.

Figs. 17 and 18 plot the traces of the number of NACKs for the first multicast round. In Fig. 17, where the initial $\rho$ value is 1, the number of NACKs received stabilizes very quickly, and the stable values are generally less than 1.5 times of $u^*$. Fig. 18 shows the case for initial $\rho = 2$. We observe that the stable values of these two figures match very well.

We then evaluate whether algorithm *AdjustRho* can control the number of NACKs for various values of $u^*$. As shown in

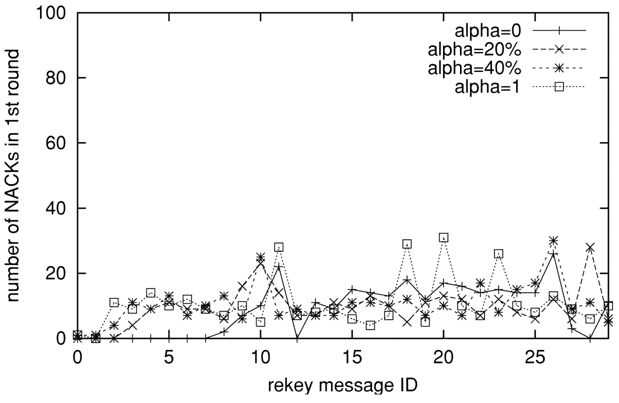


Fig. 18. Traces of the number of NACKs with initial $\rho = 2$.

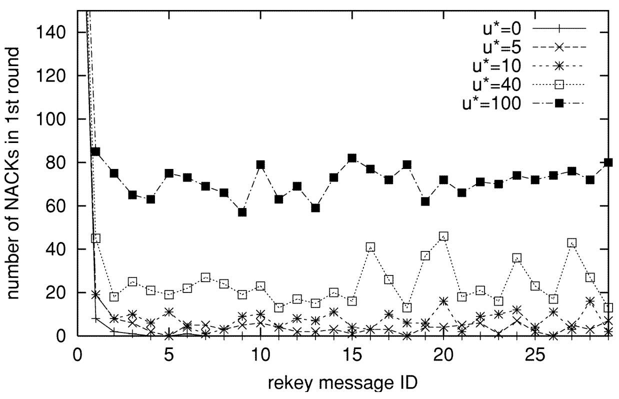


Fig. 19. Traces of the number of NACKs for various $u^*$ values with initial $\rho = 1$.

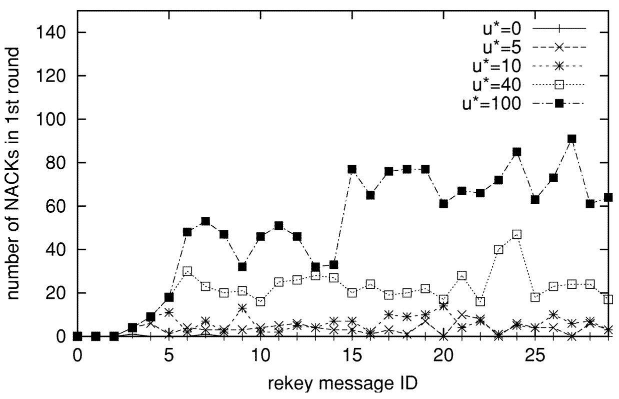


Fig. 20. Traces of the number of NACKs for various $u^*$ values with initial $\rho = 2$.

Figs. 19 (initial $\rho$ is 1) and 20 (initial $\rho$ is 2), the number of NACKs received at the key server fluctuates around each $u^*$ target number specified. However, we do observe that the fluctuations become more significant for larger values of $u^*$. Therefore, in choosing $u^*$, we need to consider the potential impact of large fluctuations when $u^*$ is large.

*2) Overhead of Adaptive FEC:* From the previous section, we know that algorithm *AdjustRho* can effectively control the number of NACKs and thus reduce delivery latency. However, compared with an approach that does not send any proactive PARITY packets at all during the first round and only generates reactive PARITY packets during the subsequent rounds, the

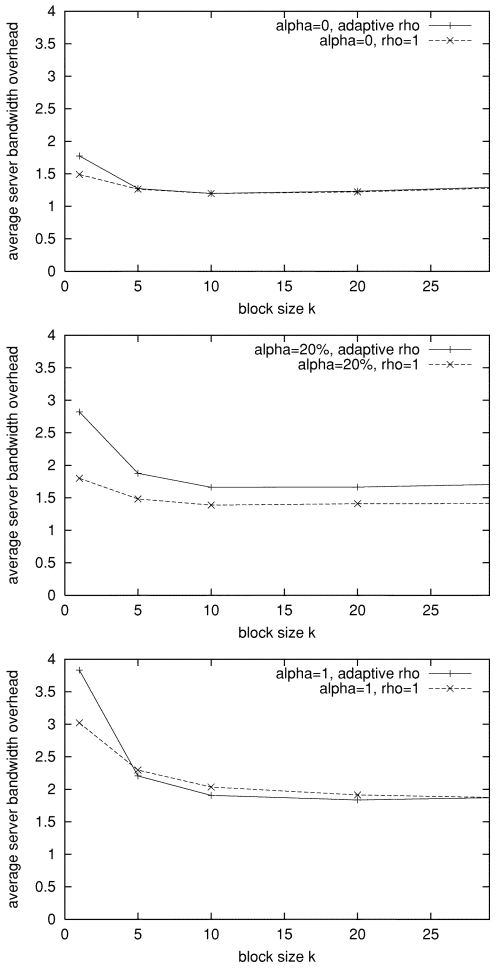


Fig. 21. Average server bandwidth overhead for adaptive FEC and for $\rho = 1$ case in various loss conditions.

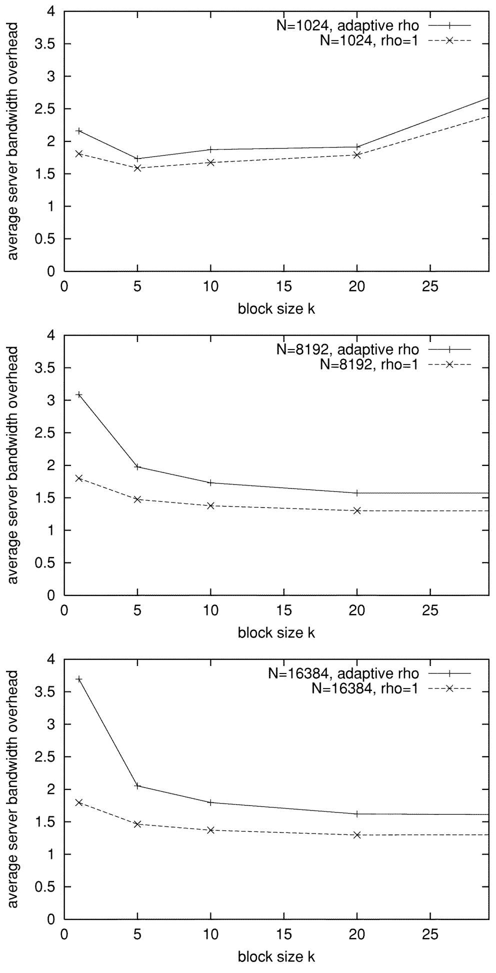


Fig. 22. Average server bandwidth overhead for adaptive FEC and for $\rho = 1$ case when group size $N$ varies.

adaptive FEC scheme may incur extra bandwidth overhead at the key server. We investigate this issue in this subsection.

We first evaluate the extra server bandwidth overhead caused by adaptive FEC in various loss conditions. Fig. 21 compares the average server bandwidth overhead for adaptive FEC[6] with the case that all PARITY packets are generated reactively (we call it $\rho = 1$ case). We observe that our adaptive scheme causes little extra server bandwidth overhead in a homogeneous low loss environment (i.e., $\alpha = 0$). For $\alpha = 1$, our scheme can even save a little bandwidth. This is because for $\rho = 1$ case, the key server takes more rounds for all users to recover their encryptions in the reactive scheme than in the adaptive scheme. Therefore, it is possible that the total number of PARITY packets generated during the rounds for $\rho = 1$ case is larger than that of the adaptive scheme. In the case of $\alpha = 20\%$, the extra bandwidth overhead generated by adaptive $\rho$ is less than 0.3 for $k \geq 10$.

We next evaluate the average server bandwidth overheads of the two schemes for various group sizes. From Fig. 22, we observe that the extra bandwidth overhead incurred by adaptive $\rho$ increases with $N$, but the extra bandwidth overhead incurred is still less than 0.42 even for $N = 16\,384$ when $k \geq 10$.

[6]To measure the average server bandwidth overhead for adaptive FEC, we set initial $\rho = 1$ and let the key server send out ten rekey messages. We then compute the average server bandwidth overhead based on the next 100 or more rekey messages.

## VII. Speedup With Unicast

Rekey transport has a soft real-time requirement, that is, it is desirable for all users to receive their new keys before the start of the next rekey interval. To meet this requirement, we have proposed in the previous section to adaptively adjust $\rho$ during the multicast phase to reduce the number of users who send NACKs. To further reduce delivery latency, 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 RTT over all users.

One issue of early unicast is its possible high bandwidth overhead at the key server. In our protocol, however, 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 $(4 + 22 \cdot l)$ bytes (see Appendix A for the format of a USR packet), 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 $u^*$ is small enough. In fact, our evaluations show that for $u^* = 20, N = 4096$, and initial $\rho = 1$, roughly five or fewer 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.

During the unicast step, to further improve reliability and reduce delivery latency, the key server sends multiple copies of a USR packet to a user who needs recovery. The number of copies to send depends on the loss rate of the user, which can be easily estimated by the key server (see [29] and [30] for details).

## VIII. Conclusion

The objective of this paper has been to present in detail our rekey transport protocol as well as its performance. Our server protocol for each rekey message consists of four phases: 1) generating a sequence of ENC packets containing encrypted keys; 2) generating PARITY packets; 3) multicast of ENC and PARITY packets; and 4) 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 after the key tree has been modified. 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. To reduce delivery latency, the key server estimates the user's loss rate, and sends multiple copies of the user's USR packet during unicast.

```
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.  ▷ $amax[i]$ record the largest number of $PARITY$ packets requested
           by users for block $i$ until now.
7.  start timer;
8.  when receiving a NACK ($m$, a list of $< a, i >$) do
9.      ▷ $m$: the ID of the user who sends the NACK.
10.     ▷ $< a, i >$: the user requests $a$ $PARITY$ packets for block $i$.
11.     if (status = MULTICAST) then
12.         $R \leftarrow R + \{m\}$;
13.         $i_m$ ← ID of the block to which the user $m$ belongs;
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 do
19.     if it is the first round then execute $UpdateRho(A)$;
20.     if conditions for switching to unicast hold then
21.         status ← UNICAST;
22.         unicast $USR$ packets to each user in $R$
23.     else
24.         if $R$ is not empty then
25.             for each block $i$ do
26.                 multicast $amax[i]$ new $PARITY$ packets;
27.                 $amax[i] \leftarrow 0$;
28.         start timer;
```

Fig. 23. Key server protocol for one rekey message.

In summary, we have the following contributions. First, a new marking algorithm for batch rekeying is presented. This algorithm facilitates key identification. 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. Finally, 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 [27] and [28].

## Appendix A
## Protocol Specification

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

Figs. 5, 25, 26, and 27 define the formats of ENC, PARITY, USR, and NACK packets, respectively. Each number in parentheses is the suggested field length, in number of bits. In a USR packet, the encryption IDs are optional if we arrange the encryptions in increasing order of ID.

## Appendix 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

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

Fig. 24. User protocol for one rekey message.

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

Fig. 25. Format of a PARITY packet.

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

Fig. 26. Format of a USR packet.

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

Fig. 27. Format of a NACK packet.

algorithm is different from those in our previous papers [13], [25].

The marking algorithm consists of two steps. In the first step, the algorithm modifies the key tree to satisfy the leave and join requests. The operations for this step are specified in Fig. 28. The n-node and ID information used in the algorithm are presented in Section IV.

In the second step, the marking algorithm constructs a rekey subtree. The operations are specified in Fig. 29. The input to the second step of the algorithm is a copy of the updated key tree. The algorithm will label all the nodes and then prune the tree. 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.  if $(J = L)$ then
2.      replace all u-nodes that have left by the u-nodes of newly joined users;
3.  else if $(J < L)$ then
4.      choose $J$ u-nodes that have smallest IDs among the $L$ departed u-nodes,
        and replace those $J$ u-nodes with joins;
5.      change the remaining $L - J$ u-nodes to n-nodes;
6.      for each k-node in order of ID from high to low do
7.          if all the children of this k-node are n-nodes then
8.              change the k-node to n-node;
9.  else    ▷ the case for $J > L$.
10.     replace the u-nodes that have left by joins;
11.     replace those n-nodes by joins whose IDs are between $n_k + 1$ and
        $d \cdot n_k + d$ (inclusive) in order of from low to high
12.     ▷ $n_k$ is the maximum k-node ID.
13.     while there are still extra joins do
14.         add children to the u-node whose ID is $n_k + 1$, move the u-node to
            become the leftmost child (with node $n_k + 1$ becoming a k-node)
            and up to $d - 1$ new joins are added as children;
15.         update $n_k$;
16. for each n-node do
17.     if the n-node has a descendant u-node then change the n-node to k-node;
```

Fig. 28. Marking algorithm step 1: Updating the key tree.

```
▷ input: a copy of updated key tree.
▷ output: rekey subtree.
▷ output: rekey subtree.
1.  for each n-node do
2.      if the n-node is created in Step 1 as a result of a u-node's departure then
3.          label the n-node as LEAVE;
4.      else remove the n-node;
5.  for each u-node do
6.      if the u-node has departed and then joined (as another user) then
7.          label it as REPLACE;
8.      else if it is a newly joined u-node then
9.          label it as JOIN;
10.     else label it as UNCHANGED;
11. for each k-node in order of ID from high to low do
12.     if all the children of the k-node are labeled as LEAVE then
13.         label the k-node as LEAVE, and remove all of its children;
14.     else if all of its children are UNCHANGED then
15.         label the k-node as UNCHANGED, and remove all of its children;
16.     else if all of its children are UNCHANGED or JOIN then
17.         label the k-node as JOIN;
18.     else label the k-node as REPLACE;
```

Fig. 29. Marking algorithm step 2: Constructing a rekey subtree.

## APPENDIX C
## PROOFS OF LEMMA 1 AND THEOREM 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. □

*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 $m' = f(x')$.

□

## APPENDIX 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.

A user can estimate the block ID to which its ENC packet belongs from the ID information contained in the received ENC packets. Suppose a user's ENC packet is the $j$th packet in block $i$. Let $\langle i, j \rangle$ denote the $\langle \text{block ID}, \text{sequence number within a block} \rangle$ pair. Whenever a user receives an ENC packet, it first computes its new user ID, denoted by $m$, then it refines its estimation of the block ID $i$. For example, if the received packet is not a duplicate and $m$ is larger than *toID* of this 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 $S_l = \{\langle i-1, k-1 \rangle, \langle i, 0 \rangle, \ldots, \langle i, j-1 \rangle\}$, and receive any one ENC packet in $S_u = \{\langle i, j+1 \rangle, \ldots, \langle i, k-1 \rangle, \langle i+1, 0 \rangle\}$, then it can determine the precise value of $i$ even if $\langle i, j \rangle$ is lost. Fig. 30 illustrates the block ID estimation. The detailed algorithm to estimate block ID is shown in Fig. 31.

A user can determine the precise value of its required block ID with high probability. The probability of such failure is as low as $p^{j+2} + p^{k-j+1} - p^{k+2}$, as shown in Lemma 2. In the worst case when $j = 0$ or $j = k-1$, the probability is about $p^2$.

*Lemma 2:* Assume packets experience independent loss. Let $p$ be the packet loss rate observed by a user. Then the probability that the user cannot determine the precise value of the ID of the block to which its specific ENC packet belongs is $p^{j+2} + p^{k-j+1} - p^{k+2} = O(p^2)$, where $j, 0 \leq j \leq k-1$ is the sequence number of its specific ENC packet and $k, k \geq 1$ is the block size.

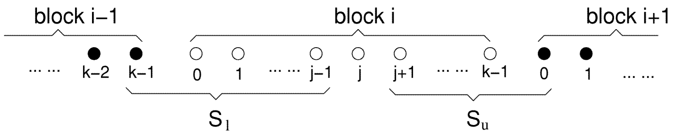


Fig. 30. Illustration of block ID estimation.

```
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** $pkt$ is a duplicate **then**      ▷ duplicate packets are only in the last block.
2.      $high \leftarrow pkt.blkID$;
3.  **else if** $(pkt.toID \leq m \leq pkt.frmID)$ **then**
4.      $high \leftarrow pkt.blkID$; $low \leftarrow pkt.blkID$;
5.  **else if** $(m > pkt.toID)$ **and** $(pkt.seqNo = k - 1)$ **then**
6.      $low \leftarrow \max\{low, pkt.blkID + 1\}$;
7.  **else if** $(m > pkt.toID)$ **and** $(pkt.seqNo < k - 1)$ **then**
8.      $low \leftarrow \max\{low, pkt.blkID\}$;
9.  **else if** $(m < pkt.frmID)$ **and** $(pkt.seqNo = 0)$ **then**
10.     $high \leftarrow \min\{high, pkt.blkID - 1\}$;
11. **else if** $(m < pkt.frmID)$ **and** $(pkt.seqNo > 0)$ **then**
12.     $high \leftarrow \min\{high, pkt.blkID\}$;
13. **else if** $(m > pkt.toID)$ **then**
14.     $high \leftarrow \min\{high, pkt.blkID +$
        $\lceil \frac{d \cdot (pkt.maxKID+1) - pkt.toID - (k-1-pkt.seqNo)}{k} \rceil\}$;
```

Fig. 31. Estimating required block ID.

*Proof:* As illustrated in Fig. 30, only if all of the ENC packets in set $S_l + \{\langle \text{i}, j \rangle\}$ are lost, or when all of the packets in set $S_u + \{\langle \text{i}, j \rangle\}$ are lost, the user cannot determine the precise value of its required block ID. The probability of such failure is $p^{j+2} + p^{k-j+1} - p^{k+2}$, which is $O(p^2)$ since $0 \leq j \leq k-1$ and $k \geq 1$. □

In the case that the user cannot determine the precise value of its block ID, it can still estimate 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 Fig. 23).

In the *EstimateBlkID* algorithm shown in Fig. 31, a user sets the initial values of the lower bound *low* and upper bound *high* as 0 and $\infty$, respectively. However, the **if** statement of lines 11–12 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 $d \cdot (\text{pkt.maxKID} + 1)$. In the worst case, one ENC packet contains encryptions for only one user, then there are at most $(d \cdot (\text{pkt.maxKID} + 1) - \text{pkt.toID})$ ENC packets each with $\text{frmID}$ subfield larger than $\text{pkt.toID}$. Therefore, the maximum block ID cannot be larger than

$$\text{pkt.blkID} + \left\lceil \frac{d \cdot (\text{pkt.maxKID}+1) - \text{pkt.toID} - (k-1-\text{pkt.seqNo})}{k} \right\rceil .$$

## ACKNOWLEDGMENT

The authors would like to thank M. S. Kim, J. Liu, and the anonymous reviewers for their constructive comments.

## REFERENCES

[1] D. Balenson, D. McGrew, and A. Sherman. (1999) Key management for large dynamic groups: One-way function trees and amortized initialization. Internet draft. [Online]. Available: http://www.securemulticast.org/smug-drafts.htm

[2] J.-C. Bolot, S. Fosse-Parisis, and D. Towsley, "Adaptive FEC-based error control for internet telephony," in *Proc. IEEE INFOCOM*, Mar. 1999, pp. 1453–1460.

[3] J. W. Byers, M. Luby, M. Mitzenmacher, and A. Rege, "A digital fountain approach to reliable distribution of bulk data," in *Proc. ACM SIGCOMM*, Vancouver, B.C., Canada, Sept. 1998, pp. 56–67.

[4] I. Chang, R. Engel, D. Kandlur, D. Pendarakis, and D. Saha, "Key management for secure internet multicast using Boolean function minimization techniques," in *Proc. IEEE INFOCOM*, vol. 2, Mar. 1999, pp. 689–698.

[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 Trans. Networking*, vol. 5, pp. 784–803, Dec. 1997.

[6] M. Handley, S. Floyd, B. Whetten, R. Kermode, L. Vicisano, and M. Luby, "The reliable multicast design space for bulk data transfer," Network Working Group, RFC 2887, Aug. 2001.

[7] H. Harney and E. Harder. (1999) Logical key hierarchy protocol. Internet draft. [Online]. Available: http://securemulticast.org/smug-related.htm

[8] Reliable multicast transport (rmt) charter, Internet Research Task Force (IRTF). [Online]. Available: http://www.ietf.org/html.charters/rmt-charter.html

[9] The Secure Multicast Research Group (SMuG), Internet Research Task Force (IRTF). [Online]. Available: http://www.securemulticast.org/

[10] S. K. Kasera, J. Kurose, and D. Towsley, "A comparison of server-based and receiver-based local recovery approaches for scalable reliable multicast," in *Proc. IEEE INFOCOM*, San Francisco, CA, Mar. 1998, pp. 988–995.

[11] R. G. Kermode, "Scoped hybrid automatic repeat request with forward error correction (SHARQFEC)," in *Proc. ACM SIGCOMM*, Sept. 1998, pp. 278–289.

[12] B. Levine and J. J. Garcia-Luna-Aceves, "A comparison of known classes of reliable multicast protocols," in *Proc. IEEE Int. Conf. Network Protocols*, Columbus, OH, Oct. 1996, pp. 112–121.

[13] X. S. Li, Y. R. Yang, M. G. Gouda, and S. S. Lam, "Batch rekeying for secure group communications," in *Proc. 10th Int. World Wide Web Conf.*, Hong Kong, May 2001, pp. 525–534.

[14] P. K. McKinley and A. P. Mani, "An experimental study of adaptive forward error correction for wireless collaborative computing," in *Proc. IEEE Symp. Applications and the Internet*, San Diego, CA, Jan. 2001, pp. 157–166.

[15] J. Nonnenmacher, E. Biersack, and D. Towsley, "Parity-based loss recovery for reliable multicast transmission," in *Proc. ACM SIGCOMM*, Sept. 1997, pp. 289–300.

[16] J. Nonnenmacher, M. Lacher, M. Jung, E. W. Biersack, and G. Carle, "How bad is reliable multicast without local recovery?," in *Proc. IEEE INFOCOM*, San Francisco, CA, Mar. 1998, pp. 972–979.

[17] S. Paul, K. Sabnani, and D. Kristol, "Multicast transport protocols for high speed networks," in *Proc. IEEE Int. Conf. Network Protocols*, Boston, MA, Oct. 1994, pp. 4–14.

[18] L. Rizzo, "Effective erasure codes for reliable computer communication protocols," *Comput. Commun. Rev.*, vol. 27, no. 2, pp. 24–36, Apr. 1997.

[19] D. Rubenstein, J. Kurose, and D. Towsley, "Real-time reliable multicast using proactive forward error correction," in *Proc. NOSSDAV*, Cambridge, U.K., July 1998, pp. 279–194.

[20] S. Setia, S. Koussih, S. Jajodia, and E. Harder, "Kronos: A scalable group re-keying approach for secure multicast," *Proc. IEEE Symp. Security and Privacy*, pp. 215–228, May 2000.

[21] D. Towsley, J. Kurose, and S. Pingali, "A comparison of sender-initiated reliable multicast and receiver-initiated reliable multicast protocols," *IEEE J. Select. Areas Commun.*, vol. 15, pp. 398–406, Mar. 1997.

[22] D. Wallner, E. Harder, and R. Agee, "Key management for multicast: Issues and architectures," Network Working Group, RFC 2627, June 1999.

[23] C. K. Wong, M. G. Gouda, and S. S. Lam, "Secure group communications using key graphs," in *Proc. ACM SIGCOMM*, Sept. 1998, pp. 68–79.

[24] C. K. Wong and S. S. Lam, "Keystone: A group key management system," in *Proc. Int. Conf. Telecommunications*, Acapulco, Mexico, May 2000.

[25] Y. R. Yang, X. S. Li, X. B. Zhang, and S. S. Lam, "Reliable group rekeying: A performance analysis," in *Proc. ACM SIGCOMM*, San Diego, CA, Aug. 2001, pp. 27–38.

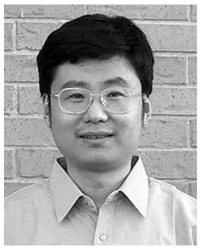
[26] J. Yoon, A. Bestavros, and I. Matta, "Adaptive reliable multicast," in *Proc. IEEE Int. Conf. Communications*, vol. 3, New Orleans, LA, June 2000, pp. 1542–1546.

[27] X. B. Zhang, S. S. Lam, and D.-Y. Lee, "Group rekeying with limited unicast recovery," Dept. Comput. Sci., Univ. Texas, Austin, Tech. Rep. TR-02-36, July 2002. (Revised, Feb. 2003).

[28] X. B. Zhang, S. S. Lam, and D.-Y. Lee, "Group rekeying with limited unicast recovery," in *Proc. IEEE Int. Conf. Communications*, vol. 1, Anchorage, AK, May 2003, pp. 707–714.

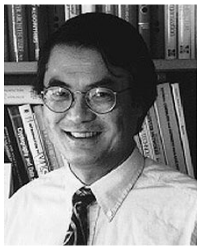
[29] X. B. Zhang, S. S. Lam, D.-Y. Lee, and Y. R. Yang, "Protocol design for scalable and reliable group rekeying," Dept. Comput. Sci., Univ. Texas, Austin, Tech. Rep. TR-02-29, June 2002. (Revised, Nov. 2002).

[30] X. B. Zhang, S. S. Lam, D.-Y. Lee, and Y. R. Yang, "Protocol design for scalable and reliable group rekeying," in *Proc. SPIE Conf. Scalability and Traffic Control in IP Networks*, vol. 4526, Denver, CO, Aug. 2001, pp. 87–108.



**X. Brian Zhang** (S'03) received the B.E. degree in computer science and technology from Tsinghua University, Beijing, China, in 1996. He is currently working toward the Ph.D. degree in computer science at the University of Texas at Austin.

His research interests are in the areas of network security, multicast, and peer-to-peer systems.



**Simon S. Lam** (S'71–M'74–SM'80–F'85) received the B.S.E.E. degree with Distinction from Washington State University, Pullman, in 1969, and the M.S. and Ph.D. degrees in engineering from the University of California at Los Angeles (UCLA) in 1970 and 1974, respectively.

From 1971 to 1974, he was a Postgraduate Research Engineer at the ARPA Network Measurement Center, UCLA, where he worked on satellite and radio packet switching networks. From 1974 to 1977, he was a Research Staff Member at the IBM T. J. Watson Research Center, Yorktown Heights, NY. Since 1977, he has been on the faculty of the University of Texas at Austin, where he is Professor and Regents Chair in Computer Sciences, and served as Department Chair from 1992 to 1994. His current research interests are in network protocol design and analysis, distributed multimedia, and Internet security services.

Dr. Lam has served on the editorial boards of IEEE/ACM TRANSACTIONS ON NETWORKING, IEEE TRANSACTIONS ON SOFTWARE ENGINEERING, IEEE TRANSACTIONS ON COMMUNICATIONS, *Proceedings of the IEEE*, and *Performance Evaluation*. He was Editor-in-Chief of IEEE/ACM TRANSACTIONS ON NETWORKING from 1995 to 1999. He currently serves on the editorial board of *Computer Networks*. He organized and was Program Chair of the inaugural ACM SIGCOMM Symposium held at the University of Texas at Austin in 1983. He is a founding Steering Committee member of the IEEE International Conference on Network Protocols. He received the 1975 Leonard G. Abraham Prize for the best paper published in IEEE TRANSACTIONS ON COMMUNICATIONS, and the 2001 William R. Bennett Prize for the best paper published in IEEE/ACM TRANSACTIONS ON NETWORKING, both from the IEEE Communications Society. He is a Fellow of the ACM (elected 1998).

**Dong-Young Lee** received the B.S. and M.S. degrees in computer science from Seoul National University, Seoul, Korea, in 1998 and 2000, respectively. He is currently working toward the Ph.D. degree in computer science at the University of Texas at Austin.

His research interests include peer-to-peer networks and streaming media.

**Y. Richard Yang** (M'01) received the B.E. degree in computer science and technology from Tsinghua University, Beijing, China, in 1993, and the M.S. and Ph.D. degrees in computer science from the University of Texas at Austin in 1998 and 2001, respectively.

Since 2001, he has been with the Department of Computer Science, Yale University, New Haven, CT, where he is an Assistant Professor. His current research interests are heterogeneous networks, mobile ad-hoc networks, and network security.