

# Protocol Design for Scalable and Reliable Group Rekeying

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

Department of Computer Sciences  
The University of Texas at Austin  
Austin, TX 78712

## 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 period, 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 simple strategy for identifying keys, encryptions, and users, and a key assignment algorithm which 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 rekeying period. For each rekey message, the protocol runs in two steps: a multicast step followed by a unicast step. Proactive FEC multicast is used to control NACK implosion and 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

## 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.<sup>1</sup> 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.<sup>2</sup> 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.<sup>2</sup>

When a user wants to join a group, the user and registration component mutually authenticate each other using a protocol such as SSL.<sup>3</sup> 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.

---

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

Further author information: {zxc, lam, dylee, yangyang}@cs.utexas.edu

In *Proceedings of SPIE Conference on Scalability and Traffic Control in IP Networks*, Vol 4526, Denver, August 2001.

We first consider the key management component, which has been the primary focus of prior work.<sup>4-9</sup> In this paper, we follow the *key tree* approach,<sup>4,5</sup> 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 propose to further reduce processing overhead by using periodic rekeying,<sup>9-11</sup> 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,<sup>2</sup> we designed and implemented a basic protocol which uses proactive FEC to improve the reliability of multicast rekey transport. We also investigated the performance issues of rekey transport<sup>9</sup> and observed that although many reliable multicast protocols have been proposed and studied in recent years,<sup>12-19</sup> 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. \*

The above requirements of rekey transport were considered and analyzed in a companion paper.<sup>9</sup> 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 containing FEC redundant information, (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.<sup>20,21</sup> 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.

---

\*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.

In summary, we have the following contributions. First, we present a detailed specification of a scalable and reliable protocol for group rekeying, together with performance results. Second, a simple key identification strategy and key assignment algorithm are presented and evaluated. Third, we show that our block partition algorithm reduces the server’s FEC encoding time 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 NACK implosion.

The balance of this paper is organized as follows. In Section 2, we briefly review the key tree and periodic batch rekeying ideas. In Section 3 we present our server and user protocols. In Section 4 we show how to construct a rekey message. The key identification strategy and key assignment algorithm are presented. The block partition algorithm is proposed and evaluated in Section 5. In section 6 we discuss how to adaptively adjust proactivity factor to control NACK implosion. In Section 7 we discuss when and how to unicast. Our conclusions are in Section 8.

## 2. BACKGROUND

We review in this section the key tree<sup>4,5</sup> and periodic batch rekeying<sup>9–11</sup> ideas and a 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 tree root. Consider a group with 9 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  $u_9$ .

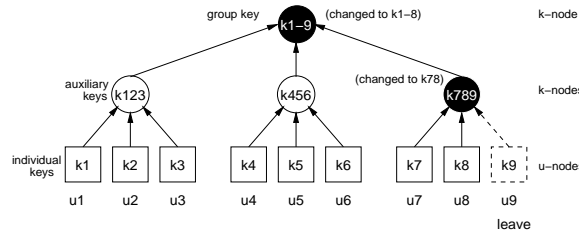


Figure 1. An example key tree

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<sup>5</sup> rekeying strategy, 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 and 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.

In the marking algorithm, the key server first modifies the key tree to reflect the leave and join requests. The u-nodes for departed users are removed, or replaced by u-nodes for newly joined users. When  $J > L$ , the key server will split the nodes after the rightmost k-node at the highest level (with the root as level 0) to expand 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 tree 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.

Appendix B shows the detailed marking algorithm.

### 3. PROTOCOL OVERVIEW

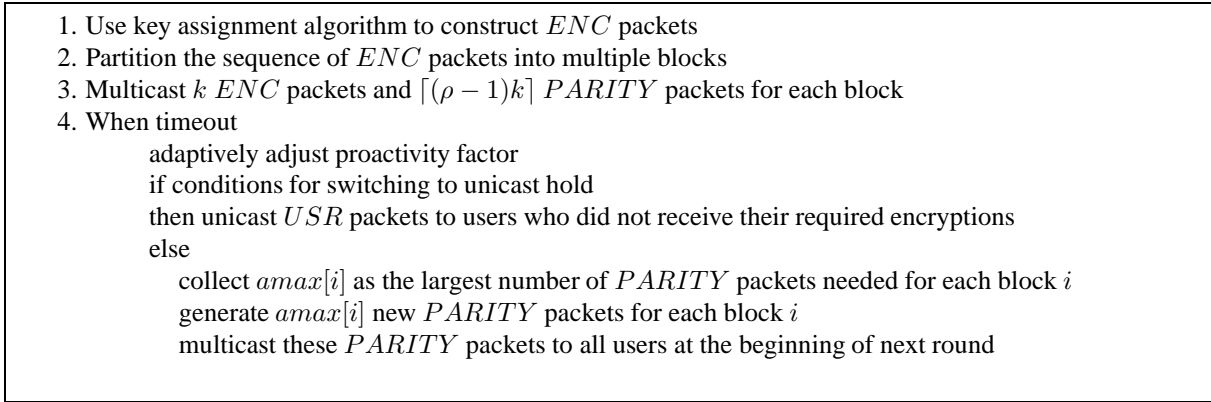
In this section, we give an overview of the rekey transport protocol. An informal specification of the key server protocol is shown in Figure 2.

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.<sup>†</sup> 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  $\lceil(\rho - 1)k\rceil$  *PARITY* packets for each block, where  $\rho$  is the *proactivity factor*.

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 which 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 the *ENC* and *PARITY* packets to the 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 the users who have not recovered their required encryptions.



**Figure 2.** Basic protocol for key server

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 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.

<sup>†</sup>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 other papers.

<p>When timeout</p> <p style="padding-left: 20px;">if received its specific <i>ENC</i> packet, or at least <math>k</math> packets in the required block, or a <i>USR</i> packet</p> <p style="padding-left: 20px;">then success</p> <p style="padding-left: 20px;">else</p> <p style="padding-left: 40px;"><math>a \leftarrow</math> number of <i>PARITY</i> packets needed for recovery</p> <p style="padding-left: 40px;">send <math>a</math> by NACK to the key server</p>
---

**Figure 3.** Basic protocol for a user

In summary, our protocol generates 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 the encryptions for a specific user; 4) *NACK* packet, which is feedback from the users 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.

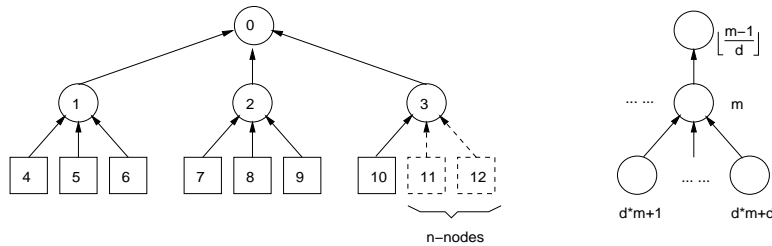
#### 4. CONSTRUCTION OF *ENC* PACKETS

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, and includes ID information 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 to generate *ENC* packets.

##### 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 to inform each user its new ID by sending a packet to the user. This approach, however, is obviously not scalable. By Lemma 4.1 and Theorem 4.2, we show that by knowing the maximum ID of the current k-nodes, each user can derive its new ID independently.

LEMMA 4.1. *If the key server uses the marking algorithm in Appendix B, then in the expanded key tree, the ID of any k-node is always less than the ID of any u-node.*

THEOREM 4.2. *For any user, let  $m$  denote the user's ID before the key server runs the marking algorithm, and  $m'$  denote the 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  $n_k < f(x') \leq d * n_k + d$ . And  $m'$  is equal to  $f(x')$ .*

A proof is shown in Appendix C. By Theorem 4.2, 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 8 fields, and contains both ID information and encryptions.

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

Figure 5. Format of an ENC packet

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 4 uniquely identify a packet. Since rekey messages seldom overlap in time for periodic batch rekeying, we use just 6 bits to identify a rekey message. Field 5 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 6 specifies that this ENC packet contains only the encryptions for users whose IDs are in the range of < frmID, toID > inclusively.

Field 7 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.

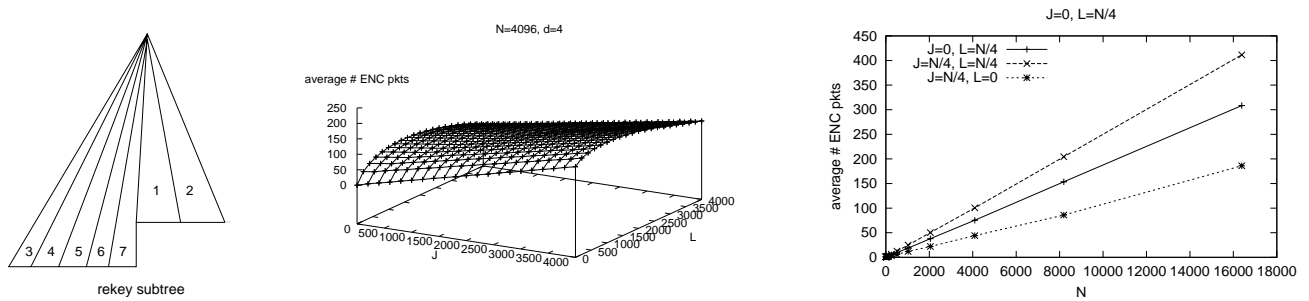
Figure 6(left) illustrates a particular run of the UKA algorithm in which 7 ENC packets are generated. UKA first sorts all of the user IDs into a list in increasing order. Then, a longest prefix of the list is extracted such that all of the encryptions needed by the users in this prefix will fill up an ENC packet. Repeatedly, UKA generates a sequence of ENC packets whose < frmID, toID > intervals do not overlap. In particular, the algorithm guarantees that the 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 into those two ENC packets; therefore, we expect that UKA would increase the bandwidth overhead at the key server.

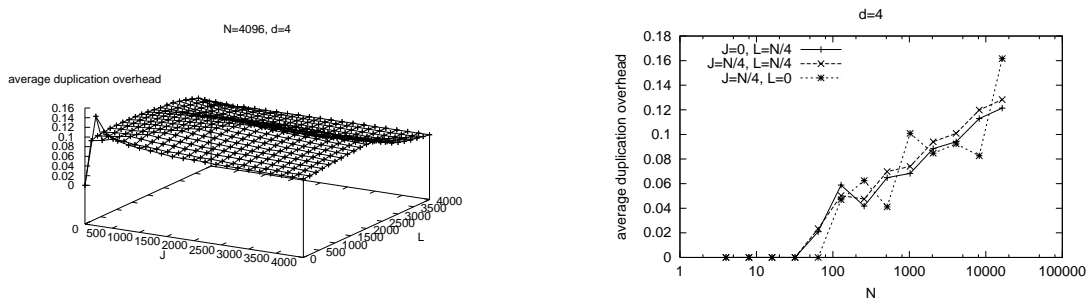
We evaluate the performance of UKA in this subsection using simulations. In the simulations, we assume that at the beginning of a rekey interval the key tree is full and balanced with  $N$  u-nodes. During the rekey interval,  $J$  join and  $L$  leave requests are processed. We further assume that the 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 1027 bytes in our experiments.



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

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  $J$ . To understand such linear increase, we need to investigate the size of the rekey subtree as a function of  $J$ . For a given number of leave requests, our marking algorithm first replaces departed u-nodes with newly joined u-nodes, and then splits the n-nodes or u-nodes at the highest level (with the root as level 0). As a result, the rekey subtree and consequently the number of ENC packets grow proportionally to  $J$  as we increase  $J$ . For a fixed  $J$ , we observe that as  $L$  increases, the number of ENC packets first increases (because more leaves means more keys to be changed), and then decreases (because now some keys can be pruned from the rekey subtree). The maximum of the number of packets is achieved when about  $N/d$  users leave the group.

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 of  $J$  and  $L$  values.



**Figure 7.** Average duplication overhead as a function of  $J$  and  $L$  (left) and as a function of  $N$  (right)

We then evaluate the duplications generated by the UKA algorithm. Define duplication overhead as the ratio of duplicated encryptions over 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  $L$ . From our marking algorithm, we know that a larger

value of  $J$  will generate a larger rekey subtree, and a larger rekey subtree will have more shared encryptions. Therefore the numerator of the duplication overhead will increase as we increase  $J$ . On the other hand, the number of encryptions, which is the denominator of our duplication overhead, increases at a faster speed as implied by Figure 6 (left). Consequently, we observe that the duplication overhead decreases roughly from 0.1 to 0.05 as we increase  $J$ . Next consider the case of a fixed  $J$ . As implied by Figure 6 (left), for a given  $J$ , a rekey subtree will first grow and then shrink as we increase  $L$ ; therefore, the number of duplications will also first increase and then decrease. However, since the number of duplications changes at a faster speed than the number of encryptions does, 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 contained in an *ENC* packet with a packet size of 1027 bytes. For  $J = N/4, L = 0$ , the rekey subtree is very sparse, and thus the curve of duplication overhead fluctuates around the curve of  $J = L = N/4$ .

## 5. BLOCK PARTITIONING

After running the *UKA* assignment algorithm to generate *ENC* packets, 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 bandwidth overhead, a large block size can significantly increase encoding and decoding time.<sup>22,23,17</sup> For example, using the RSE coder by L. Rizzo,<sup>22</sup> 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  $J = 0$  and  $L = N/4$ , the key server can generate up to 107 *ENC* packets with a packet size of 1027 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.

Below we first present our block partition algorithm for a given block size  $k$ . Then we discuss how to choose the block size.

### 5.1. The block partition algorithm

For a given block size  $k$ , to reduce the encoding time at the key server, we partition the *ENC* packets into multiple blocks. The key server first sorts the *ENC* packets according to their generating orders. Then during each iteration, the key server first increases the current block ID, takes  $k$  packets from the top of the *ENC* packets to form a block, assigns these *ENC* packets the current block ID, and increases sequentially the sequence number within the current block. To form the last block, the key server may need to duplicate the *ENC* packets in the last block until there are  $k$  packets.<sup>‡</sup>

One issue of partitioning the *ENC* packets into blocks is that if a user loses its *ENC* packet, the user will not be able to know directly the block to which its *ENC* packet belongs, so the user needs to estimate the block ID to which its *ENC* specific packet belongs. Our estimation algorithm guarantees that even if the user cannot determine the accurate value of its block ID, which happens with a very low probability, the user can still estimate a possible range of the block ID. For this case, during feedback, the user will then require *PARITY* packets for each block within this range. For an algorithm to estimate block ID, we refer the interested readers to Appendix D.

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

<sup>‡</sup>A flag bit may be used in an *ENC* packet to specify whether the packet is a duplicate. A duplicated *ENC* packet will be used in FEC decoding performed by users, but will not be used for block ID estimation. Also the key server may distribute such duplicates over several blocks.

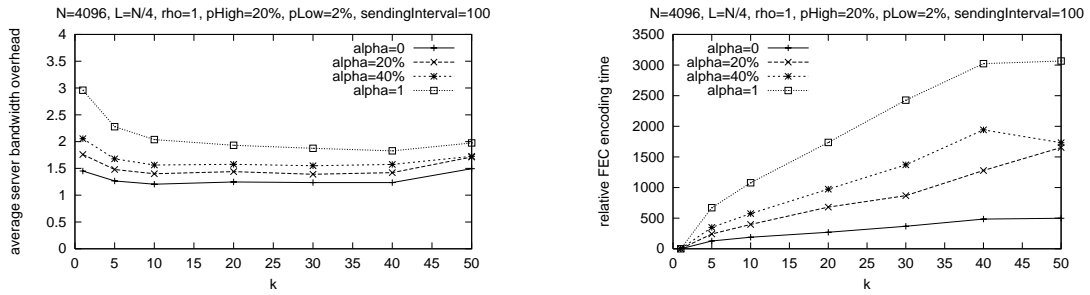
## 5.2. Choosing block size

The block partition algorithm discussed in the previous subsection operates for a given block size  $k$ . To determine the block size, we need to evaluate the impact of block size on two performance metrics.

Our first performance metric is the key server's multicast bandwidth overhead, which is defined as the ratio of  $h'$  and  $h$ , where  $h$  is the number of *ENC* packets in a rekey message, and  $h'$  is the total number of packets that the key server multicasts to make all of the users recover their specific *ENC* packets. To evaluate the bandwidth overhead, we only consider the impact of block size  $k$  and use  $\rho = 1$  for the experiments in this section. We observe that the average bandwidth overhead at the key server for  $\rho = 1$  is typically no more than that for  $\rho > 1$ . The joined effects of block size  $k$  and adaptive  $\rho$  on the key server bandwidth overhead will be evaluated in Section 6.

Our 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 direct impact on the users' FEC decoding time, the impact is small because in our protocol a vast majority of users can receive their specific *ENC* packets, and thus do not have any decoding overhead.

We use simulations to evaluate the effects of block size. We adopt the network topology used by J. Nonnenmacher, etc.<sup>19</sup> which is a tree hierarchy that connects the key server to a backbone network through a source link, and connects each user to the backbone through a receiver link. As for our loss model, we assume that the source link has a fixed loss rate of  $p_s$ , and the backbone is loss-free. We assume that a fraction  $\alpha$  of the  $N$  users have a high loss rate of  $p_h$ , and the others have a low loss rate of  $p_l$ . At a given loss rate, we use a two-state continuous time Markov chain<sup>17</sup> to simulate burst loss. We assume that the average duration of a burst loss is  $\frac{100}{p}$  msec, and the average duration of a loss-free time is  $\frac{100}{1-p}$  msec, where  $p$  is the link loss rate. The default values of our simulations are  $N = 4096$ ,  $d = 4$ ,  $J = 0$ ,  $L = N/d$ ,  $\alpha = 20\%$ ,  $p_h = 20\%$ ,  $p_l = 2\%$ ,  $p_s = 1\%$ , the key server's sending rate is 10 packets/second, and the length of an *ENC* packet is 1027 bytes. These simulation topology and parameters will also be used in the experiments in the following sections unless stated otherwise.



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

We first consider the effects of block size on the key server's bandwidth overhead. As shown in Figure 8 (left), we observe that the key server's average bandwidth overhead is not sensitive to the block size for  $k \geq 5$ . To explain this phenomenon, we observe that  $k$  has the following two effects on the key server's bandwidth overhead. First, as  $k$  decreases, the number of users who need to receive their *ENC* packets from a given block decreases, and thus a *PARITY* packet can only recover a smaller number of users. Therefore, we expect a higher bandwidth overhead as block size decreases. On the other hand, as block size decreases, the number of packets that a user needs in order to recover its lost *ENC* packet decreases. Therefore, we expect the bandwidth overhead for retransmission decreases. For  $k \geq 5$ , these two factors almost balance each other, and we observe a flat bandwidth overhead curve. The high bandwidth overhead for  $k = 50$  comes from the duplicated *ENC* packets in the last block.

We next consider the effects of block size on the key server's overall FEC encoding time. If we use L. Rizzo's RSE coder,<sup>22</sup> the encoding time for one *PARITY* packet is approximately a linear function of block size  $k$ . Therefore, the overall total encoding time for all *PARITY* packets is the product of the total number of *PARITY* packets and the encoding time for one *PARITY* packet. Now consider the total number of *PARITY* packets. 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. As implied by the flat bandwidth overhead curves in Figure 8 (left), the total number of *PARITY* packets is not sensitive to block size  $k$  for  $k \geq 5$ . Therefore, we anticipate that the overall encoding time for all *PARITY* packets will be approximately a linear function of block size  $k$ . Figure 8 (right) confirms our analysis and shows the normalized overall encoding time (assuming  $k$

time units to generate one *PARITY* packet for block size  $k$ ). We also observe from this figure some irregularity on the overall encoding time for a large block size. Such irregularity can be explained by the duplicated packets in the last block, which do not need any encoding time.

In summary, we observe that for  $\rho = 1$ , a small block size  $k$  can be chosen to provide fast FEC encoding without a large bandwidth overhead. For the following experiments, we choose  $k = 10$  as the default value.

## 6. ADAPTIVE PROACTIVE FEC MULTICAST

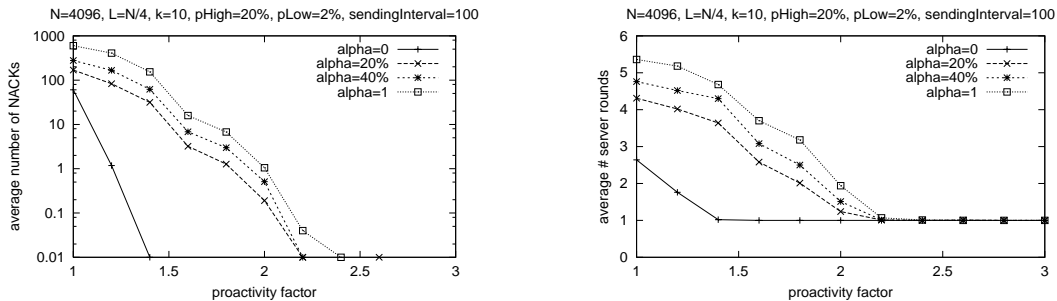
In the previous section, we discussed how to partition the *ENC* packets of a rekey message into blocks and generate  $\lceil(\rho - 1)k\rceil$  *PARITY* packets for each block. The discussion, however, assumes a given proactivity factor  $\rho$ . In this section, we investigate how to determine  $\rho$ .

We observe that  $\rho$  is an effective mechanism for controlling NACK implosion at the key server. Feedback implosion occurs when many users simultaneously send feedbacks to the key server. Mechanisms such as structure-based feedback aggregation or timer-based NACK suppression<sup>14,24–26</sup> have been proposed to reduce feedback traffic. However, structure-based mechanisms rely on a tree hierarchy of designated nodes to aggregate feedbacks. In timer-based NACK suppression mechanisms, users use a random delay timer to avoid feedback implosion. The extra delay introduced by delay timers makes it harder for users to meet the soft deadline of group rekeying. In this section, we present our algorithm to adaptively adjust the proactivity factor to avoid NACK implosion.

### 6.1. Impact of proactivity factor

To design the algorithm to adapt  $\rho$ , we need to first evaluate the impact of  $\rho$  on the number of NACKs, the delivery latency at users, and the bandwidth overhead at the key server.

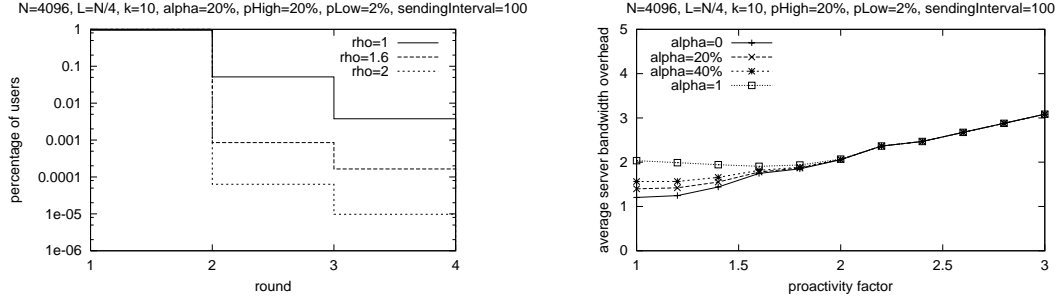
We first evaluate the impact of  $\rho$  on the number of NACKs. Figure 9 (left) plots the average number of NACKs received at the key server at the end of the first round. Note that y-axis is in log scale. We observe that the average number of NACKs decreases exponentially as a function of  $\rho$  (a previous study has made a similar observation<sup>21</sup>). One conclusion we draw from this figure is that the number of NACKs is very sensitive to  $\rho$ , and a small increase of  $\rho$  can substantially reduce the number of NACKs.



**Figure 9.** Average number of NACKs of the first round (left) and average number of rounds for all users to receive their encryptions (right) as a function of  $\rho$

We next evaluate the impact of  $\rho$  on delivery latency. Figure 9 (right) plots the average number of rounds for all users to receive their encryptions. From this figure, we observe that the average number of rounds decreases linearly as we increase  $\rho$ , until  $\rho$  is large enough so that the effect of  $\rho$  diminishes and the curve levels off. Figure 10 (left) plots the percentage of users who need a given number of rounds to receive their encryptions. The x-axis is the number of rounds for a user to receive its encryptions and the y-axis is the percentage of users who need the number of rounds. For  $\rho = 1$ , we observe that more than 94.4% 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  $\rho$  on the average server bandwidth overhead, as shown in Figure 10 (right). For a small  $\rho$ , 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 the subsequent rounds to allow users to recover their packets. Therefore, we observe that the increase of  $\rho$  has little effect on the average server bandwidth overhead. When  $\rho$  becomes large, however, the bandwidth overhead during



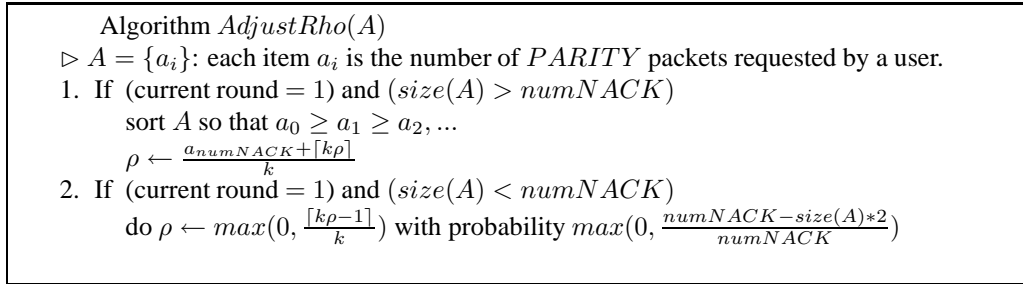
**Figure 10.** Percentage of users who need specific rounds to receive their encryptions (left) and average server bandwidth overhead as a function of  $\rho$  (right)

the first round can dominate the overall bandwidth overhead, and we observe that 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; 2) It will slightly reduce delivery latency; 3) It will increase the key server's bandwidth overhead when  $\rho$  is larger than needed.

## 6.2. Adjustment of proactivity factor

Motivated by the effect of the proactivity factor  $\rho$ , we present in Figure 11 our algorithm to adaptively adjust  $\rho$ . The basic idea of the algorithm is to adjust  $\rho$  such that the key server will receive a target number of NACKs.



**Figure 11.** The algorithm to adaptively adjust proactivity factor

The input to the algorithm *AdjustRho* is a list  $A$ . Each item of  $A$  is the number of *PARITY* packets requested by a user. If a user requests packets for a range of blocks, the maximum number of *PARITY* packets requested by the user is recorded into  $A$ . For example, if a user requests 2 *PARITY* packets for block 1 and 4 packets for block 2, then 4 will be recorded into  $A$ .

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

In the first case, the key server receives more NACKs than it targets. For this case, the server selects the  $(numNACK + 1)^{th}$  largest item (denoted by  $a_{numNACK}$ ) from  $A$ , and increases  $\rho$  so that  $a_{numNACK}$  additional proactive *PARITY* packets will be generated for each block of the next rekey message. Consider this example. Assume that 10 users  $u_i, i = 0, \dots, 9$  send NACKs for the current rekey message, and user  $u_i$  requests  $a_i$  *PARITY* packets. For illustration purposes, we assume  $a_0 \geq a_1 \geq \dots \geq a_9$ , and the number of NACKs that the key server targets 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  $\{u_2, u_3, \dots, u_9\}$  have a higher probability to recover their *ENC* packets within a single round. This is because according to the current rekey message, if users  $\{u_2, u_3, \dots, u_9\}$  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 it targets. Although receiving less NACKs is better in terms of avoiding NACK implosion, the small number of NACKs received may imply that the current proactivity factor is too high, and thus may cause high bandwidth overhead. Therefore, the key server reduces  $\rho$  by one *PARITY* packet with probability proportional to  $\frac{numNACK - size(A) * 2}{numNACK}$ .

Our algorithm *AdjustRho* will not only adjust  $\rho$  according to *numNACK*, but also adjust *numNACK* dynamically. We observe that a smaller *numNACK* implies a smaller number of NACKs, and therefore a smaller average delivery latency. On the other hand, a smaller *numNACK* may also imply a larger server bandwidth overhead. Given these, we update *numNACK* according to the following heuristics (where *maxNACK* is an upper bound of *numNACK*):

1. If all users meet deadline for the current rekey message, the key server updates *numNACK* as  $\min(numNACK + 1, maxNACK)$  to save server bandwidth overhead.
2. If *i* receivers miss deadline, the key server updates *numNACK* as  $\max(numNACK - i, 0)$  to increase the number of users who can meet deadline.

If *numNACK* exceeds the upper bound, the server may experience NACK implosion. The exact value of *maxNACK* will be a configuration parameter and depend on a key server's available bandwidth and processing power.

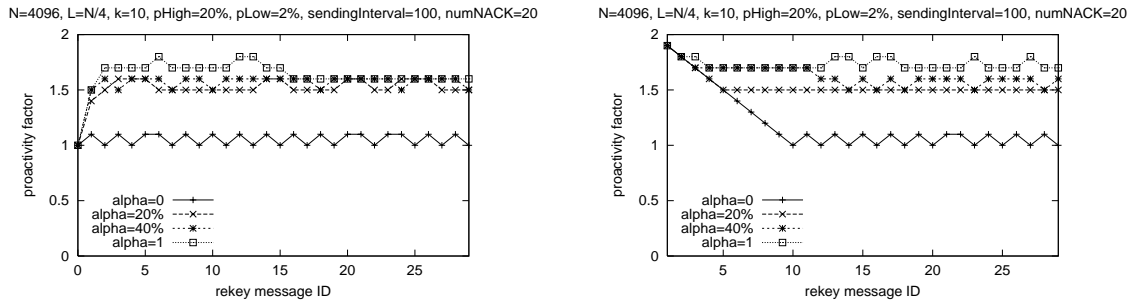
### 6.3. Performance evaluation

We use simulations to evaluate the *AdjustRho* algorithm. In Section 6.3.1, we evaluate the performance of our algorithm to avoid NACK implosion. In Section 6.3.2, we investigate how to choose block size *k* for the adaptive  $\rho$  scenario. In Section 6.3.3, we investigate how to choose *maxNACK*. In Section 6.3.4, we evaluate the server bandwidth overhead for adaptive proactive FEC.

#### 6.3.1. Controlling NACK implosion

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

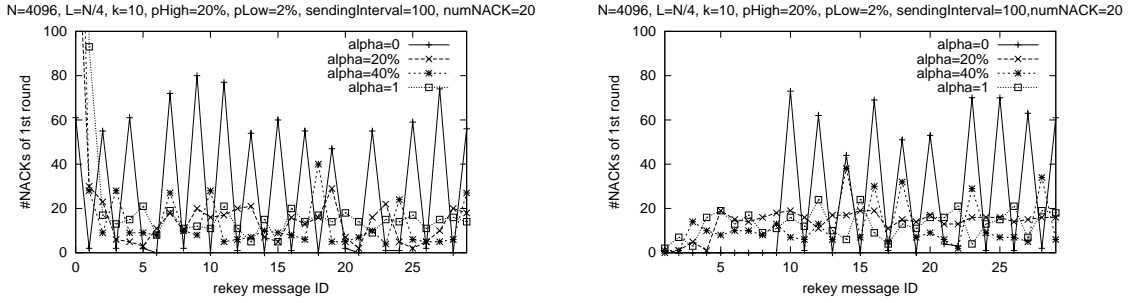
Figure 12 shows how  $\rho$  is adaptively adjusted when the key server sends a sequence of rekey messages. For initial  $\rho = 1$  as shown in the left figure, we observe that it takes only a couple of rekey messages for  $\rho$  to settle down to stable values. For initial  $\rho = 2$  as shown in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



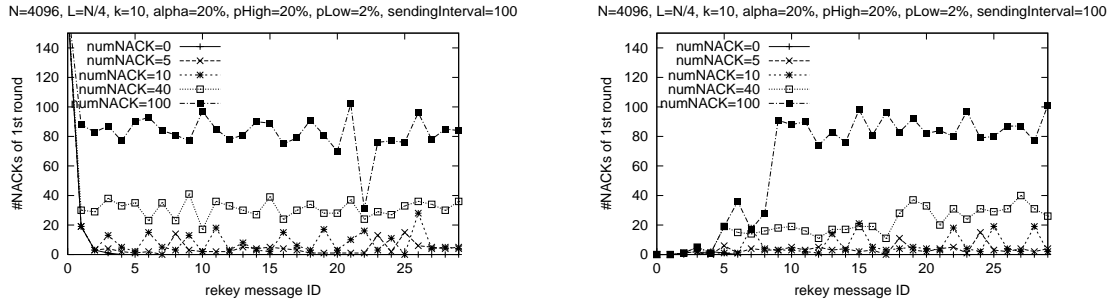
**Figure 12.** Adjusting of proactivity factor, initially  $\rho = 1$  (left) and initially  $\rho = 2$  (right)

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  $\rho$  is 1, the number of NACKs received become stabilized very quickly, and the stable values are generally less than 1.5 times of *numNACK* for  $\alpha > 0$ . For  $\alpha = 0$ , which means all users have a low loss rate of 2%, the number of NACKs fluctuates in a large range because the number of users who can receive their encryptions during the first round is very sensitive to  $\rho$  for small loss rate. The sharp slope of the curve for  $\alpha = 0$  in Figure 9 (left) conforms this sensitivity. The right figure shows the case for initial  $\rho = 2$ . We observe that the stable values of those two figures match very well.

Next we evaluate whether *AdjustRho* algorithm can control the number of NACKs for various *numNACK*s. As shown in the left figure (initially  $\rho$  is 1) and right figure (initially  $\rho$  is 2) of Figure 14, the numbers of NACKs received at the key server



**Figure 13.** Number of NACKs received, initially  $\rho = 1$  (left) and initially  $\rho = 2$  (right)

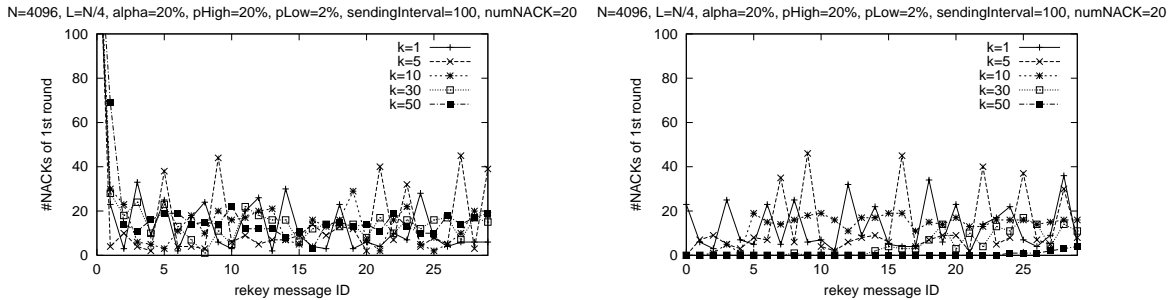


**Figure 14.** Number of NACKs for different  $numNACK$ s, initially  $\rho = 1$  (left) and initially  $\rho = 2$  (right)

fluctuate around the target value for a wide range of  $numNACK$  values. However, we do observe that the fluctuations become more significant for larger  $numNACK$ . Therefore, to choose  $maxNACK$ , we need to consider the potential impact of large fluctuations when  $maxNACK$  is large. In our following experiments, we choose 20 as the default value of  $numNACK$ .

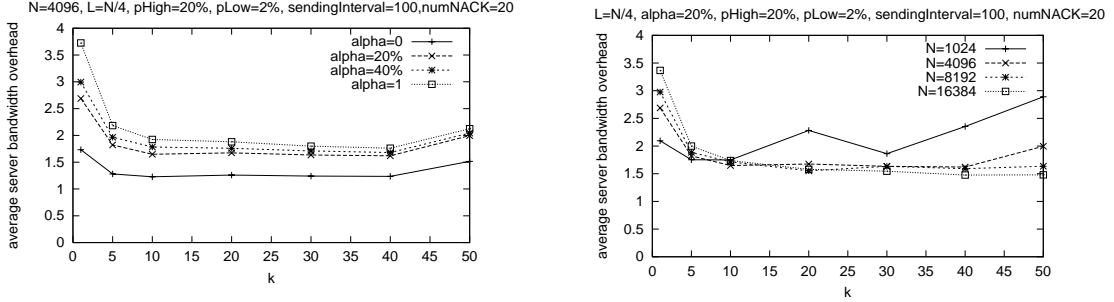
### 6.3.2. Choosing block size

In section 5.2, we have discussed how to choose block size  $k$  for  $\rho = 1$ . In this section, we reconsider this problem for a new scenario where  $\rho$  is adaptively adjusted. To determine the block size, we consider the following factors:

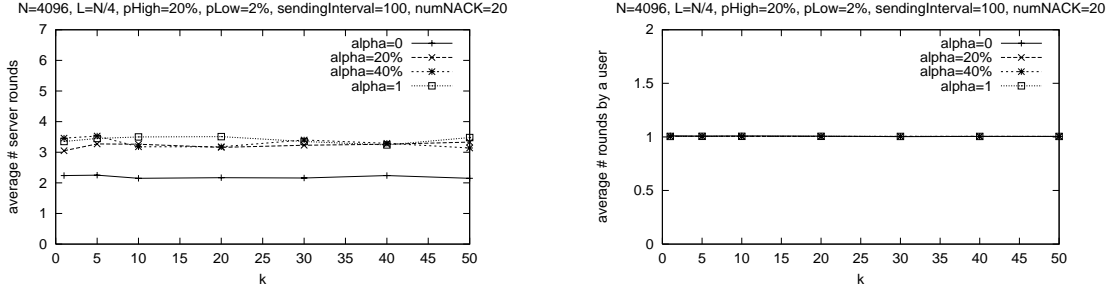


**Figure 15.** Average number of NACKs for different  $k$ , initially  $\rho = 1$  (left) and initially  $\rho = 2$  (right)

1. Fluctuations in the number of NACKs received. As shown in 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  $numNACK$ .
2. Server bandwidth overhead. First consider Figure 16 (left), which shows the average bandwidth overhead at the key server as a function of  $k$  when  $\rho$  is adaptively adjusted. We observe that the average server bandwidth overhead is very high for  $k = 1$ ; then it decreases and becomes flat as we increase  $k$ . The higher bandwidth overhead for  $k = 50$  comes from the duplicated packets in the last block. This observation is almost the same as what we see for  $\rho = 1$ , except that



**Figure 16.** Average bandwidth overhead as a function of block size for different  $\alpha$  (left) and for different  $N$  (right)



**Figure 17.** Average number of rounds for all users to receive their encryptions (left) and average number of rounds needed by a user (right) as a function of  $k$

the bandwidth overhead for  $k = 1$  is much higher in the adaptive  $\rho$  scenario. This is because for  $k = 1$ , one block contains only one *ENC* packet. When the key server increases  $\rho$ , the key server will have to generate at least one more *PARITY* packet for each block (that is, for just one *ENC* packet) to have any effect. For example, for initial  $\rho = 1$ , any increase of  $\rho$  will at least double the total number of packets sent during the first round. Given such large granularity adjustment, the key server experiences high bandwidth overhead for  $k = 1$ .

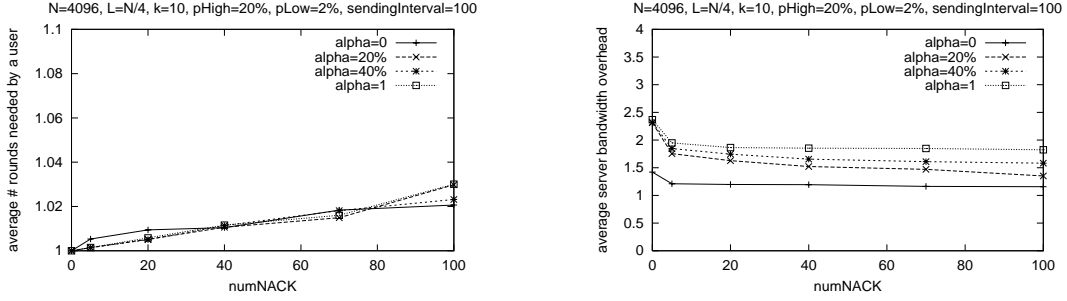
Next consider Figure 16 (right), which shows the average server bandwidth overhead as a function of group size when  $\rho$  is adaptively adjusted. The figure shows the same trend as the left figure. However, the average server bandwidth overhead fluctuates a lot for  $N = 1024$ . This is because the rekey message contains only 26 *ENC* packets for  $N = 1024$ ,  $J = 0$ ,  $L = N/4$ . Hence the duplicated packets in the last block can significantly affect the bandwidth overhead at the key server if block size is large.

3. 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 bandwidth overhead at the key server is not sensitive to block size. As a result, the overall FEC encoding time at the key server will linearly increase with block size  $k$ .
4. 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, the change of block size  $k$  does not have much impact on the delivery latency at users. To further validate this result, Figure 17 (right) plots the average number of rounds for a single user to receive its encryptions. From this figure, we again observe that block size  $k$  does not have any noticeable effect on delivery latency. We further notice that using our algorithm, the average number of rounds for a user to receive its encryptions is close to 1.

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

### 6.3.3. Choosing *maxNACK*

Next, we evaluate the impact of *maxNACK*, which is the upper bound of *numNACK*. The following observations about *numNACK* will help us choose *maxNACK*.



**Figure 18.** Average number of rounds needed by a user (left) and average server bandwidth overhead (right) as a function of  $numNACK$

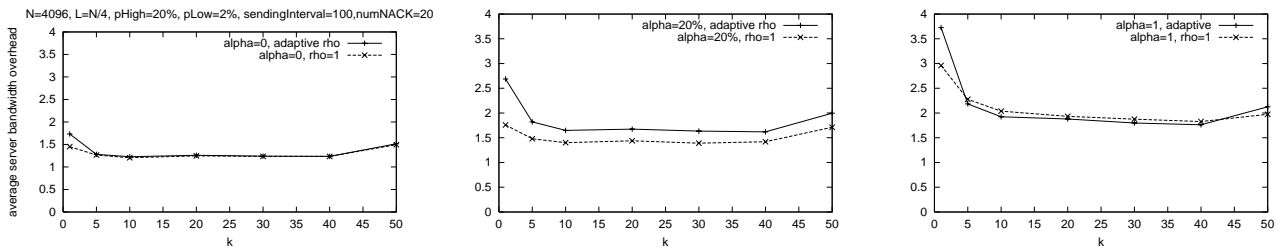
1. First, a large  $numNACK$  can cause NACK implosion.
2. Second, as observed in Section 6.3.1, a large  $numNACK$  can cause large fluctuations in the number of NACKs received by the server.
3. Third, an increase of  $numNACK$  will slightly increase the delivery latency. From Figure 18 (left), we observe that the average number of rounds for a user to receive its encryptions will increase linearly with  $numNACK$ . However, the increasing speed is very slow because more than 94.4% users can receive their encryptions within a single round even for  $\rho = 1$  when  $\alpha = 20\%$ .
4. Fourth, a very small  $numNACK$  may cause a large average bandwidth overhead at the key server. As shown in Figure 18 (right), we observe that for  $numNACK = 0$  and  $\alpha > 0$ , the average overhead at the key server bandwidth overhead can be as high as 2.3. However, as we increase  $numNACK$ , the bandwidth overhead decreases and becomes flat for  $numNACK \geq 5$ .

Given the above observations, we know that  $maxNACK$  should be large enough ( $maxNACK \geq 5$ ) to avoid large bandwidth overhead at the key server. On the other hand,  $maxNACK$  should be small enough to avoid NACK implosion, and also to reduce fluctuations in the number of NACKs.

The above observations can also help us choose the initial value of  $numNACK$ . If the rekey interval is small, a small  $numNACK$  is desired because delivery latency is the major performance metric for a small rekey interval. Otherwise, a large initial value of  $numNACK$  can be chosen to reduce the bandwidth overhead at the key server.

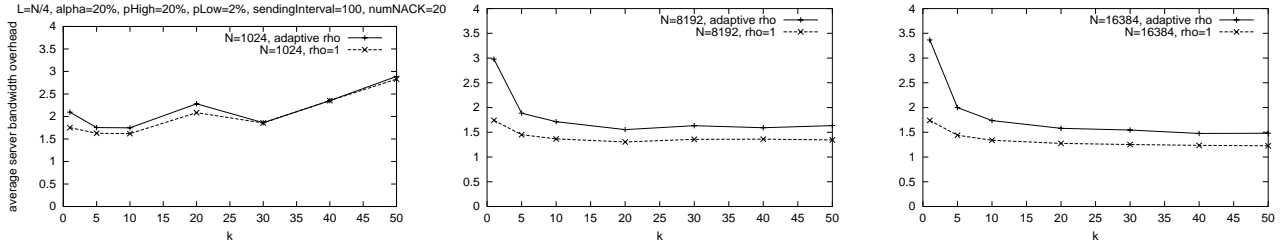
### 6.3.4. Overhead of adaptive proactive FEC

From the previous section, we know that the *AdjustRho* algorithm can effectively avoid NACK implosion and 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 adaptive proactive FEC scheme may incur extra bandwidth overhead at the key server. We investigate this issue in this section.



**Figure 19.** The extra server bandwidth caused by adaptive  $\rho$  for different  $\alpha$

We first evaluate the extra bandwidth overhead at the key server caused by proactive FEC. Figure 19 shows the results for various loss environments. We observe that compared with the approach where all *PARITY* packets are generated reactively



**Figure 20.** The extra server bandwidth caused by adaptive  $\rho$  for different  $N$

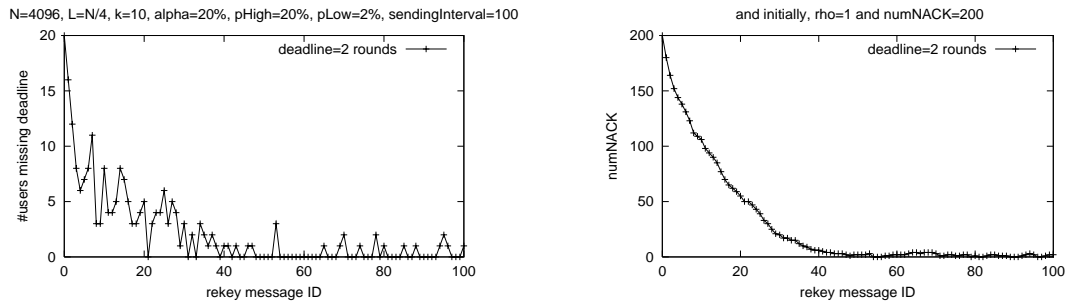
(i.e.  $\rho = 1$ ), our adaptive proactive scheme causes little extra server bandwidth overhead in homogeneous low loss situations (i.e.  $\alpha = 0$ ). For  $\alpha = 1$ , our scheme can even save a little bandwidth. This is because for  $\rho = 1$ , the key server takes much more rounds for all users to recover their encryptions than the adaptive proactive scheme. And during each round, the key server generates *PARITY* packets according to the maximum number requested. Therefore, it is possible that the total number of *PARITY* packets generated during so many rounds for  $\rho = 1$  is larger than the adaptive proactive scheme. In a heterogeneous environment such as  $\alpha = 20\%$ , the extra bandwidth overhead generated by adaptive  $\rho$  is less than 0.25 for  $k \geq 5$ .

We next consider the server bandwidth overhead for various group size. From Figure 20, we observe that the extra bandwidth overhead incurred by adaptive  $\rho$  increases with  $N$ . The extra bandwidth overhead, however, is still less than 0.4 even for  $N = 16384$ .

## 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, in the previous section, we have proposed to adaptively adjust  $numNACK$  and  $\rho$  during the multicast phase to reduce the the number of users missing deadline. However, these approaches may not be enough because  $numNACK$  does not directly affect delivery latency at users.

Consider Figure 21, which shows the number of users missing deadline and the adjustment of  $numNACK$ , when we set the deadline at 2 rounds, the initial  $\rho$  as 1, and initial  $numNACK$  as 200 (a rather high initial value). We observe that the number of users missing deadline reduces dramatically during the first few rekey messages because of the rapid decrease of  $numNACK$ . However, when  $numNACK$  becomes stable, there are still a few users missing deadline.



**Figure 21.** Number of users who missed deadline (left) and adjusting of  $numNACK$  (right), for initially  $\rho$  is 1,  $numNACK$  is 200

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 a duration of a multicast round is typically larger than the largest 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 the users can quickly and efficiently receive their encryptions.

## 7.1. When to switch to unicast

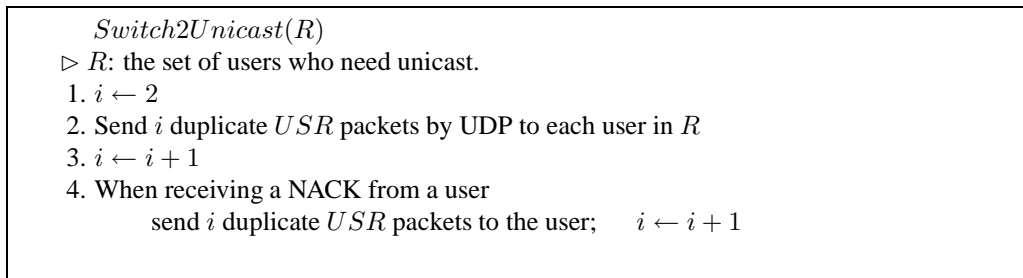
One issue of early unicast is its possible high bandwidth overhead at the key server. This is because the server has to send separate packets for each user who did not receive their encryptions.

However, in our protocol, unicast will not cause large bandwidth overhead at the key server due to the following two reasons. First, the packet size of the *USR* packets sent during unicast is much smaller than those of *ENC* and *PARITY* packets. In our protocol, a *USR* packet contains only the encryptions for a specific user, and its packet size is at most  $(3 + 20h)$  bytes, where  $h$  is the height of the key tree. On the other hand, the packet sizes of *ENC* and *PARITY* packet are typically more than one kilobyte long. Second, our protocol guarantees that only a few users need unicast. From Figure 10, we observe that the vast majority of the users can receive their encryptions within a single multicast round. In fact, our evaluations show that for  $numNACK = 20$ , roughly 5 or less users need retransmissions after two multicast rounds.

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 (plus UDP headers) is no more than that of *PARITY* packets needed for the next multicast round.

For our protocol, the duration of each multicast round is not fixed. It is adjusted so that all users are expected to meet deadline. In particular, if some users miss deadline, we propose that the duration of a round should be reduced by the missing time; otherwise, the duration of a round should be increased by a small value.

## 7.2. Unicast protocol



**Figure 22.** Unicast protocol for the key server

The server's unicast protocol is shown in Figure 22. To provide fast delivery, the key server sends an increasing number of duplicated *USR* packets during unicast. In particular, the server sends two duplicate *USR* packets over a certain interval to each user in  $R$  at the beginning of unicast. The number of duplicates will then be incremented by 1 for each subsequent retransmission. Since the size of  $R$  and the length of a *USR* packet is small, such duplications will not cause large bandwidth and processing overheads for the key server.

## 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 for 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 rate 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 *ENC* and *PARITY* packets to all users. The major problem in this phase is to control NACK implosion. 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, while the extra bandwidth overhead incurred is small. Additionally, adaptively adjusting the proactivity factor reduces the delivery latency of users.

In the fourth phase, the key server switches to unicast to reduce the worst-case delivery latency. The major problem in this phase is to determine when to switch to unicast such that unicast will not cause large server bandwidth overhead, and how to do unicast to provide smaller delivery latency. In our protocol, a vast majority of users can receive or recover their required encryptions within one multicast round. Also, during unicast the *USR* packet transmitted is very small. Based on these observations, we let the key server switch to unicast after one or two multicast rounds (depending upon deadline). To provide even faster delivery, the key server duplicates the *USR* packet during unicast.

In summary, we have the following contributions. First, we present a detailed specification of a scalable and reliable protocol for group rekeying, together with performance results. Second, a simple key identification strategy and key assignment algorithm are presented and evaluated. Third, we show that our block partition algorithm reduces the server's FEC encoding time 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 NACK implosion.

## ACKNOWLEDGMENTS

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

## REFERENCES

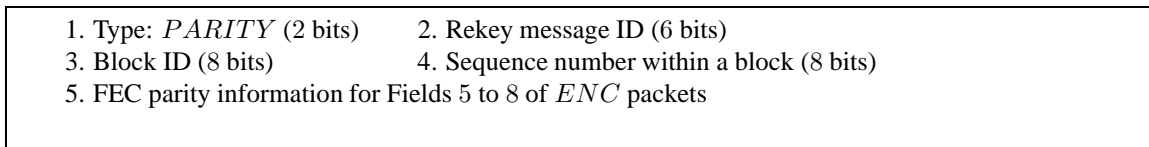
1. I. R. T. F. (IRTF), "The secure multicast research group (SMuG)." <http://www.ipmulticast.com/community/smug/>.
2. C. K. Wong and S. S. Lam, "Keystone: a group key management system," in *Proceedings of ICT 2000*, (Acapulco, Mexico), May 2000.
3. A. O. Freier, P. Karlton, and P. C. Kocher, "The SSL protocol version 3.0." Work in progress, IETF Internet-Draft, Mar. 1996.
4. D. Wallner, E. Harder, and R. Agee, *Key Management for Multicast: Issues and Architectures*, *INTERNET-DRAFT*, Sept. 1998.
5. C. K. Wong, M. G. Gouda, and S. S. Lam, "Secure group communications using key graphs," in *Proceedings of ACM SIGCOMM '98*, Sept. 1998.
6. H. Harney and E. Harder, *Logical Key Hierarchy Protocol*, *INTERNET-DRAFT*, Mar. 1999.
7. I. Chang, R. Engel, D. Kandlur, D. Pendarakis, and D. Saha, "Key management for secure Internet multicast using boolean function minimization techniques," in *Proceedings of IEEE INFOCOM '99*, vol. 2, Mar. 1999.
8. D. Balenson, D. McGrew, and A. Sherman, *Key Management for Large Dynamic Groups: One-way Function Trees and Amortized Initialization*, *INTERNET-DRAFT*, 1999.
9. Y. R. Yang, X. S. Li, X. B. Zhang, and S. S. Lam, "Reliable group rekeying: A performance analysis," in *Proceedings of ACM SIGCOMM 2001*, (San Diego, CA), Aug. 2001.
10. X. S. Li, Y. R. Yang, M. G. Gouda, and S. S. Lam, "Batch rekeying for secure group communications," in *Proceedings of Tenth International World Wide Web Conference (WWW10)*, (Hong Kong, China), May 2001.
11. S. Setia, S. Koussih, S. Jajodia, and E. Harder, "Kronos: A scalable group re-keying approach for secure multicast," in *Proceedings of IEEE Symposium on Security and Privacy*, (Berkeley, CA), May 2000.
12. I. R. T. F. (IRTF), "Reliable Multicast Research Group." <http://www.nard.net/tmont/rm-links.html>.
13. S. Paul, K. Sabnani, and D. Kristol, "Multicast transport protocols for high speed networks," in *Proceedings of IEEE ICNP '94*, (Boston, MA), Oct. 1994.
14. 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), pp. 784–803, 1997.
15. 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), pp. 398–406, 1997.
16. B. Levine and J. Garcia-Luna-Aceves, "A comparison of known classes of reliable multicast protocols," in *Proceedings of IEEE ICNP '96*, (Columbus, OH), Oct. 1996.
17. J. Nonnenmacher, E. Biersack, and D. Towsley, "Parity-based loss recovery for reliable multicast transmission," in *Proceedings of ACM SIGCOMM '97*, Sept. 1997.

18. S. K. Kasera, J. Kurose, and D. Towsley, "A comparison of server-based and receiver-based local recovery approaches for scalable reliable multicast," in *Proceedings of IEEE INFOCOM '98*, (San Francisco, CA), Mar. 1998.
19. J. Nonnenmacher, M. Lacher, M. Jung, E. Biersack, and G. Carle, "How bad is reliable multicast without local recovery?," in *Proceedings of IEEE INFOCOM '98*, (San Francisco, CA), Mar. 1998.
20. R. G. Kermodé, "Scoped Hybrid Automatic Repeat reQuest with Forward Error Correction (SHARQFEC)," in *Proceedings of ACM SIGCOMM '98*, Sept. 1998.
21. D. Rubenstein, J. Kurose, and D. Towsley, "Real-time reliable multicast using proactive forward error correction," in *Proceedings of NOSSDAV '98*, July 1998.
22. L. Rizzo, "Effective erasure codes for reliable computer communication protocols," *Computer Communication Review*, Apr. 1997.
23. J. W. Byers, M. Luby, M. Mitzenmacher, , and A. Rege, "A digital fountain approach to reliable distribution of bulk data," in *Proceedings of ACM SIGCOMM '98*, (Vancouver, B.C.), Sept. 1998.
24. M. Grossglauser, "Optimal deterministic timeouts for reliable scalable multicast," in *Proceedings of IEEE INFOCOM '96*, Mar. 1998.
25. H. W. Holbrook, S. K. Singhal, and D. R. Cheriton, "Log-based receiver-reliable multicast for distributed interactive simulation," in *Proceedings of ACM SIGCOMM '95*, 1995.
26. J. Nonnenmacher and E. Biersack, "Optimal multicast feedback," in *Proceedings of IEEE INFOCOM '98*, (San Francisco, CA), July 1998.

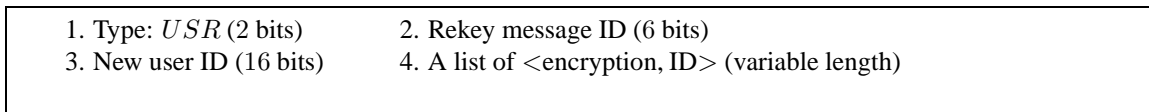
## APPENDIX A. PROTOCOL SPECIFICATION

### A.1. Formats of packets

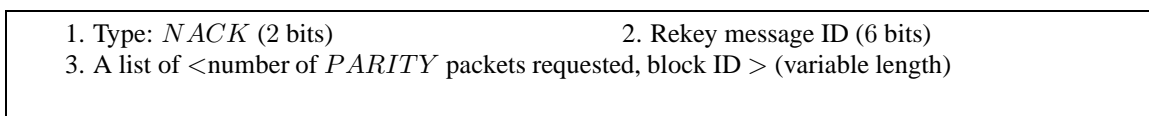
Figure 5, 23, 24 and 25 define the formats of an *ENC*, *PARITY*, *USR* and *NACK* packet respectively. In a *USR* packet, the encryption IDs are optional if we arrange the encryptions in increasing order of ID.



**Figure 23.** Format of a *PARITY* packet



**Figure 24.** Format of a *USR* packet



**Figure 25.** Format of a *NACK* packet

### A.2. The specification of protocols

The protocol for the key server is shown in Figure 26. And the protocol for a user is shown in Figure 27. In both protocols, we consider only one rekey message. In the protocols, both the server and users behave based on rounds. In fact, it is feasible for a user to send a NACK as soon as it detects a loss, and for the server to multicast *PARITY* packets as soon as it receives a NACK from any user. An arising problem is that the server has to determine whether it has already satisfied or partially satisfied an incoming NACK. This problem can be easily solved by containing in each NACK the maximum sequence number of the packets received by the user in a specific block. This idea was first proposed by D. Rubenstein, etc.<sup>21</sup>

```

1. status = MULTICAST
2. For each block: multicast  $k$  ENC packets and  $\lceil(\rho - 1)k\rceil$  PARITY packets
3.  $R \leftarrow$  empty set //  $R$  is the set of users who send NACK
4.  $A \leftarrow$  empty list //  $A$  contains NACK information
5. For each block ID  $i$ :  $amax[i] \leftarrow 0$ 
6. Start timeout
8. AcceptNACK( $m$ , a list of  $\langle a, i \rangle$ )
   //  $m$ : the ID of the user who sends the NACK
   //  $\langle a, i \rangle$ : the user requests  $a$  PARITY packets for block  $i$ 
   if (status=MULTICAST)
   then
      $R \leftarrow R + \{m\}$ 
     append the maximum  $a$  of  $\langle a, i \rangle$  list to  $A$ 
     for each entry  $\langle a, i \rangle$  in the list:  $amax[i] \leftarrow \max(amax[i], a)$ 
   else send  $USR$  packets to  $m$  // refer to  $Switch2Unicast(R)$ 
9. When timeout
   UpdateRho( $A$ )
   if (conditions for switching to unicast hold)
   then status = UNICAST;  $Switch2Unicast(R)$ 
   else if ( $R$  is not empty)
     for each block  $i$ : multicast  $amax[i]$  new PARITY packets;  $amax[i] \leftarrow 0$ 
     start timeout

```

**Figure 26.** Key server protocol for one rekey message

## APPENDIX B. MARKING ALGORITHM

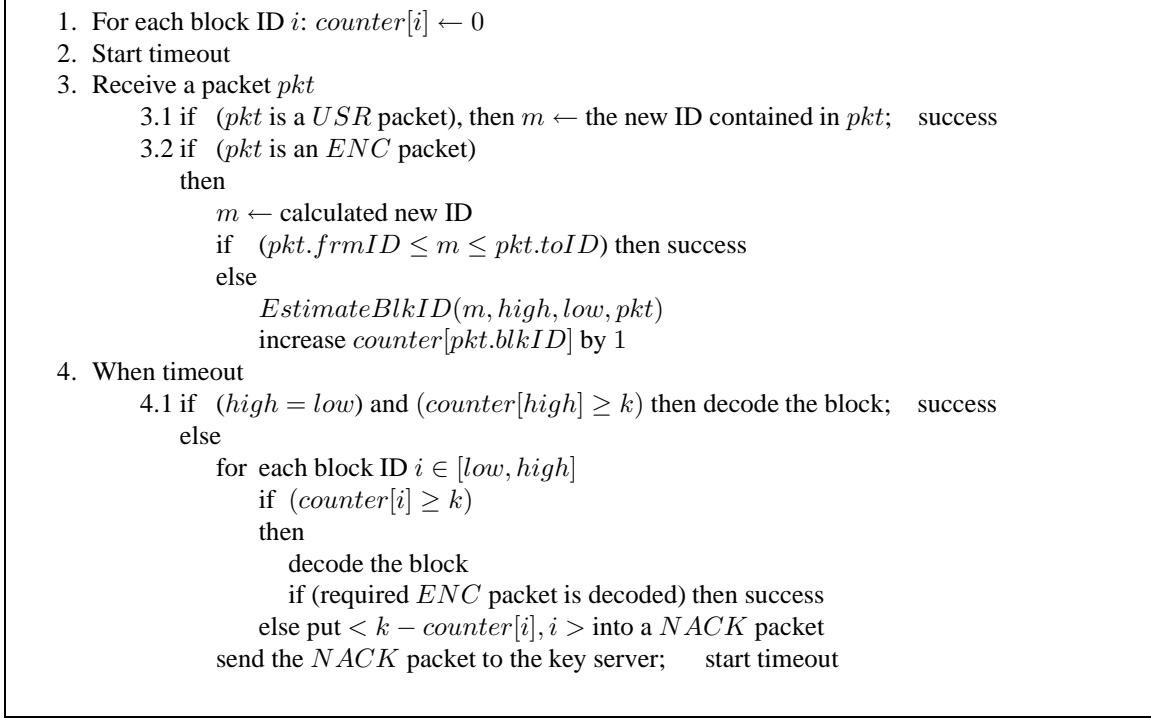
In periodic batch rekeying, the key server collects  $J$  join and  $L$  leave requests during a rekey interval. At the end of the interval, the server runs the following marking algorithm to update the key tree and construct a rekey subtree. The marking algorithm is slightly different from the one presented in our previous paper.<sup>9,10</sup> 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  $d * n_k + d$  (inclusive) in order 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.

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 newly joined u-node as Join, a u-node who has departed and then joined as Replace, 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.



**Figure 27.** User protocol for one rekey message

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 assign algorithm to assign encryptions into packets.

## APPENDIX C. PROOFS OF LEMMA AND THEOREM

### C.1. Proof of Lemma 4.1

1. The property holds for the initial key tree constructed with only join requests.
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 the joined u-nodes replace the departed u-nodes in our marking algorithm.
  - (b) For  $J > L$ , newly joined u-nodes first replace the 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 4.2

1. There exists such an integer  $x' \geq 0$  such that  $n_k < f(x') \leq d * n_k + d$ , 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, and the new ID  $m'$  is the form of  $f(x')$  for an integer  $x' > 0$ . By Lemma 4.1,  $n_k < m'$  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  $d * n_k + d$ . Therefore  $m' \leq d * n_k + d$
2. Suppose besides  $m'$ , there exists another leftmost descendant (denoted by  $m''$ ) of  $m$  which also satisfies the condition  $n_k < m'' \leq d * n_k + d$ . 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.

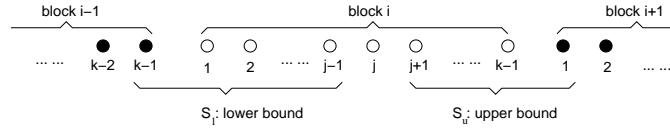
- (b) Since  $m'$  is the ancestor of  $m''$ ,  $n_k$  is the parent node of  $d * n_k + d$ , and by the assumption  $m'' \leq d * n_k + d$ , we have  $m' \leq n_k$ . This contradicts Lemma 4.1 since  $m'$  is a u-node.

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.

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  $j^{\text{th}}$  packet in block  $i$ . Let  $\langle i, j \rangle$  denote the  $\langle$ block ID, sequence number within a block $\rangle$  pair. Whenever a user receives an *ENC* packet, it can refine its estimation of the block ID  $i$ . For example, if  $m > \text{toID}$  of a received packet, then  $i \geq$  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  $S_l = \{\langle i-1, k-1 \rangle, \langle i, 0 \rangle, \dots, \langle i, j-1 \rangle\}$ , and receive any one *ENC* packet in  $S_u = \{\langle i, j+1 \rangle, \dots, \langle i, k-1 \rangle, \langle i+1, 0 \rangle\}$ , then it can determine the accurate value of  $i$  even when  $\langle i, j \rangle$  is lost. The detailed algorithm to estimate block ID is shown in Figure 29. Figure 28 illustrates the block ID estimation.



**Figure 28.** Illustration of block ID estimation

A user can determine the accurate value of the block ID with high probability. Only when all of the *ENC* packets in set  $S_l + \{\langle i, j \rangle\}$  are lost, or when all of the packets in set  $S_u + \{\langle i, j \rangle\}$  are lost, the user cannot determine the accurate value of the 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 loss 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 requires *PARITY* packets for each block within the estimated block ID range.

Algorithm EstimateBlkID ( $m, low, high, pkt$ )

- ▷  $m$  is the user's ID who calls this algorithm.
- ▷  $low$  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  $(pkt.toID \leq m \leq pkt.frmID)$  then  $high \leftarrow pkt.blkID$ ;  $low \leftarrow pkt.blkID$
2. If  $(m > pkt.toID)$  and  $(pkt.seqNo = k - 1)$  then  $low \leftarrow \max(low, pkt.blkID + 1)$
3. If  $(m > pkt.toID)$  and  $(pkt.seqNo < k - 1)$  then  $low \leftarrow \max(low, pkt.blkID)$
4. If  $(m < pkt.frmID)$  and  $(pkt.seqNo = 0)$  then  $high \leftarrow \min(high, pkt.blkID - 1)$
5. If  $(m < pkt.frmID)$  and  $(pkt.seqNo > 0)$  then  $high \leftarrow \min(high, pkt.blkID)$
6. If  $(m > pkt.toID)$  then  

$$high \leftarrow \min(high, pkt.blkID + \lceil \frac{d*(pkt.maxKID+1) - pkt.toID - (k-1 - pkt.seqNo)}{k} \rceil)$$

**Figure 29.** Estimating required block ID

Initially, a user sets the lower bound  $low$  as 0, and upper bound  $high$  as infinity. The step 6 in Figure 29 guarantees that the final value of  $high$  will not be infinity. 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  $d * (pkt.maxKID + 1)$ . In the worst case, one *ENC* packet contains encryptions for only one user, then there are at most  $(d * (pkt.maxKID + 1) - pkt.toID)$  *ENC* packets whose  $frmID$  sub-field is larger than  $pkt.toID$ . Therefore, the maximum block ID cannot be larger than  $pkt.blkID + \lceil \frac{d*(pkt.maxKID+1) - pkt.toID - (k-1 - pkt.seqNo)}{k} \rceil$ .