The Dissertation Committee for Xincheng Zhang
certifies that this is the approved version of the following dissertation:

Protocol Design for Scalable and Reliable Group Rekeying

Committee:

Simon S. Lam, Supervisor

James C. Browne

Gustavo de Veciana

Mohamed G. Gouda

John Hasenbein

Aloysius K. Mok

Protocol Design for Scalable and Reliable Group Rekeying

by

Xincheng Zhang, B.E., M.S.

DISSERTATION

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

DOCTOR OF PHILOSOPHY

THE UNIVERSITY OF TEXAS AT AUSTIN

August 2005

To my mother and father

Acknowledgments

I am very grateful to my advisor, Professor Simon S. Lam, for his encouragement and guidance throughout the course of this research. I also wish to express my appreciation to other members of my doctoral committee, Professors James C. Browne, Gustavo de Veciana, Mohamed Gouda, John Hasenbein, and Aloysius Mok.

I had a great experience in collaborating with Dong-Young Lee, Huaiyu Liu, and Yang Richard Yang. Throughout these years, my friends including Min Sik Kim, Jia Liu, and Hanbing Liu have provided much needed support and encouragement.

This research was supported in part by the National Science Foundation Research grants CNS-0434515, ANI-0319168, and ANI-9977267, Texas Advanced Research Program 003658-0439-2001, and the James C. Browne Fellowship from the Department of Computer Sciences of the University of Texas at Austin.

Protocol Design for Scalable and Reliable Group Rekeying

Publication No. _____

Xincheng Zhang, Ph.D.
The University of Texas at Austin, 2005

Supervisor: Simon S. Lam

In secure group communications, group users share a symmetric key, called group key. The group key is used for encrypting data traffic among group users or restricting access to resources intended for group users only. A key server needs to change the group key after users join and leave (called *group rekeying*), by composing a rekey message that consists of encrypted new keys (encryptions, in short) and delivering it to all users. When group size is large, it becomes infeasible to rekey per user join or leave because of its high processing and bandwidth overheads. Instead, the key server changes the group key per rekey interval, the length of which indicates how tight the group access control is. It is desired to reduce the overhead of group rekeying as much as possible in order to allow frequent rekeying.

To address the scalability issue of the key server, Wong, Gouda, and Lam proposed the key tree approach in 1998. The same idea also appears in

RFC 2627. The scalability of rekey transport, however, was not addressed. The objective of this dissertation is to design a scalable and reliable rekey transport protocol and evaluate its performance. Rekey transport differs from data transport because rekey messages require scalable, reliable, and real-time delivery. Furthermore, each user needs only a small subset of all of the encryptions in a rekey message.

We have proposed a scalable and reliable rekey transport protocol; its efficiency benefits from the special properties of rekey transport. The protocol runs in two steps: a multicast step followed by a unicast recovery step. Proactive forward error correction (FEC) is used in multicast to reduce delivery latency and limit the number of users who need unicast recovery. The unicast recovery step provides eventual reliability; it also reduces the worst-case delivery latency as well as user bandwidth overhead. In the protocol design, various technical issues are addressed. First, a key identification scheme is proposed for each user to identify the subset of new keys that it needs. The communication cost of this scheme is only several bytes per packet. Second, we investigate how to space the sending times of packets to make proactive FEC resilient to burst loss. Lastly, an adaptive FEC scheme is proposed to make the number of users who need unicast recovery controlled around a small target value under dynamic network conditions. This scheme also makes FEC bandwidth overhead and rekey interval close to the the feasible minima.

Application-layer multicast (ALM) offers new opportunities to do naming and routing. In the dissertation, we have studied how to use ALM to

support concurrent rekey and data transport in secure group communications. Rekey traffic is bursty and requires fast delivery. It is desired to reduce rekey bandwidth overhead as much as possible since it competes for available bandwidth with data traffic. Towards this goal, we propose a multicast scheme that exploits proximity in the underlying network. We further propose a rekey message splitting scheme to significantly reduce rekey bandwidth overhead at each user access link and network link. We formulate and prove correctness properties for the multicast scheme and rekey message splitting scheme. Simulation results show that our approach can reduce rekey bandwidth overhead from several thousand encryptions to less than ten encryptions for more than 90% of users in a group of 1024 users.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Many emerging Internet applications, such as grid computing, restricted teleconferences, pay-per-view of digital media, multi-party games, virtual private networks, and distributed interactive simulations will benefit from using a secure group communications model [20]. In this model, members of a group share a symmetric key, called **group key**, which is known only to group users and a 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 desired 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).

1.1 Prior work

There have been extensive research results on the design of group key management in recent years [4, 8, 12, 17, 30, 50, 52, 55]. In particular, the key

tree approach [50, 52] uses a hierarchy of keys to facilitate group rekeying, and it reduces the server processing time complexity of group rekeying from $O(N)$ to $O(\log_d(N))$, where N is the group size and d is the key tree degree. This approach was shown to be optimal in terms of server communication cost per user join or leave [47].

A key tree is a rooted tree with the group key as root [52]. It has two types of nodes: u-nodes and k-nodes. Each **u-node** corresponds to a particular user, and it contains the individual key of the user. A user's **individual key** is known only by the user and the key server. A **k-node** contains either the group key or an auxiliary key. In the key tree approach, *each user is given its individual key as well as all of the keys contained in the k-nodes along the path from its corresponding u-node to the root node.* Consider a group with nine users. An example key tree is shown in Figure 1.1. In this example, user u_9 is given the three keys k_9, k_{789}, and k_{1-9}. Key k_9 is the individual key of u_9, key k_{789} is an auxiliary key shared by u_7, u_8, and u_9, and key k_{1-9} is the group key shared by all nine users.[1]

When a user joins or leaves the group, the key server needs to change all of the keys on the path from the user's corresponding u-node to the root node. For example, in Figure 1.1, suppose that u_9 leaves the group. To update the key tree, the key server removes the u-node containing k_9 from the key tree, changes k_{789} to k_{78}, and changes k_{1-9} to k_{1-8}. To distribute the new keys

[1]The key server knows every key in the key tree.

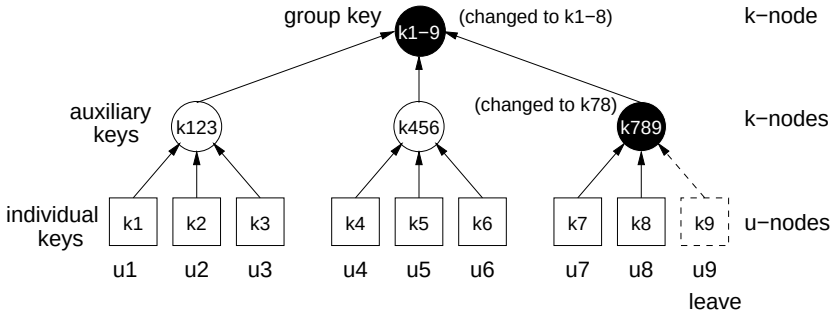


Figure 1.1: An example key tree.

to the remaining users using the group-oriented rekeying strategy [52], the key server uses the key in each child node of the updated k-node to encrypt the new key in the updated k-node. In this example, the key server generates the following encrypted new keys: $\{k_{78}\}_{k_7}$, $\{k_{78}\}_{k_8}$, $\{k_{1-8}\}_{k_{123}}$, $\{k_{1-8}\}_{k_{456}}$, and $\{k_{1-8}\}_{k_{78}}$. Here $\{k'\}_k$ denotes key k' encrypted by key k, and is referred to as an **encryption**. All encryptions are put in a single **rekey message**, which is multicasted to all remaining users. Each user, however, does not need to receive the entire rekey message since it needs only a small subset of all encryptions. For example, u_7 needs only $\{k_{1-8}\}_{k_{78}}$ and $\{k_{78}\}_{k_7}$.

When group size is large, or when users join and leave frequently, it becomes infeasible to rekey after every join or leave request because of its high processing and bandwidth overheads. To further improve the scalability of the key server, periodic **batch rekeying** was proposed [27, 45, 55]. In batch rekeying, the key server collects join and leave requests during each **rekey interval**. At the end of the rekey interval, the key server processes these requests as a batch, and generates a single rekey message. The rekey message

is sent to group users during the next rekey interval. We use T to denote the length of a rekey interval.

In batch rekeying, a new group key will not be generated until the end of a rekey interval. Suppose the new group key will be used to encrypt group communications s time units later.[2] Then a departed user can still read future group communications for up to $T + s$ time units after its departure. Therefore, T is a measure of the granularity of group access control. A small T is preferable for tight group access control.

Group rekeying requires reliable delivery of new keys to users. This is because the key server uses keys generated in one rekey interval to encrypt new keys of the next rekey interval. Reliable delivery of rekey messages has not received much attention in prior work. In Keystone [53], a basic protocol was designed and implemented that uses proactive forward error correction (FEC) to improve the reliability of multicast rekey transport. Some preliminary performance results of rekey transport were presented in [55].

1.2 Scope of the dissertation

The objective of this dissertation is to study the rekey transport problem. We will identify the special properties and requirements of rekey transport, and investigate various technical issues involved. In particular, we will

[2]We need to set $s > 0$ since it takes time to deliver a rekey message to users. In the rekeying protocol presented in Chapter 2 and 3, almost all users can receive their required new keys by the end of a single multicast round. Therefore, we can set s as the duration of a multicast round.

design protocols for scalable and reliable group rekeying, and evaluate the protocols with analytical results and simulations.

1.2.1 Properties of rekey transport

Many reliable multicast protocols for data transport have been proposed in recent years [15, 19, 21, 26, 32, 33, 35, 48]. These protocols, however, are not efficient for rekey transport because rekey transport differs from data transport. More specifically, in rekey transport, each user needs only a small subset of encryptions in a rekey message. In data transport, each user typically needs every packet. Furthermore, the amount of rekey traffic is usually much smaller than that of data traffic, and the sending rate of rekey traffic is constrained at a small fraction of total available bandwidth.

1.2.2 Requirements of rekey transport

Rekey transport has the following requirements.

- Reliable delivery requirement — This requirement arises because the key server uses keys generated in one rekey interval to encrypt new keys of the next rekey interval. Each user however does not need to receive the entire rekey message since it needs only a small subset of all encryptions.
- Soft real-time requirement — It has two meanings. First, it is required that almost all users can receive their required new keys before the new keys are used. This is because a user has to buffer incoming encrypted data packets and future (encrypted) rekey messages before receiving its

new keys to decrypt them. Second, it is desired to make rekey interval as small as possible to achieve tight group access control.

- Scalability requirement — The processing and bandwidth overheads 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 and allow a small rekey interval T.

1.2.3 Rekey transport protocol

We have proposed a scalable and reliable rekey transport protocol [57–59, 61]; its efficiency benefits from the special properties of rekey transport. The protocol runs on top of IP multicast or other multicast schemes.

Our server protocol for each rekey message consists of five phases: (i) updating the key tree to reflect user joins and leaves, and generating a sequence of encryptions; (ii) assigning the encryptions into packets (called **rekey packet**s) by running a key assignment algorithm, (iii) generating packets containing FEC redundant information (called **parity packet**s), (iv) multicast of rekey and parity packets, and (v) transition from multicast to unicast.

To achieve reliability, our protocol runs in two steps: a multicast step followed by a unicast recovery step. During the multicast step, which lasts for a single multicast round, almost all users will receive their new keys because each user needs only one specific rekey packet (guaranteed by our key assignment algorithm) and proactive FEC is also used. Subsequently, for users who cannot receive or recover their new keys in the multicast step, each of them sends a

NACK packet to the key server. The key server then sends each such user its required keys via unicast. Since each user needs only a small number of new keys, and there are few users remaining in the unicast recovery step, our protocol achieves reliability without incurring a large bandwidth overhead.

To meet the soft real-time requirement, proactive FEC is used in the multicast step to reduce delivery latency [22, 44]. In particular, only a target number of NACKs (which is usually a small value) are expected to be sent to the key server at the end of the multicast round. To achieve a small rekey interval T, our protocol seeks the smallest feasible T that allows the delivery of new keys to be accomplished at a constrained rate.

Towards a scalable design, we use the following ideas.

- 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.

- To reduce each user's processing requirement, our key assignment algorithm assigns all of the encryptions for a user into a single rekey packet. As a result, the vast majority of users can receive their specific rekey packets directly, and do not perform FEC decoding.

- To reduce key server bandwidth requirement, our protocol uses multicast to send a rekey message to users initially, and the sending rate is constrained at a small fraction of total available bandwidth.

- To reduce a user's bandwidth requirement, we use unicast for each user who cannot receive or recover its new keys during the multicast step. In this way, a small number of users in high-loss environments will not cause our protocol to perform multicast for multiple rounds to all users.

In the protocol design, we investigate the following technical issues.

- Key identification — This issue arises since each user needs only a small subset of all of the new keys after group rekeying. Therefore, we need to provide an efficient scheme for each user to identify which new keys it needs in a rekey message. The challenge is that the key server keeps changing the structure of the key tree to reflect user joins and leaves for each rekey interval. As a result, the positions of a user's required keys in the key tree keep changing. Each user, however, does not keep track of the key tree structure. To complicate the issue, each user needs to further identify which particular rekey packet contains its required new keys. In the case that its specific rekey packet is lost, the user needs to determine to which block its specific rekey packet belongs, so that it can use FEC decoding to recover the whole block. To address these issues, we propose a key, encryption, and user identification scheme, a new marking algorithm for the key server to update the key tree structure,[3]

[3]Two different marking algorithms were proposed in [27, 55], which aim to maintain a balanced key tree. However, none of them provides a simple and efficient way for a user to keep track of its up-to-date ID and identify its required keys.

and a block ID estimation algorithm. The communication cost of our scheme is only several bytes per packet.

- Proactive FEC — It was observed that FEC is very effective in improving reliability for data transport if packets experience independent loss, but it is not effective if packets are lost in a burst [32]. Our goal is to make proactive FEC effective for group rekeying even when packets experience burst loss, by exploiting the special properties of rekey transport. For this purpose, we analyze the performance of proactive FEC for group rekeying using both the Bernoulli and continuous-time Markov models. Using a constrained non-linear optimization technique, we further propose a scheme to space the sending time of packets such that packets of the same block tend to experience independent loss. Our evaluation shows that this scheme can significantly improve the probability for each user to receive or recover its new keys. In particular, with the packet spacing scheme, an increase of the number of parity packets per block (denoted by h) can exponentially reduce the number of NACKs, and we have $h = O(\log N)$ if the expected number of NACKs is constant.

- Adaptive FEC — We investigate how to control the number of NACKs (denoted by u) to be around a small target value under dynamic network conditions. A small u is desired to reduce delivery latency and limit unicast recovery overhead. However, there are tradeoffs between u, h, and T. More specifically, to make u small, the key server needs to increase h.

To deliver the increased rekey traffic at a small constrained rate, the key server may have to increase rekey interval T. However, a small h is also preferable to reduce rekey bandwidth overhead, and a small T preferable to achieve tight group access control. Therefore, it is desired for the key server to seek the smallest feasible (h, T) pair that makes u controlled around a target value. This is challenging since network conditions are dynamic, and network topology is unknown to the key server. We investigate the tradeoffs between u, h, and T using analytic models and simulations. We further propose an adaptive FEC scheme. This scheme can adaptively choose from among all feasible (h, T) pairs one with h and T values close to their feasible minima, and make u controlled around a specified value under dynamic network conditions.

1.2.4 Rekeying using application-layer multicast

Application-layer multicast (ALM) offers new opportunities to do naming and routing [6, 13]. In the dissertation, we will study how to make ALM efficient for secure group communications [62]. Our work is the first attempt on how to efficiently support both rekey and data transport using ALM.

Using ALM to support concurrent rekey and data transport in secure group communications creates new challenges. In particular, bursty rekey traffic competes for available bandwidth with data traffic, and thus considerably increases the load of bandwidth-limited links, such as the access links of users that are close to the root of the ALM tree. Congestion at such an access link

causes data losses for all of the downstream users. Therefore, it is desired to reduce rekey bandwidth overhead as much as possible.

In our approach, each user, which is an end host, in the group is assigned a unique ID that is a string of digits. All of the user IDs and their prefixes are organized into a tree structure, referred to as **ID tree**. In addition, each user maintains a **neighbor table** that supports hypercube routing [24, 25, 28, 37, 42, 63]. The neighbor tables embed multicast trees rooted at the key server and each user. Therefore, the key server or any user can send a message to every one else via multicast. We propose a multicast scheme using the neighbor tables for both rekey and data transport.

To provide fast delivery of rekey messages, we propose a distributed user ID assignment scheme that exploits proximity in the underlying network. By virtue of this scheme, each multicast tree embedded in the neighbor tables tends to be topology-aware. As a result, when a message is forwarded from its multicast source towards a user during multicast, it tends to be always forwarded geographically in the direction towards the user, rather than being forwarded over links that may go back and forth across continents.

To reduce rekey bandwidth overhead, we observe that each user needs only a small subset of encryptions in every rekey message. Therefore, it is desired to let each user receive only the encryptions needed by itself or its downstream users. The challenging issue is how each user knows who are its downstream users and which encryptions are needed by these users.

To address this issue, we propose to modify the key tree to make its structure match that of the ID tree. An identification scheme is then proposed to identify each key and encryption. With this scheme, a user can easily determine whether an encryption is needed by itself or its downstream users by checking the encryption's ID. A message splitting scheme is further proposed to let each user receive only the encryptions needed by itself or its downstream users. The splitting scheme can significantly reduce rekey bandwidth overhead at each user access link and network link.

It is possible to perform rekey message splitting on top of an existing ALM scheme such as the ones in [6, 13, 23, 38, 43, 64]. If we use an existing ALM scheme to replace the proposed multicast scheme, then in order to perform rekey message splitting each user has to keep track of who are its downstream users and which encryptions are needed by them. As a result, it incurs a large maintenance cost for the users who are close to the root of the ALM tree since each of them has $O(N)$ downstream users. In our approach, each user does not need to maintain states of its downstream users in order to perform rekey message splitting. Furthermore, the proposed splitting scheme is more effective in reducing rekey bandwidth overhead than what could be achieved with an existing ALM scheme.

We have formulated and proved correctness properties for the multicast scheme and rekey message splitting scheme. We have also conducted extensive simulations to evaluate the approach. Simulation results show that for 78% of users in a group of 226 users, the latency from a sender to each of these

users over the multicast paths is less than twice the unicast delay between the sender and such user. Furthermore, with the rekey message splitting scheme, more than 90% of users in a group of 1024 users can reduce their incoming and outgoing rekey bandwidth overhead from several thousand encryptions to less than ten encryptions.

The balance of the dissertation is organized as follows. In Chapter 2, we will present our rekey transport protocol. The protocol will be analyzed and further refined in Chapter 3. In Chapter 4, we will study how to use ALM to support group rekeying. Future work is discussed in Chapter 5. We conclude in Chapter 6. Three appendices are included. In particular, Appendix A gives proofs for the lemmas and theorems presented in the dissertation.

Chapter 2

Protocol design

In this chapter, 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 [50, 52] for secure groups and periodic batch rekeying [27, 45, 55]. At the beginning of each rekey interval, the key server sends all users a rekey message consisting of encrypted new keys (**encryption**s, 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 and fast delivery of new keys. For each rekey message, the protocol runs in two steps: a multicast step followed by a unicast recovery step. Proactive FEC multicast is used to reduce delivery latency. In particular, it attempts to deliver new keys to all users with a high probability in a single multicast round. 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 a single multicast round, 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 the 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.

Notation used in this chapter is defined in Table 2.1.

symbol	*description*
d	degree of a key tree
h	number of parity packets per FEC block
k	FEC block size (number of rekey packets per block); also denotes a key when it appears in $\{k'\}_k$
ρ	proactivity factor, defined as $(h+k)/k$
J	number of join requests in a rekey interval
L	number of leave requests in a rekey interval
N	number of users in a group
$maxKID$	largest k-node ID
u^*	target number of NACKs
α	percentage of high loss rate users

Table 2.1: Notation used in Chapter 2.

2.1 Protocol overview

In this section, we give an overview of our rekey transport protocol. The key server's behavior is described in Figure 2.1.

At the beginning of each rekey interval, the key server first runs a marking algorithm to generate encryptions. It then executes a key assignment algorithm to assign the encryptions into **rekey packet**s.[1] Our key assignment

[1] A rekey 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. run a marking algorithm to generate encryptions
2. run a key assignment algorithm to construct rekey packets
3. partition the sequence of rekey packets into blocks of k packets
4. generate h parity packets for each block
5. multicast k rekey packets and h parity packets for each block
6. **when** timeout **do**
7. collect NACK packets from users
8. adaptively adjust the proactivity factor
9. send unicast recovery packets to each user who sends a NACK

Figure 2.1: Basic protocol for key server.

algorithm ensures that all of the encryptions needed by a user are assigned into one rekey packet.

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

Then the key server multicasts all of the rekey and parity packets to all users. A user can receive or recover its required encryptions in any one of the following three cases: 1) The user receives the specific rekey packet that contains all of the encryptions for the user. 2) The user receives at least k rekey or parity packets from the block that contains its specific rekey packet, and thus the user can recover its specific rekey packet through FEC decoding. 3) The user receives a unicast recovery packet during a subsequent unicast recovery step. The unicast recovery packet contains all of the encryptions the

user needs.

After multicasting rekey and parity packets to users, the server waits for the duration of a round, which is typically larger than the largest round-trip time over all users. Meanwhile, the key server collects **NACK packet**s from users who cannot receive or recover their required encryptions in the multicast step. At the end of the multicast round, based on the NACK information received, the key server adaptively adjusts the proactivity factor to control the number of NACKs for the next rekey message. At the same time, the key server switches to unicast and sends **unicast recovery packet**s to each user who sends a NACK.[2]

An informal specification of the user protocol is shown in Figure 2.2. In our protocol, a NACK-based feedback mechanism is used because the vast majority of users can receive or recover their required encryptions in the multicast step, which consists of a single round. In particular, for each rekey message, a user checks whether it has received its specific rekey packet or can recover the block that contains this packet through FEC decoding. If not, the user sends a NACK packet to the key server. In the NACK, the user specifies the number of parity packets (denoted by a) needed to recover its required block.[3] By the property of Reed-Solomon encoding, a is equal to k minus the number of rekey

[2]To provide fast recovery, the key server can send unicast recovery packets to a user once it receives a NACK from the user. In this dissertation, however, for clear presentation, we assume that the key server starts to send unicast recovery packets at the end of the multicast round.

[3]In a NACK packet, a user may request parity packets for a range of blocks if the user cannot determine to which block its specific rekey packet belongs. See Section 2.3.1.

```
1. when timeout do
2.     if received its specific rekey packet or at least k rekey and parity
       packets from its required block, or a unicast recovery packet then
3.         recover its required block through FEC decoding if needed
4.         retrieve required encryptions
5.     else
6.         a ← number of parity packets needed for recovery
7.         unicast a NACK packet that contains the value of a to the key server
8.         start the timer
```

Figure 2.2: Basic protocol for a user.

and parity packets received in the block that contains the user's specific rekey packet.

In summary, our protocol uses four types of packets: 1) rekey packet, which contains encryptions for a set of users; 2) parity packet, which contains FEC redundant information produced by a RSE coder; 3) unicast recovery packet, which contains all of the encryptions for a particular user; 4) NACK packet, which is feedback from a user to the key server.

Note that protocols given in Figure 2.1 and 2.2 only outline the behaviors of the key server and users. More detailed specifications of these protocols are given in Appendix B.1.

2.2 Construction of rekey Packets

At the beginning of each rekey interval, the key server processes the J join and L leave requests collected in the previous rekey interval by running a marking algorithm. The marking algorithm, presented in Appendix B.3, is different from those in the previous papers [27, 55]. The marking algorithms

presented in [27, 55] aim to maintain a balanced key tree. None of them, however, provides a simple and efficient way for a user to keep track of its up-to-date ID and identify its required keys (see Section 2.2.1).

In the marking algorithm, the key server first modifies the structure of the key tree to satisfy join and leave requests. The u-nodes for departed users are removed or replaced by u-nodes for newly joined users. If $J > L$, the key server will "split" nodes after the rightmost k-node at the highest level (with the root at level 0, the lowest) to accommodate the extra joins. After modifying the key tree, the key server changes the key in each k-node (if the node exists in the updated key tree) along the path from every newly joined or departed u-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, the direct children of the updated k-nodes, and the edges connecting the updated k-nodes with their direct children. Given a rekey subtree, the key server generates encryptions in the following way: 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.

After generating a sequence of encryptions, the key server then runs a key assignment algorithm to assign the encryptions into rekey packets. To increase the probability for each user to receive its required encryptions within a single multicast round, our key assignment algorithm guarantees that all of the encryptions for a given user are assigned into a single rekey packet. For each user to identify its specific rekey packet and extract its encryptions from

the rekey packet, the key server assigns a unique ID for each key, encryption, and user; such ID information is included in rekey packets.

In the remaining part of this section, we first discuss how to assign an ID for each key, encryption, and user. Then we define the format of a rekey packet. Lastly, we present and evaluate our key assignment algorithm.

2.2.1 Key identification

After group rekeying, each user needs only a small subset of all of the new keys. Therefore, we need to provide an efficient scheme for each user to identify which (encrypted) new keys it needs in a rekey message. The challenge is that the key server keeps changing the structure of the key tree to reflect user joins and leaves for each rekey interval. As a result, the positions of a user's required keys in the key tree keep changing. Each user, however, does not keep track of the key tree structure. To address this issue, we propose a key, encryption, and user identification scheme.

To uniquely identify each key, the key server assigns an integer as the ID of each node in a key tree. More specifically, the key server first expands the key tree to make it full and balanced by adding null nodes, which we refer to as **n-node**s. 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 that do not contain keys. 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. We then define the ID of a key to be the ID of the node that contains the key. Figure 2.3 (left) illustrates how to assign node IDs in an expanded key tree with a degree of three. In this example, the group consists of seven users.

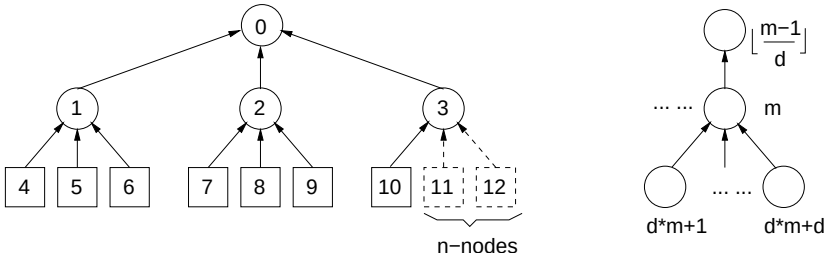


Figure 2.3: Illustration of key identification.

In the key identification scheme, the IDs of a node and its parent node in the expanded key tree have the following relationship: If a node has an ID of m, then its parent node has an ID of $\lfloor \frac{m-1}{d} \rfloor$, where d is the key tree degree. Figure 2.3 (right) illustrates this ID relationship.

To uniquely identify an encryption $\{k'\}_k$, we define the ID of the encryption to be the ID of the encrypting key k 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 the user's individual key. If a user knows its user ID, then given an encryption with its ID, by the simple ID relationship between parent

and child nodes, the user can easily determine whether the encrypted key is on the path from the user's corresponding u-node to the tree root. Recall that in the key tree approach, a user needs only the keys on the path from its corresponding u-node to the tree root. Therefore, a user can determine whether it needs the encrypted key in a given encryption (with its ID) if the user knows its ID. So the remaining issue is how a user knows its ID.

A user can get its initial ID from the key server when it joins the group. However, the key server may keep changing the ID of the user for each rekey interval. This is because when users join and leave, the marking algorithm needs to modify the structure of the key tree, and thus the IDs of some nodes will be changed. For a user to determine its up-to-date ID, 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.

To provide an efficient way for each user to derive its update-to-date ID, our marking algorithm updates the key tree structure such that two invariants are maintained, as stated in Lemma 2.2.1 and 2.2.2.

Lemma 2.2.1. *In the marking algorithm specified in Appendix B.3, if the key server changes the position of a u-node, then the new position of the u-node must be a leftmost descendant of its original position.*

Lemma 2.2.2. *After the key server updates the structure of the key tree using the marking algorithm specified in Appendix B.3, the ID of any k-node is always less than the ID of any u-node in the updated key tree.*

If the key server changes the position of a u-node, by Lemma 2.2.2, the ID of the u-node's new position must be in the range of $maxKID + 1$ and $d \cdot maxKID + d$, inclusively. Here $d \cdot maxKID + d$ is the ID of the rightmost child of the k-node $maxKID$. In this ID range, there is only one possible position that is a leftmost descendant of the u-node's old position. In this way, the user can determine the ID of the new position of its corresponding u-node, as stated in the following theorem.

Theorem 2.2.3. *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 $maxKID$ denote the largest 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 $maxKID < f(x') \leq d \cdot maxKID + d$. And m' is equal to $f(x')$.*

By Theorem 2.2.3, a user can derive its current ID by knowing its old ID and the largest ID of the current k-nodes.

2.2.2 Format of rekey packets

We now define the format of a rekey packet. (Formats of other packets are specified in Appendix B.2.) As shown in the Figure 2.4, a rekey packet has nine fields, and contains both ID information and encryptions. Each number in the parentheses of Figure 2.4 is the suggested field length, in number of bits.

1. Type: rekey packet(3)	2. Duplication flag (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)	

Figure 2.4: Format of a rekey packet.

The ID information in a rekey packet allows a user to identify the packet, extract its required encryptions, and update its user ID (if changed). In particular, Fields 1 to 5 uniquely identify a packet. A flag bit in Field 2 specifies whether this packet is a duplicate; this field will be further explained in Section 2.3. Field 6 is the largest ID of the current k-nodes. Each user can derive its current user ID based upon this field and its old user ID. Field 7 specifies that this rekey packet contains encryptions only for the users whose new IDs are in the range of $<frmID, toID>$, inclusively.

Field 8 of a rekey packet contains a list of encryption and its ID pairs. After the encryption payload, a rekey packet may be padded by zero to have fixed length because FEC encoding requires fixed length packets. Padding by zero does not cause any ambiguity since no encryption has an ID of zero.

2.2.3 User-oriented Key Assignment (UKA) algorithm

Given the format of a rekey packet, we next discuss how to assign encryptions into rekey packets. This is performed by our key assignment algorithm, which is referred to as the User-oriented Key Assignment (UKA) algorithm. The algorithm guarantees that all of the encryptions for a user are

assigned into a single rekey packet.

Figure 2.5 illustrates a particular run of the UKA algorithm in which seven rekey packets are generated. In the algorithm, the key server first puts all 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 whose IDs are in this prefix will fill up a rekey packet. Repeatedly, the key server generates a sequence of rekey packets whose $<frmID, toID>$ intervals do not overlap. In particular, the algorithm guarantees that $toID$ of a previous rekey packet is less than the $frmID$ of the next packet. This property is useful for block ID estimation to be performed by a user (see Section 2.3.1).

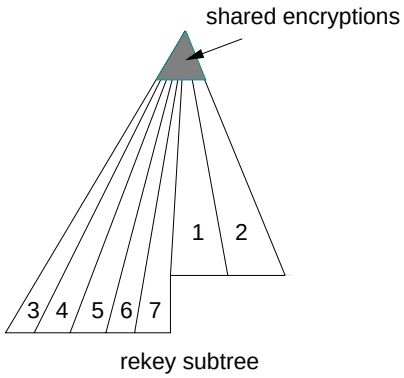


Figure 2.5: Illustration of a particular run of the UKA algorithm.

2.2.4 Performance of the UKA algorithm

The UKA algorithm assigns all of the encryptions for a user into a single rekey packet, and thus a majority of users can receive their specific rekey packets in the multicast step. This helps reducing the number of NACKs sent

to the key server.

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 rekey packets, their shared encryptions need to be duplicated in these two rekey packets. Therefore, we expect that the UKA algorithm would increase the bandwidth overhead at the key server.

We evaluate the performance of the UKA algorithm 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. In 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 a rekey packet as 1028 bytes. In all of our simulations in this chapter, 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 plotted in Figure 2.6. For a fixed L, we observe that the average number of rekey packets increases linearly with J. For a fixed J, we observe that as L increases, the number of rekey packets first increases (because more leaves imply more keys to be changed), and then decreases (because now some keys can be pruned from the rekey subtree).

Next we investigate the size of a rekey message as a function of N, as shown in Figure 2.7. We observe that the average number of rekey packets in

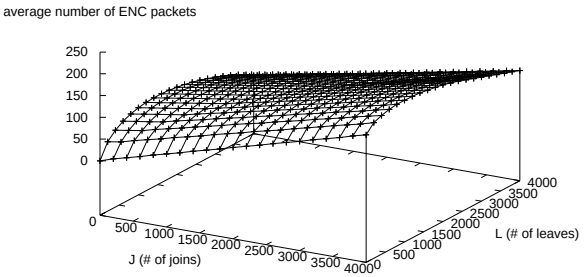


Figure 2.6: Average number of rekey packets as a function of J and L for $N = 4096$.

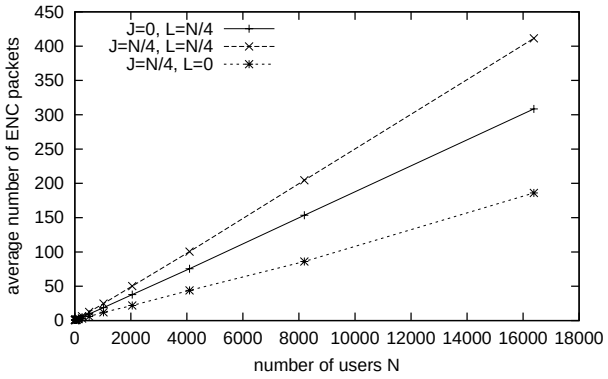


Figure 2.7: Average number of rekey packets as a function of N.

a rekey message increases linearly with N for three combinations of J and L values.

With the UKA algorithm, some encryptions are duplicated in rekey packets. We define **duplication overhead** as the ratio of duplicated encryptions in all rekey packets to the total number of encryptions in a rekey subtree. Figure 2.8 plots 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 average duplication overhead decreases from about 0.1 to 0.05 as we increase J. Next consider the case of a fixed J. We observe that the average duplication overhead first increases and then decreases as we increase L.

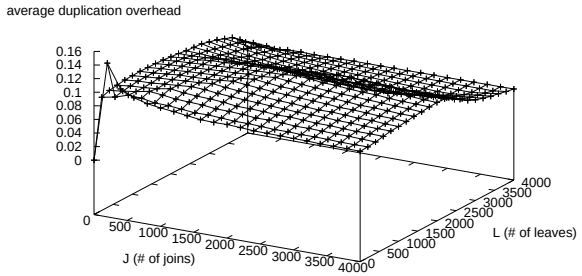


Figure 2.8: Average duplication overhead as a function of J and L for $N = 4096$.

Lastly, we plot in Figure 2.9 the average duplication overhead as a function of N. 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 $\log_d(N)$. For $J = N/4$ and $L = 0$, the rekey subtree is very sparse, and thus the curve of the average duplication overhead fluctuates around the curve of $J = L = N/4$.

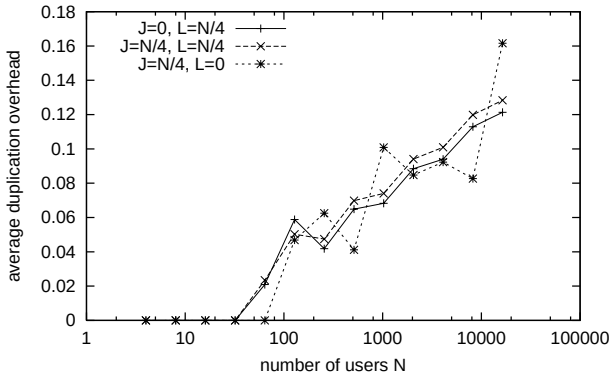


Figure 2.9: Average duplication overhead as a function of N.

2.3 Block partitioning

After running the UKA algorithm to construct rekey packets, the key server next generates parity packets for the rekey packets using a Reed-Solomon Erasure (RSE) coder.

Although grouping all rekey packets into a single RSE block may reduce server bandwidth overhead, a large block size can significantly increase encoding and decoding time [9, 32, 40]. For example, using the RSE coder of Rizzo [40], 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 rekey 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 rekey packets with a packet size of 1028 bytes. Given such a large number of rekey packets, it is necessary to partition them into multiple blocks in order to reduce the key server's encoding time.

Consider the rekey packets sequenced in order of generation by the 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 rekey packets until there are k packets to fill the last block. (The key server may choose rekey packets from other blocks to duplicate, but all duplicates are used to fill the last block.) We use a flag bit in each rekey packet to specify whether the packet is a duplicate, as shown in Figure 2.4. A duplicate rekey packet has the same contents in all fields as the original packet except for the <block ID, sequence number> and duplication flag bit fields. A new <block ID, sequence number> pair is assigned to each duplicate rekey packet because Reed-Solomon encoding needs to uniquely identify every packet, duplicate or not.

2.3.1 Block ID estimation

One issue that arises from partitioning rekey packets into blocks is that if a user lost its specific rekey packet, the user needs to determine the block to which its rekey packet belongs. Then the user will try to recover this block through FEC decoding. We present an algorithm in Appendix B.4 for users to estimate the ID of the block to which its specific rekey packet belongs. With

this algorithm, the probability that a user cannot determine the precise value of the ID of its required block 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 required block ID. It will then request parity packets for every block within this range when it sends a NACK packet. When the key server receives the NACK, it considers only the number of parity packets requested for the user's required block when it adjusts the proactivity factor.

2.3.2 Packets sent in interleaving pattern

After forming the blocks of rekey packets, the key server generates parity packets, and multicasts all rekey 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 are separated by a larger time interval, and thus are less likely to experience the same burst loss on a link. We will investigate this issue in detail in Chapter 3.

2.3.3 Choosing block size

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

The first metric is **server bandwidth overhead**, which is defined to be the ratio of v' to v, where v is the number of rekey packets in a rekey message, and v' is the total number of packets that the key server multicasts to enable recovery of specific rekey packets by all users.

To evaluate server bandwidth overhead, we use a **multi-round multicast protocol**. In this protocol, no unicast recovery is involved. For each rekey message, the key server performs multicast for multiple rounds until all users receive or can recover their specific rekey packets. The behavior of the key server and each user in the first round is the same as what is described in Figure 2.1 and 2.2.

The second and remaining rounds are performed as follows. Suppose that the specific rekey packet of user w belongs to block i, $i \geq 0$. Let $a_{i,w}$ denote the number of parity packets requested by user w for block i through its NACK packet sent in the previous round. Let $amax[i]$ be the largest number of parity packets requested in the previous round for block i by all of the users whose specific rekey packets belong to block i, that is, $amax[i] = \max\{a_{i,w}|$ for all user w whose specific rekey packet belongs to block $i\}$. Then at the beginning of this round, for each block i, the key server generates $amax[i]$ new parity packets for this block, and multicasts these packets to all users. At the user side, if a user still cannot recover its specific rekey packet, it constructs a new NACK packet in the same way as in the first round, and sends it to the key server.

The second metric is **overall FEC encoding time**, defined to be 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 the FEC decoding time at the user side, the impact is small because in our protocol, the vast majority of users can receive their specific rekey packets and thus do not perform any FEC decoding.

We use simulations to evaluate the impact of block size. To support a large group size, we developed our own simulator for a model proposed and used by J. Nonnenmacher, et al. [33]. In this model, the key server connects to a backbone network via a source link, and each user connects to the backbone network via a receiver link. The backbone network is assumed to be loss-free. The source link has a fixed loss rate of p_s. A fraction α of all group 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, say p, we use a two-state continuous-time Markov chain [32] to simulate burst loss. More specifically, the average duration of a burst loss is $\frac{100}{p}$ milliseconds, and the average duration of loss-free time between consecutive loss bursts is $\frac{100}{1-p}$ milliseconds.[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/second, and the rekey interval is 60 seconds, and the length of a rekey packet is 1028 bytes. The same simulation topology and parameter values will also be used in the experiments described in the following sections of this chapter unless otherwise stated.

[4]This network topology and loss model are simplistic compared to the Internet. They are however used for simulating a large group size (up to 16384 users). Please see Chapter 3 for simulation results from the use of *ns* and GT-ITM for a smaller group size.

The impact of block size on server bandwidth overhead is shown in Figure 2.10. In the simulations, the key server sets $\rho = 1$ (that is, $h = 0$) when it sends out each rekey message.[5] Observe that the average server bandwidth overhead is not sensitive to the block size k for $k \geq 5$.

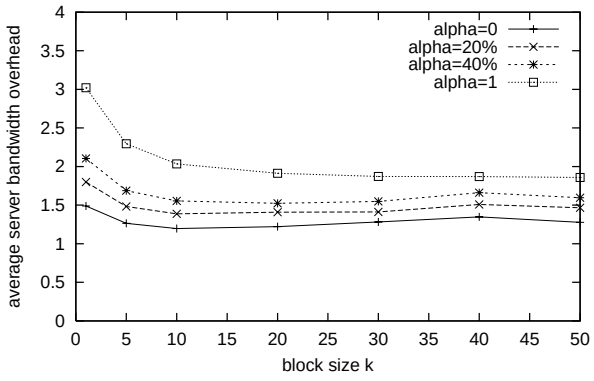


Figure 2.10: Average server bandwidth overhead as a function of block size.

We next consider the impact of block size k on overall FEC encoding time. If we use Rizzo's RSE coder [40], 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. And the encoding time for one parity packet is approximately a linear function of block size k. Figure 2.11 plots the relative overall encoding time (assuming k time units to generate one parity packet when block size is k) as a function of block size.

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

[5]We also investigated the case that the key server adaptively adjusts ρ for consecutive rekey messages. Results are similar. See technical report [60] for details.

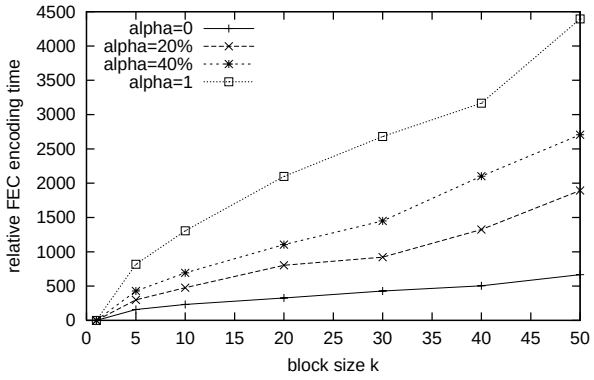


Figure 2.11: Relative overall FEC encoding time as a function of block size.

server bandwidth overhead. For simulations in the following sections of this chapter, we choose $k = 10$ as the default value unless otherwise specified.

2.4 Adaptive FEC multicast

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

Proactive FEC has been widely used to improve reliability and reduce delivery latency [7, 18, 22, 29, 32, 44, 56]. 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, users may have to depend on unicast recovery 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 cannot receive enough packets to recover its required block, and the key server would be overwhelmed by NACKs.

The appropriate value of the proactivity factor depends on network topology and network conditions including loss state of network links, number of users in a session, and number of sessions using proactive FEC. Such information is not completely known 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 let the key server adaptively adjust the 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.

2.4.1 Impact of proactivity factor

Before designing an algorithm to adjust the proactivity factor ρ, we evaluate the impact of ρ on the number of NACKs, the delivery latency at users, and server bandwidth overhead. We use the multi-round multicast protocol described in Section 2.3.3 for all of the simulations presented in this subsection. In these simulations, the value of the proactivity factor is specified as an experiment parameter. Therefore, the key server does not adaptively adjust ρ for consecutive rekey messages.

We first evaluate the impact of ρ on the number of NACKs. Figure 2.12 plots the average number of NACKs for the first round as a function of ρ. Note

that y-axis is in log scale. We observe that the average number of NACKs decreases exponentially as we increase ρ. (A similar observation was made in a previous study for data transport [44].)

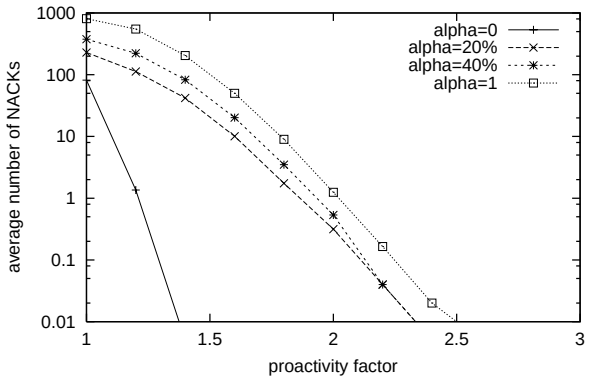


Figure 2.12: Average number of NACKs in the first round as a function of ρ.

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

Table 2.2: Percentage of users on average who need a given number of rounds to receive or recover their required encryptions.

We next evaluate the impact of ρ on delivery latency. Table 2.2 shows the percentage of users on average who need a given number of rounds to receive or recover their required 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 ρ on average server bandwidth overhead, as shown in Figure 2.13. For ρ close to 1, the key server sends a small number of proactive parity packets during the first round, but it needs to send much more reactive parity packets in subsequent rounds to allow users to recover their required encryptions. As a result, a small increase of ρ has little impact on the average server bandwidth overhead. When ρ becomes large, the rekey and parity packets sent during the first round dominates the server bandwidth overhead, and the average server bandwidth overhead increases linearly with ρ.

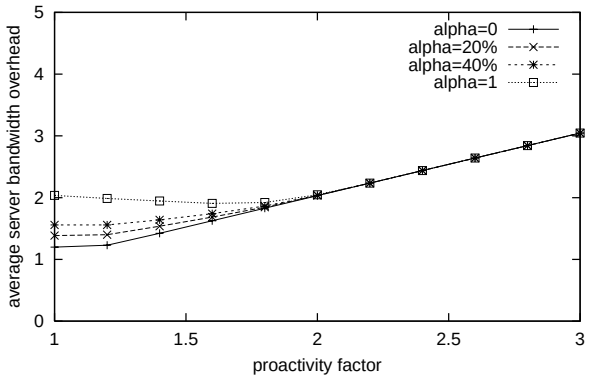


Figure 2.13: Average server bandwidth overhead as a function of ρ.

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

2.4.2 Adjustment of proactivity factor

We present in Figure 2.14 an algorithm for the key server to adaptively adjust the proactivity factor ρ. In the algorithm, according to the NACK information received for the current rekey message, the key server chooses a new value of ρ 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 multicast step for each rekey message. Note that the multicast step consists of a single multicast round (see Figure 2.1).

AdjustRho (u^*, A)
▷ u^*: target number of NACKs
▷ A: each item in A is the number of parity packets requested by
a user for its required block
1. **if** $(size(A) > u^*)$ **then**
2. $a^{u^*+1} \leftarrow (u^*+1)$th (starting from 1) largest item in A
3. $h \leftarrow h + a^{u^*+1}$
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. $\rho \leftarrow (h+k)/k$

Figure 2.14: Algorithm to adaptively adjust the proactivity factor.

The input to the algorithm *AdjustRho* is the target number of NACKs u^* and a list A. Each item in A is the number of parity packets requested by a user through its NACK packet. If a user requests parity packets for a range of blocks, the key server records into A only the number of parity packets requested for the block that contains the user's specific rekey packet.

The algorithm works as follows. For each rekey message, at the end of the multicast step, the key server compares the number of NACKs it has

received (that is, $sizeof(A)$) and the number of NACKs it targets (that is, u^*). The comparison results in two cases.

In the first case, the key server receives more NACKs than it targets. In this case, the server selects the $(u^* + 1)$th (starting from 1) largest item (denoted by a^{u^*+1}) from A, and increases ρ such that a^{u^*+1} additional proactive parity packets will be generated for each block of the next rekey message. For example, suppose ten users, u_1, u_2, ..., u_{10}, have sent NACKs for the current rekey message, and user u_i requests a^i parity packets for the block to which its specific rekey packet belongs. For illustration, we assume $a^1 \geq a^2 \geq ... \geq a^{10}$, and the target number of NACKs is 2, that is, $u^* = 2$. Then according to our algorithm, for the next rekey message, the key server will send a^3 additional parity packets so that users u_3, u_4, ..., u_{10} have a higher probability to recover their specific rekey packets in the multicast step. This is because according to the current rekey message, if users u_3, u_4, ..., u_{10} were to receive a^3 more parity packets, they could have recovered their specific rekey packets.

In the second case, the key server receives less NACKs than it targets. Although receiving less NACKs is better in terms of reducing delivery latency, the small number of NACKs received may indicate that the current proactivity factor is too high, thus causing high bandwidth overhead. Therefore, in this case, our algorithm reduces ρ by one parity packet with probability equal to $\frac{u^* - 2 \cdot size(A)}{u^*}$.

2.4.3 Performance evaluation

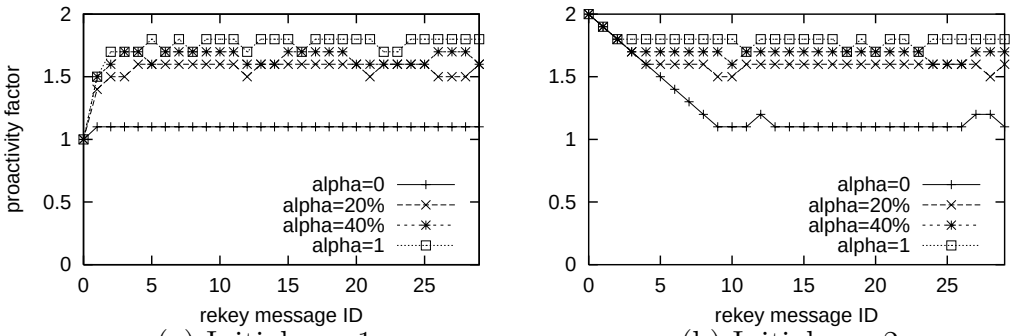
We use simulations to evaluate the algorithm *AdjustRho*. We will first investigate whether our protocol can effectively control the number of NACKs, and then evaluate extra server bandwidth overhead that it may incur. We choose 20 to be the default value of u^* unless otherwise specified.

2.4.3.1 Controlling the number of NACKs

For all of the simulations presented in this subsection, we use the rekey protocol specified in Figures 2.1 and 2.2. Recall that in this protocol, the multicast step consists of a single round, and the key server adaptively adjusts the proactivity factor at the end of the multicast round.

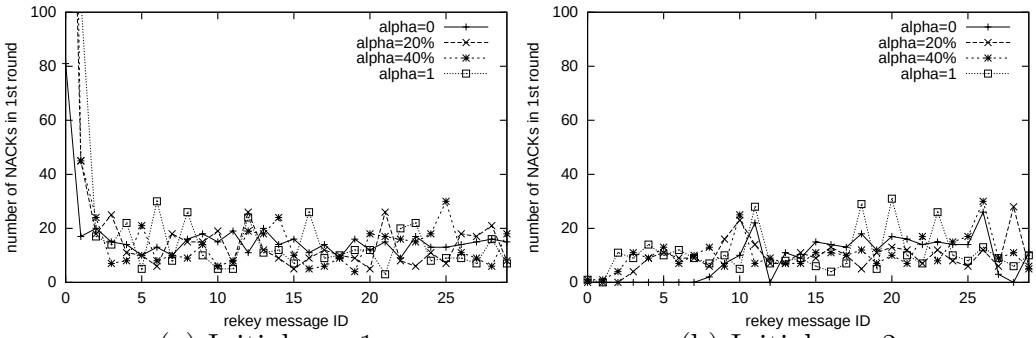
Before evaluating whether the algorithm *AdjustRho* can control the number of NACKs, we first investigate the stability of the algorithm.

Figure 2.15 shows how ρ is adaptively adjusted when the key server sends a sequence of rekey messages. Each curve in the figure (and all remaining figures in this subsection) is a trace obtained from one typical simulation run. For the case of initial $\rho = 1$, we observe that it takes only two or three rekey messages for ρ to settle down to stable values, as shown in Figure 2.15 (a). In the case of initial $\rho = 2$, we observe that ρ keeps decreasing until it reaches stable values, as shown in Figure 2.15 (b). Comparing Figures 2.15 (a) and (b), we note that the stable values of these two figures match each other very well.



(a) Initial $\rho = 1$ (b) Initial $\rho = 2$

Figure 2.15: Traces of proactivity factor.



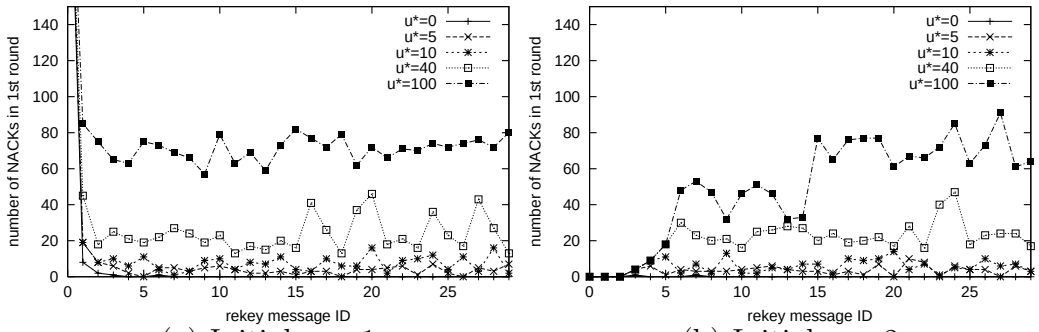
(a) Initial $\rho = 1$ (b) Initial $\rho = 2$

Figure 2.16: Traces of the number of NACKs for various loss conditions.

Figure 2.16 plots the traces of the number of NACKs returned in the multicast step. In Figures 2.16 (a) where the initial ρ value is 1, the number of NACKs received stabilizes very quickly, and the stable values are generally less than 1.5 times of u^*. Figures 2.16 (b) shows the case for initial $\rho = 2$. We observe that the stable values of these two figures match very well.

We then evaluate whether the algorithm *AdjustRho* can control the

number of NACKs for various values of u^*. As shown in Figure 2.17, the number of NACKs received at the key server fluctuates around each 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.



(a) Initial $\rho = 1$ (b) Initial $\rho = 2$

Figure 2.17: Traces of the number of NACKs for various target number of NACKs.

2.4.3.2 Overhead of adaptive FEC

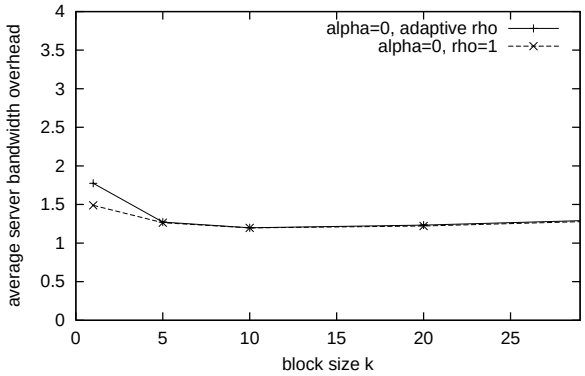
From the previous subsection, we know that the algorithm *AdjustRho* can effectively control the number of NACKs and thus reduce delivery latency. However, compared with an approach that does not send any proactive parity packets at all during the first round and only generates reactive parity packets during the subsequent rounds, the adaptive FEC scheme may incur extra server bandwidth overhead. We investigate this issue in this subsection.

We first evaluate the extra server bandwidth overhead caused by adap-

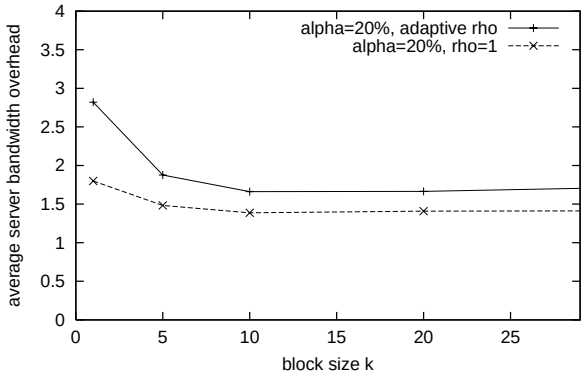
tive FEC under various loss conditions. Figure 2.18 compares the average server bandwidth overhead for the adaptive FEC scheme with the case that all parity packets are generated reactively (we call it $\rho = 1$ case). In the simulations, we use the multi-round multicast protocol described in Section 2.3.3. To measure average server bandwidth overhead for the adaptive FEC scheme, we let the key server adaptively adjust the proactivity factor for consecutive rekey messages that it multicasts. In particular, we set initial $\rho = 1$, and let the key server send out 10 rekey message. We then compute average server bandwidth overhead based on the next 100 or more rekey messages. In contrast, for $\rho = 1$ case, the key server sets $\rho = 1$ for each rekey message it sends out.

From Figure 2.18, we observe that our adaptive scheme causes little extra average server bandwidth overhead in a homogeneous low loss environment, that is, $\alpha = 0$ (recall that α is the percentage of high loss rate users). For $\alpha = 1$, the adaptive 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 ($\rho = 1$ case) 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 average server bandwidth overhead generated by the adaptive scheme is less than 0.3 for $k \geq 10$.

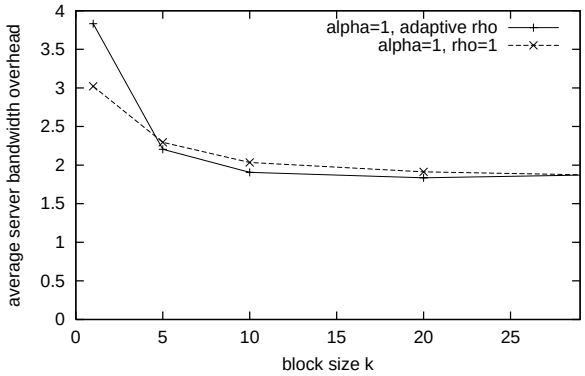
We next compare the average server bandwidth overhead of the two schemes for various group sizes. From Figure 2.19, we observe that the extra



(a) No high loss rate users ($\alpha = 0$).

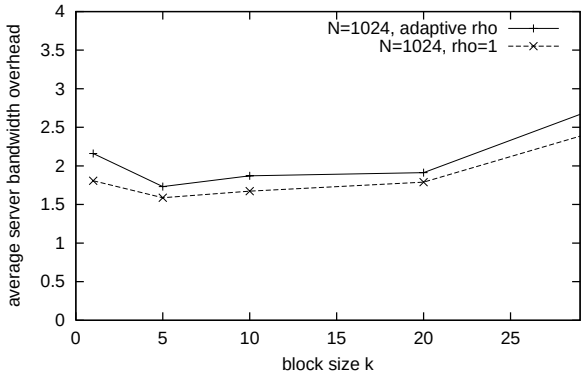


(b) 20% high loss rate users ($\alpha = 20\%$).

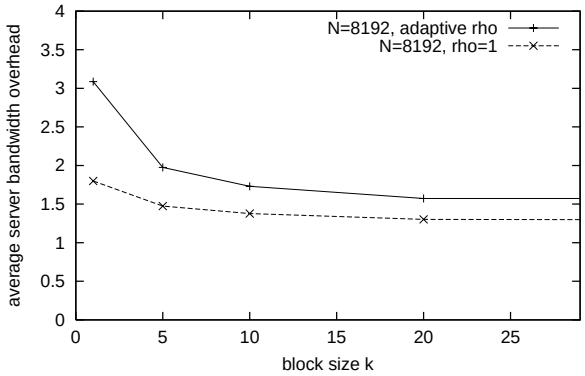


(c) All users have high loss rate ($\alpha = 100\%$).

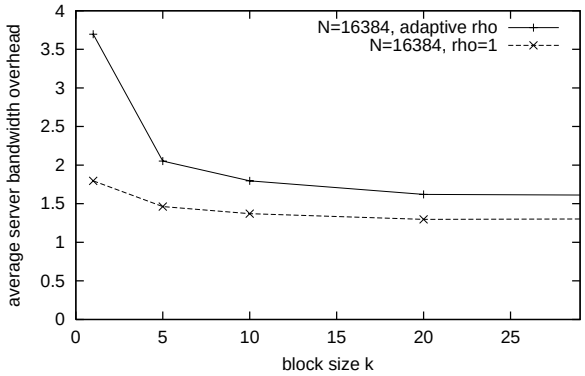
Figure 2.18: Average server bandwidth overhead for the adaptive FEC scheme and for $\rho = 1$ case under various loss conditions.



(a) $N = 1024$



(b) $N = 8192$



(c) $N = 16384$

Figure 2.19: Average server bandwidth overhead for the adaptive FEC scheme and for $\rho = 1$ case when the group size N varies.

average server bandwidth overhead incurred by the adaptive scheme increases with N. But the extra average bandwidth overhead incurred is still less than 0.42 even for $N = 16384$ when $k \geq 10$.

2.5 Speedup with unicast

Rekey transport has a soft real-time requirement, that is, it is required that almost all users can receive their required new keys before the new keys are used. This requirement arises because a user has to buffer incoming encrypted data packets and future (encrypted) rekey messages before receiving its new keys to decrypt them. To meet this requirement, we proposed to use adaptive FEC in the multicast step to reduce the number of users who send NACKs. To further reduce delivery latency, the key server will switch to unicast after a single multicast round. Unicast can reduce delivery latency compared to multicast because the duration of a multicast round is typically larger than the largest round-trip time over all users.

One issue of early unicast is its possible high unicast recovery bandwidth overhead at the key server. In our protocol, however, unicast recovery will not cause large bandwidth overhead at the key server for the following two reasons. First, the size of a unicast recovery packet is much smaller than that of a rekey or parity packet. In our protocol, a unicast recovery packet contains only the encryptions for a specific user. The number of encryptions needed by a user is less than or equal to the height of the key tree. On the other hand, the size of a rekey or parity packet is typically more than one kilobyte long.

Second, our adaptive FEC scheme ensures that only a few users need unicast if the target number of NACKs is small enough.

2.6 Summary

In this chapter, we present in detail our rekey transport protocol as well as its performance. Our server protocol for each rekey message consists of five phases: (i) updating the key tree to reflect user joins and leaves, and generating a sequence of encryptions; (ii) assigning the encryptions into rekey packets by running a key assignment algorithm, (iii) generating parity packets, (iv) multicast of rekey and parity packets, and (v) transition from multicast to unicast.

In the first and second phases, the key server runs a marking algorithm to generate encryptions and then uses a key assignment algorithm to construct rekey packets. The major problem in these two phases is to allow a user to identify its required new keys after the key tree has been modified. To solve the problem, first we assign a unique integer ID to each key, encryption, and user. Second, we propose a new marking algorithm that facilitates key identification. Third, our key assignment algorithm ensures that all of the encryptions for a user are assigned into a single rekey packet. By including a small amount of ID information in rekey packets, each user can easily identify its specific rekey packet and extract the encryptions it needs.

In the third phase, the key server uses a RSE coder to generate parity packets for rekey 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 determine to which block its specific rekey packet belongs.

In the fourth phase, the key server multicasts both rekey and parity packets to all users. This proactive FEC multicast can effectively reduce delivery latency of users; however, a large proactivity factor may increase 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 simulations show that the number of NACKs can be effectively controlled around a target number, thus achieving low delivery latency, while the extra server bandwidth overhead incurred is small.

In the fifth phase, the key server switches to unicast to provide eventual reliability and also reduce the worst-case delivery latency. In our protocol, each unicast recovery packet contains only the encryptions for a particular user. Furthermore, the number of users who need unicast recovery is effectively controlled around a target value (which is usually small). Therefore, unicast recovery does not incur high bandwidth overhead at the server.

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. Lastly, an adaptive algorithm to adjust the proactivity factor is proposed and evaluated. The algorithm is found to be effective in controlling the number of NACKs and reducing delivery latency. The algorithm will be further analyzed and refined in Chapter 3.

Chapter 3

Protocol analysis and refinement

In the previous chapter, we presented the design and specification of a scalable and reliable group rekeying protocol. In this protocol, a key server can deliver a rekey message to a large number of users efficiently using multicast. For reliable delivery, we proposed the use of forward error correction (FEC) in an initial multicast, followed by the use of unicast recovery for users that cannot receive or recover their new keys from the multicast. In this chapter, we investigate how to limit unicast recovery to a small fraction r of the user population. By specifying a very small r, almost all users in the group will receive their new keys within a single multicast round.

We present analytic models for deriving r as a function of the amount of FEC redundant information (denoted by h) and the rekey interval duration (denoted by T) for both Bernoulli and continuous-time Markov Chain loss models. From our analyses, we conclude that r decreases roughly at an exponential rate as h increases. We then present an adaptive FEC scheme to adaptively adjust (h, T) to achieve a specified r. In particular, the scheme chooses from among all feasible (h, T) pairs one with h and T values close to their feasible minima. The adaptive FEC scheme also adapts to an increase

in network traffic. Simulation results using *ns*-2 show that with network congestion the adaptive FEC scheme can still achieve a specified r by adjusting values of h and T.

Notation used in this chapter is defined in Table 3.1.

symbol	*description*
BW_m	amount of multicast traffic (in bytes per rekey message)
BW_u	amount of unicast recovery traffic (in bytes per rekey message)
h	number of parity packets per FEC block
h_l	a lower bound of h
k	FEC block size (number of rekey packets per block)
n	number of users in a group
r	fraction of users that require unicast recovery
r^*	target value of r
T	length of rekey interval (in seconds)
u^*	target number of NACKs
(h^*, T^*)	smallest feasible (h, T) pair for a specified r^*
(h', T')	a (h, T) pair that is close to (h^*, T^*)

Table 3.1: Notation used in Chapter 3.

3.1 Overview of group rekeying protocol

In this section, we give an overview of our group rekeying protocol, which is a refinement of the protocol presented in Chapter 2. In particular, the key server will adaptively adjust the value of T in addition to h at the end of the multicast round.

The key server protocol for one rekey message is as follows.

- At the beginning of each rekey interval, the key server processes all of the join and leave requests collected in the previous rekey interval by running

a marking algorithm. The marking algorithm generates a sequence of encrypted new key, called **encryption**. In the key tree approach, a user needs a particular encryption only if the encryption contains a key that is on the path from the user's corresponding u-node to the root node.

- The key server assigns the encryptions into **rekey packet**s. Our key assignment algorithm (presented in Section 2.2.3) guarantees that all of the encryptions for a given user are assigned into a single rekey packet.

- The key server partitions the sequence of rekey packets into multiple blocks. Each block contains k rekey packets.[1] We call k the **block size**. The key server then generates h **parity packet**s for each block using a Reed-Solomon Erasure (RSE) coder [40].

- The key server multicasts k rekey packets and h parity packets for each block during this rekey interval.

- The key server collects NACK packets from users. For each user who sends a NACK, the key server sends **unicast recovery packet**s each containing all of the encryptions needed by the user.

- After collecting NACKs, the key server adjusts the values of h and T for the next rekey message according to the NACK information received.

[1]The key server may need to duplicate some packets for the last block so that there are exactly k packets for each block.

At the user side, following a timeout, a user checks whether it has received or can recover its required encryptions. A user can recover its required encryptions in any one of the following three cases: 1) The user receives the specific rekey packet that contains the user's required encryptions. 2) The user receives at least k rekey and parity packets from the block that contains its specific rekey packet, and thus the user can recover its specific rekey packet through FEC decoding. 3) The user receives a unicast recovery packet during subsequent unicast recovery step. The unicast recovery packet contains all of the encryptions needed by the user.

If the user cannot receive or recover its required encryptions, it will report a NACK packet to the key server. In the **NACK packet**, the user specifies the number of parity packets it needs to recover the block to which its specific rekey packet belongs.[2] By the property of Reed-Solomon encoding, this value is equal to k minus the number of rekey and parity packets received in the block to which the user's specific rekey packet belongs.

3.2 Analyses

In the rekey transport protocol, the key server uses an FEC scheme to send $k+h$ packets for each block within rekey interval T, such that most users can receive or recover their required encryptions in the multicast step. We call

[2] A user will request parity packets for a range of blocks in its NACK if it cannot determine which block contains its specific rekey packet, as discussed in Appendix B.4. When the key server receives the NACK, it considers only the number of parity packets requested for the user's required block when it executes the adaptive FEC scheme.

the (expected) fraction of users who cannot receive or recover their required encryptions during multicast as **residual error rate**, denoted by r. For these users, the key server will use unicast to deliver their required encryptions to them. These users, however, have to buffer incoming data packets that are encrypted by the new group key until they receive the new keys. Hence, a small r is preferable in terms of reducing the buffering overhead at the user side as well as reducing the key server's unicast recovery traffic.

To achieve a small r, we may need to increase T (the duration of rekey interval). More specifically, to make r smaller, the key server needs to increase h (the number of parity packets per block) and thus the amount of rekey traffic. To deliver the increased rekey traffic at a small constrained rate, the key server has to increase T. Otherwise, if the key server increases the sending rate of rekey traffic instead of increasing T, rekeying traffic may hurt the performance of other flows in the Internet [2, 3].

On the other hand, as a measure of the granularity of group access control, a small T is preferable. This is because all join and leave requests issued in the same rekey interval are processed in a batch. Thus a new group key will not be generated and used until the end of each rekey interval. Suppose the new group key will be used to encrypt group communications s seconds later. Then a departed user can still read future group communications for up to $T + s$ seconds after its departure. Therefore, a small T is preferable for tight group access control.

In this section, we will investigate the tradeoffs between r, T, and h. To

do this, we will first analyze r as a function of h and T, and then investigate the impact of rekey bandwidth constraint on the relationships among h, T, and r.

3.2.1 Analytic models

In this subsection, we will analyze r as a function of h and T, denoted by $r = f(h, T)$. Both the Bernoulli and Markov loss models are considered. For simplicity of analyses, we assume that users experience independent and homogeneous (same loss parameters) losses. Under this assumption, r equals the probability that a user cannot receive or recover its required encryptions during multicast. For simplicity, we still call this probability residual error rate.

3.2.1.1 Bernoulli model for independent loss

We now derive the expression of $r = f(h, T)$ for the case that T is large, that is $f(h, \infty)$. In this case, packets sent consecutively can be spaced widely enough such that they will likely experience independent losses. Temporally independent losses can be simulated by the Bernoulli loss model. In particular, letting p denote the packet loss rate seen by each user, we have

$$
\begin{aligned}
r &= f(h, \infty) && (3.1)\\
&= p \cdot \sum_{i=0}^{k-1} \left(\begin{array}{c} k+h-1 \\ i \end{array} \right) (1-p)^i p^{k+h-1-i} && (3.2)\\
&= p^{k+h} \cdot \sum_{i=0}^{k-1} \left(\begin{array}{c} k+h-1 \\ i \end{array} \right) (\frac{1}{p} - 1)^i && (3.3)
\end{aligned}
$$

where k is the block size. Intuitively, r equals the probability that the user does not receive its specific rekey packet, and it receives less than k packets from the block to which its specific rekey packet belongs.

From Equation 3.3, we observe that for a fixed k and p, $\sum_{i=0}^{k-1} \left(\begin{array}{c} k+h-1 \\ i \end{array} \right) (\frac{1}{p} - 1)^i$ is a polynomial function of h with a degree of $k-1$. Letting $\mathcal{P}^{k-1}(h)$ denote this polynomial function, we have

$$r = p^{k+h} \cdot \mathcal{P}^{k-1}(h) \tag{3.4}$$

$$= O(p^h). \tag{3.5}$$

From this expression, we observe that the effect of h on r comes from the product of two terms: p^{k+h} and $\mathcal{P}^{k-1}(h)$. When h increases, the first term p^{k+h} decreases exponentially, while the second term $\mathcal{P}^{k-1}(h)$ increases as a polynomial function of h. Since the first term changes at a faster rate than the second, we expect that increasing h will reduce r exponentially. Furthermore, from Equation 3.5, we have $h = O(\log(1/r))$.

3.2.1.2 Markov model for burst loss

In the subsection above, we consider the case that T is large, and thus derive $r = f(h, \infty)$ based on the Bernoulli loss model. If T is small, however, packets sent within interval T will likely experience temporally dependent losses. To analyze $r = f(h, T)$ for a small T, we apply a Markov loss model used in [7, 32] to investigate correlated losses between consecutive packets.

In the Markov loss model, a two-state continuous-time Markov chain

$\{X_t\} \in \{0, 1\}$ is used to model the packet losses. In particular, a packet transmitted at time t is lost if $\{X_t\} = 1$ and not lost if $\{X_t\} = 0$. The generator matrix of this Markov chain is

$$\mathcal{Q} = \begin{bmatrix} -\mu_0 & \mu_0 \\ \mu_1 & -\mu_1 \end{bmatrix}.$$

Its rate transition diagram is shown in Figure 3.1.

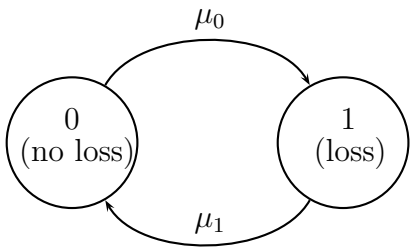


Figure 3.1: Transition diagram of the two-state Markov chain.

Let π_i, $i = 0, 1$, be the stationary distribution of this Markov chain. Let $p_{i,j}(\tau)$ denote the probability that the process is in state j at time $t + \tau$ given that it was in state i at time t. That is $p_{i,j}(\tau) = P(X_{t+\tau} = j | X_t = i)$. Then we have $\pi_0 = \mu_1/(\mu_0 + \mu_1)$, $\pi_1 = \mu_0/(\mu_0 + \mu_1)$, and

$$p_{1,1}(\tau) = (\mu_0 + \mu_1 \cdot \exp(-(\mu_0 + \mu_1)\tau))/(\mu_0 + \mu_1) \quad (3.6)$$

$$= \pi_1 + \pi_0 \cdot \exp(-(\mu_0 + \mu_1)\tau). \quad (3.7)$$

Before analyzing $r = f(h, T)$, we first need to figure out how to space packets when they are sent out within rekey interval T. It is well known that residual error rate is sensitive to the packet spacing under burst losses [32].

Therefore, we are concerned with how to space packets so as to minimize r while they are sent out within interval T.[3]

To answer this question, we observe that in our group rekeying protocol, a particular user needs packets only from the block to which its specific rekey packet belongs. Therefore, we consider the case that the key server sends only a block of $k+h$ packets within interval T to a particular user. We are concerned with how to space the $k+h$ packets so as to minimize the residual error rate for this user.

Let τ_i denote the interval between the times at which the ith and $(i+1)$th packets are sent, $i = 1, 2, ..., k+h-1$. Those packets tend to experience burst losses when T is small. We assume that the probability of more than one burst loss duration happening within a small interval T is low. Suppose that the specific rekey packet that this user requires is at the mth position, $m = 1, 2, ..., k$. Given that the loss duration starts from the jth packet, where $j = 1, 2, ..., m$ and $j+h \geq m$, we can derive the conditioned residual error rate

[3]Bolot, et al. investigated another case of this problem in a similar way [7]. The optimization problem in [7] was presented for a unicast telephony application, and the paper maximized the probability that at least one packet out of k packets is received. In our multicast based group rekeying protocol, however, each user needs to receive its specific rekey packet, or receive at least k packets out of the $k+h$ packets from the block to which its specific rekey packet belongs.

for this user as

$$r_{m,j} = \pi_1 \cdot \prod_{i=j}^{j+h-1} p_{1,1}(\tau_i) \quad (3.8)$$

$$= \pi_1 \cdot \prod_{i=j}^{j+h-1} (\pi_1 + \pi_0 \cdot \exp(-(\mu_0 + \mu_1)\tau_i)) \quad (3.9)$$

where the right side expression represents the probability that the jth packet and the following h packets are lost.

From Equation 3.9 we observe that the conditional probability $r_{m,j}$ is a decreasing function of τ_i for $j \leq i \leq j+h-1$. To minimize $r_{m,j}$, we should have $\tau_i = 0$ for $i < j$ or $i > j+h-1$ since $\sum_{i=1}^{k+h-1} \tau_i = T$. Then the desired values of $\{\tau_i \mid j \leq i \leq j+h-1\}$ should be the solution to the following constrained non-linear optimization problem:

$$\begin{array}{ll} \text{minimize} & \pi_1 \cdot \prod_{i=j}^{j+h-1}(\pi_1 + \pi_0 \cdot \exp(-(\mu_0 + \mu_1)\tau_i)) \\ \text{subject to} & \sum_{i=j}^{j+h-1} \tau_i = T. \end{array}$$

Solving it using the standard Lagrange multipliers method, we get $\tau_j = \tau_{j+1} = ... = \tau_{j+h-1}$. Therefore, to minimize $r_{m,j}$ for a particular user, the packets should be equally spaced.

Our eventual goal is to minimize the residual error rate for each user without conditioning on the start point of the loss duration. We observe that different users may need different specific rekey packets, and the loss duration may start from different packets. That is, m and j may vary from user to user.[4] To minimize the sum of residual error rates for all users, we argue that

[4]In our key assignment algorithm presented in Section 2.2.3, each rekey packet contains encryptions for roughly equal number of users.

we should have $\tau_i = \tau_j, \forall i, j = 1, 2, ..., k + h - 1$.

Thus we get our packet spacing strategy as follows. If the rekey message consists of only one block of packets, all of the packets should be equally spaced in rekey interval $[t,\ t + T]$ including both endpoints. If the key server has multiple blocks to send, the packets belonging to the same block should be equally spaced. The packets from different blocks, however, should be sent in an interleaved fashion. That is, the ith packets from each blocks should be sent together without spacing. Though these packets will likely experience burst losses, it is harmless since each particular user needs packets only from one specific block. Figure 3.2 illustrates how the key server should space the sending times of packets. In this example, the rekey message is divided into three blocks, and each block contains $k + h = 4$ rekey and parity packets.

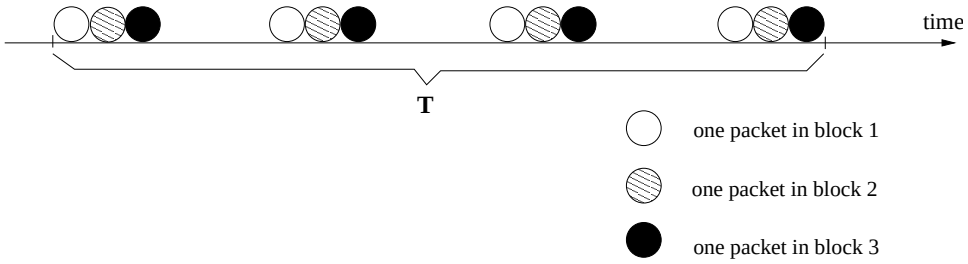


Figure 3.2: Illustration of our packet spacing scheme.

With equal spacing between packets, we can derive a lower bound of r by assuming that at most one loss duration happens during rekey interval T,

as follows:

$$r = f(h,T) \tag{3.10}$$

$$\geq \pi_1 \cdot \exp(-\mu_1 \cdot h \cdot \tau) \tag{3.11}$$

$$\approx \pi_1 \cdot \exp(-\frac{\mu_1 \cdot h \cdot T}{h+k-1}) \tag{3.12}$$

where $\exp(-\mu_1 \cdot h \cdot \tau)$ is the probability that the loss duration lasts for at least $h \cdot \tau$ time interval. Factor τ here is the spacing interval between two consecutive packets of the same block. Roughly speaking, τ equals $T/(h+k-1)$ if we ignore packet transmission time.

3.2.1.3 Illustration of $r = f(h,T)$

We now illustrate the function $r = f(h,T)$ with numerical and simulation results. We set $p = 0.06$ for the Bernoulli loss model, and $\mu_0 = 0.75$ and $\mu_1 = 11.75$ (thus $\pi_1 = 0.06$) for the two-state Markov model.

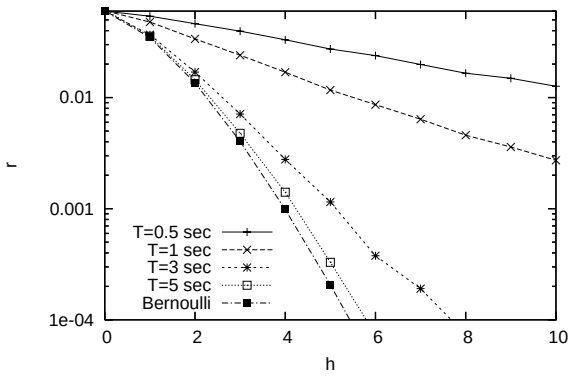


Figure 3.3: r as a function of h.

Figure 3.3 plots the value of r as a function of h. The figure contains

five curves. One is for the Bernoulli loss model, and the remaining four are based on the Markov loss model for different values of T, that is, $T = 0.5$, 1, 3, and 5 second(s). For the Bernoulli loss model, we use numerical results based on Equation 3.3. For the Markov loss model, we use simulations to compute the value of r for various h and T values. Each point in the four curves is the average value based on 100 simulation runs.

As can be seen from Figure 3.3, as h increases, r decreases roughly linearly on a logarithmic scale. Therefore, we expect that $h = O(\log(1/r))$ (see Equation 3.5). We also observe that when T is small, packets sent consecutively will likely experience burst losses, and thus the Markov model gives larger r than the Bernoulli model. When T increases, the curve of r produced by the Markov model will gradually approach that of the Bernoulli model.

From now on, given (h, T), we use the larger value produced by Equation 3.3 and 3.12 to approximate the actual r. The value we choose is still a lower bound. For the evaluation in this section, however, the conclusions we draw based on the lower bound will usually hold for the actual r. Furthermore, we expect the value we choose will be close to the actual r if T is not very small.

3.2.2 Rekey bandwidth constraint

In the previous subsections, we analyzed $r = f(h, T)$ under the Bernoulli and Markov loss models. In our derivations, we assume that h and T are independent variables. The relationship between h and T, however, should be

constrained because the available rekey bandwidth is usually limited.

Rekey bandwidth constraint requires that rekey traffic should not exceed a given sending rate at any time. This constraint arises from the fact that rekey traffic has to share available bandwidth with data traffic, while the total available bandwidth is determined by network conditions and users' receiving capacities. For example, in secure group communication applications such as pay-per-view of digital media, restricted teleconferences, and multi-party games, there typically exists a considerable amount of data traffic among group users. The data traffic competes for available bandwidth with rekey traffic. Therefore, usually only a small percentage of total available bandwidth can be allocated for group rekeying. Let $b(t)$ denote the allowed sending rate (in bytes per second) for rekey messages at time t. In the literature, there are extensive research results on how to determine the unicast or multicast sending rate under dynamic network conditions. We refer interested readers to related papers such as [16, 41, 51, 54]. Here, we assume that $b(t)$ is a given system parameter.

We claim that $b(t)$ will not sharply change with time t. It is true that the total available bandwidth shared by data and rekey traffic is a dynamic function of time. However, we can adjust the rate of data traffic to keep $b(t)$ smooth. From now on, we assume that $b(t)$ is constant for the duration of a rekey interval.

Before formulating the rekey bandwidth constraint, we introduce some notation. Let n be the number of users in the system, s_m be the length of

a multicast packet (in bytes), s_u be the unicast recovery packet length (in bytes), n_b be the number of blocks in a rekey message,[5] and w be the expected number of unicast transmissions (and retransmissions) in order to deliver a unicast recovery packet to a user.

Then at the key server side, the amount of multicast traffic (in bytes per rekey message) is

$$BW_m(h) = (k + h) \cdot n_b \cdot s_m. \tag{3.13}$$

After multicast, there are about $n \cdot r$ users who cannot receive or recover their required encryptions. The key server sends unicast recovery packets to them to provide eventual reliability. The key server's unicast recovery traffic (in bytes per rekey message) is

$$BW_u(r) = n \cdot r \cdot w \cdot s_u. \tag{3.14}$$

In summary, our **rekey bandwidth constraint** can be formulated as[6]

$$BW_m(h) + BW_u(r) \leq T \cdot b(t). \tag{3.15}$$

We now evaluate the impact of h and T on the amount of **rekey traffic**, which equals $BW_m(h) + BW_u(r)$. As a concrete example, suppose that at

[5] n_b is in fact a function of rekey interval T; however, it can be treated as a given value while the rekey bandwidth constraint is formulated. In particular, at the end of one rekey interval T_i, the key server generates a rekey message that has n_b (a function of T_i) blocks. This rekey message will be sent out within the next rekey interval T_{i+1}. As far as the rekey interval T_{i+1} is concerned, n_b is a given value.

[6] Since it takes time for the key server to receive NACKs, unicast recovery for one rekey message has to be executed during the subsequent rekey intervals. However, as long as Inequality 3.15 holds for each rekey interval, the rekey traffic will not exceed allocated bandwidth over a long term.

the beginning of each rekey interval the key tree (with degree 4) is balanced with 768 users. During each rekey interval, 192 join and 192 leave requests are processed. We further assume that the leave requests are uniformly distributed over the users. We set the length of a multicast packet as 1005 bytes (including UDP and IP header sizes). The length of a unicast recovery packet is 132 bytes. This is determined by the height of the key tree. We set block size $k = 14$ and unicast retransmission factor $w = 1.3$.

Figure 3.4 illustrates the rekey traffic (in bytes per rekey message) as a function of h for various values of T (in seconds). As can be seen, as a function of h, rekey traffic first decreases and then increases linearly when h increases. This is because when h is small, unicast recovery traffic produced by large r dominates the overall rekey traffic. When h increases, unicast recovery traffic sharply decreases and eventually diminishes. And then multicast traffic begins to dominate and increase as a linear function of h. On the other hand, as a function of T, rekey traffic is large for small T, and then it decreases and keeps unchanged as T increases. This is explained by the fact that burst losses produce large r when T is small. Large r requires large unicast recovery traffic.

We next investigate the impact of the rekey bandwidth constraint on the relationship between h and T. As a concrete example, we set $b(t) = 100$ Kbps. Because of the rekey bandwidth constraint, we expect that some (h, T) pairs will violate the constraint. Figure 3.5 shows the (h, T) pairs that satisfy the rekey bandwidth constraint. We observe that when T or h is small, the

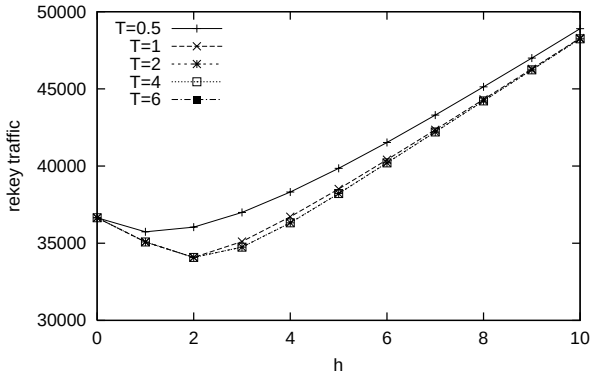


Figure 3.4: Rekey traffic (bytes per rekey message) as a function of h for various values of T (seconds).

bandwidth constraint is not satisfied because of high unicast recovery traffic, as implied by Figure 3.4. Furthermore, large h is not allowed since it produces high multicast traffic. From Figure 3.5 we can draw another important conclusion. That is, T has to be in the order of seconds to satisfy the rekey bandwidth constraint given the configuration of this example.

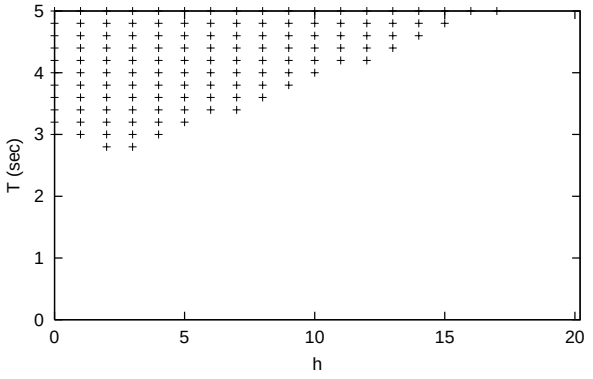


Figure 3.5: Feasible (h, T) pairs for $b(t) = 100$ Kbps.

Based on this observation, we predict that our FEC scheme will be very effective for our group rekeying protocol. To see this, we first notice that T cannot be very small because of the rekey bandwidth constraint, as implied by Figure 3.5. Second, each particular user needs packets only from one specific block. Therefore, when $k + h$ packets of the same block are equally spaced within interval T, these packets tend to experience independent losses. As a result, we expect that each user has a high probability to recover the block that it requires.

3.2.3 Tradeoffs between r, T, and h

Up to now, we have analyzed $r = f(h, T)$ under the Bernoulli and Markov loss models, and also quantified the impact of rekey bandwidth constraint on the relationship between h and T. We are now ready to investigate the tradeoffs between r, T, and h.

Recall that r is the expected fraction of users who cannot receive or recover their new keys during the multicast step, while T measures the group access control granularity. Small values of r and T are preferable. However, achieving a small r and a small T are conflicting goals.

In practice, we would like to give higher priority to r than to T. This is because r is directly related to the performance seen by each user. For this purpose, we specify a **target residual error rate** (denoted by r^*) as a system parameter. We aim to make sure that current (h, T) values satisfy $f(h, T) \leq r^*$ as well as the rekey bandwidth constraint. As a concrete example,

we set $b(t) = 100$ Kbps and $r^* = 5/n$, where $n = 768$. Figure 3.6 plots feasible (h, T) pairs that satisfy $f(h, T) \leq r^*$ as well as the rekey bandwidth constraint.

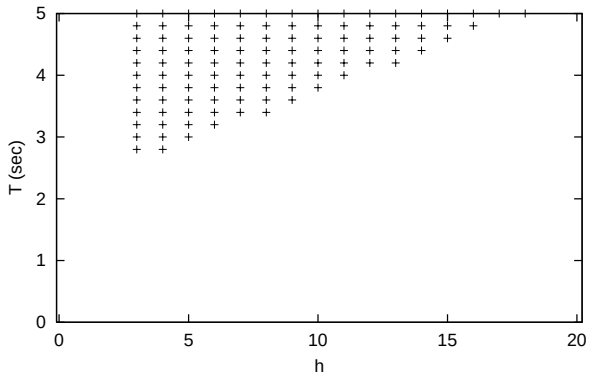


Figure 3.6: Feasible (h, T) pairs for $r^* = 5/768$ and $b(t) = 100$ Kbps.

Among all feasible (h, T) pairs for a given r^*, the one with the smallest T is preferred. Let (T^*, h^*) be such a pair, that is, $T^* = \min\{T \mid \exists h,\ s.t.\ f(h, T) \leq r^*, BW_m(h) + BW_u(f(h, T)) \leq T \cdot b(t)\}$, $h^* = \min\{h \mid f(h, T^*) \leq r^*, BW_m(h) + BW_u(f(h, T^*)) \leq T^* \cdot b(t)\}$. Figures 3.7 and 3.8 plot the values of T^* and h^* for various r^*. (In Figures 3.7 and 3.8, h' and T' are approximations of h^* and T^*. They can be computed without knowledge of the mathematical expression for $r = f(h, T)$. See Section 3.3.1.) First consider h^*. From Figure 3.7 we observe that when r^* decreases from 0.1 to 0.0001 on a logarithmic scale, h^* increases roughly at a linear rate. This confirms that our FEC scheme is very effective in reducing r. We next examine T^* as a function of r^*, as shown in Figure 3.8. Recall that when r^* decreases from a large value, unicast recovery traffic will decrease significantly. The decrease of unicast recovery traffic will balance the increase of multicast traffic. As a result, the curve of T^* keeps

flat when r^* decreases from 0.1 to 0.001. When r^* further decreases, unicast recovery traffic diminishes and multicast traffic will dominate. Consequently, T^* will increase at a similar rate as h^*. A direct conclusion from Figures 3.7 and 3.8 is that we can achieve a very small r without significantly increasing h and T.

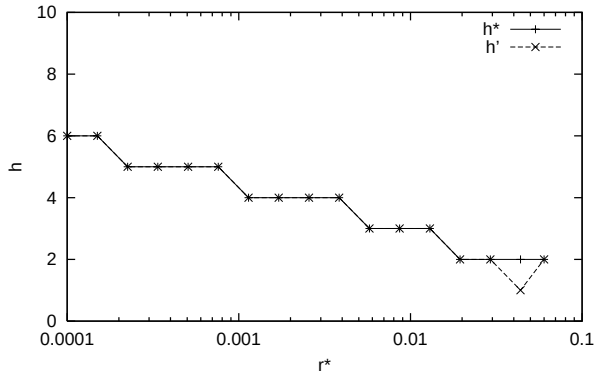


Figure 3.7: h^* and h' as functions of r^*.

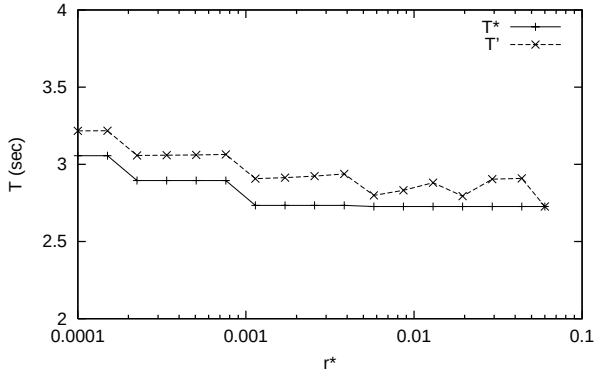


Figure 3.8: T^* and T' as functions of r^*.

3.2.4 Further discussion

In our previous analyses, we assume that loss parameters p, μ_0, and μ_1 are independent of h and T. Then from Equations 3.3 and 3.12, we conclude that r can be made as small as possible by increasing h and T.

This conclusion seems to conflict with previous research results such as [2, 3], which observed that large FEC traffic may increase residual error rate at receivers. A further investigation, however, will resolve this discrepancy. In the FEC schemes investigated in [2, 3], the sender sends $k + h$ packets for each block within fixed time interval T. Therefore, the traffic rate will increase proportionally if h is increased. When h is too large, FEC traffic will eventually overflow router buffers, and thus cause congestion. On the other hand, in our group rekeying protocol, the rate of rekey traffic is constrained by $b(t)$. In practice, the value of $b(t)$ can be updated over time and reflect up-to-date network conditions. However, since rekey traffic is usually much smaller than data traffic, we expect that rekey traffic will not hurt network performance as long as $b(t)$ is updated in a smooth manner.

3.3 Adaptive FEC Protocol

From our analyses in Section 3.2, we observe that our FEC scheme is very effective in reducing r. As a result, we can achieve a very small r without significantly increasing h and T. In this section, we will discuss how to determine (h^*, T^*) for any specified r^* under dynamic network conditions.

3.3.1 Foundation

In practice, it seems hard to find the exact T^* and h^* for a specified r^*. This is because the general form of $f(h,T)$ is usually unknown. In particular, the expression of $f(h,T)$ depends on network loss conditions as well as network topology. Therefore, it is desirable to design a method to find a near-optimal pair without knowledge of the mathematical expression for $r = f(h,T)$.

Theorem 3.3.1 shows how to find a feasible pair (h',T') that is close to (h^*,T^*).

Theorem 3.3.1. *Given that $f(h,T)$ is a non-increasing function of h and T, let (h',T') be a solution to the following set of inequalities,*

$$BW_m(h) + BW_u(r^*) = T \cdot b(t) \tag{3.16}$$

$$f(h,T) \leq r^* \tag{3.17}$$

$$f(h-1,T) > r^* \quad for \quad h > 0 \tag{3.18}$$

then we have $h' \leq h^*$ and $T' - T^* \leq \frac{BW_u(r^*) - BW_u(f(h^*,T^*))}{b(t)}$.

We expect that the difference between T' and T^* is very small in practice. By Theorem 3.3.1, we know that it is bounded by $\frac{BW_u(r^*) - BW_u(f(h^*,T^*))}{b(t)} = \frac{n \cdot w \cdot s_u \cdot (r^* - f(h^*,T^*))}{b(t)}$. This bound will be close to 0 if r^* is small enough. To see this, we first notice that the length of a unicast recovery packet (denoted by s_u) is usually very small since each unicast recovery packet contains encryptions only for one particular user. Second, we expect that $f(h^*,T^*)$ will be close to r^* when r^* is small.

Figures 3.7 and 3.8 compare the values of h' with h^* and T' with T^* for various values of r^*. The numerical results are based on the loss models described in Section 3.2. From the figures, we observe that h' is the same as h^* for a large range of r^*, and the difference between T' and T^* is very small.

3.3.2 Framework of our adaptive FEC scheme

Based on Theorem 3.3.1, we design an iterative algorithm to find (h', T') for any specified r^*. Recall that r measures the (expected) fraction of users who send NACKs. The number of users in the system changes with time. Therefore, instead of specifying a fixed r^*, we define a target number of NACKs as our system parameter. Let u^* denote the target number of NACKs.

In fact, the number of NACKs directly reflects residual error rate. Given an (h, T) pair, the number of NACKs returned to the key server is a random variable. Let U denote this random variable, and $E(U)$ be its expectation. Assuming that users have independent and homogeneous losses, we have

$$\mathbf{P}\{U = u\} = \left(\begin{array}{c} n \\ u \end{array} \right) r^u (1-r)^{n-u} \tag{3.19}$$

$$E(U) = n \cdot r. \tag{3.20}$$

Therefore, we have $u^* = n \cdot r^*$ if n is a constant. Furthermore, since we have $h = O(\log(1/r))$ (see Equation 3.5 and Figure 3.3), it follows that $h = O(\log(n/E(U))$. Thus we have $h = O(\log n)$ if the expected number of NACKs $E(U)$ is constant.

The framework of our adaptation scheme is shown in Figure 3.9. The beauty of this scheme lies in the fact that it does not require knowledge of the mathematical expression for $r = f(h, T)$.

1. **if** $E(U) > u^*$ **then**
2. $\quad h \leftarrow h + \Delta h$
3. $\quad T \leftarrow \frac{BW_m(h)+BW_u(u^*/n)}{b(t)}$
4. **if** $E(U) \leq u^*$ **then**
5. $\quad h \leftarrow \max(h-1, 0)$
6. $\quad T \leftarrow \frac{BW_m(h)+BW_u(u^*/n)}{b(t)}$

Figure 3.9: Framework of our adaptive FEC scheme.

This scheme works as follows. If $E(U) > u^*$ (and thus $r > r^*$), the key server increases h by a certain value, denoted by Δh, so that hopefully Inequality 3.17 will hold for future rekey messages. On the other hand, if $E(U) \leq u^*$ (and thus $r \leq r^*$), the key server will reduce h by 1 to make sure that current h satisfies Inequality 3.18. At any time, whenever h is updated, T will be updated according to Equation 3.16. Finally the values of h and T will be around h' and T'. Now the remaining issues are how to determine Δh and how to tell whether $E(U) > u^*$ or $E(U) \leq u^*$.

3.3.3 When to update h

We first consider when the key server should increase h in the adaptive FEC scheme. From the framework above, we know that the key server should increase h if $E(U) > u^*$. To estimate $E(U)$, it seems that the key server should collect a large number of sample values of U from consecutive rekey messages. This however will significantly slow down the system's responsiveness to sud-

den network congestion, and thus may cause poor performance in terms of r metric. On the other hand, a hasty estimation of $E(U)$ may let the key server increase h unnecessarily, and thus hurt the system performance in terms of T metric. (Recall that T will increase proportionally with h when $b(t)$ and n are constant in our framework above.) As a tradeoff, we argue that r metric may be more important than T metric. Therefore, it is preferable for our protocol to have quick responsiveness to network congestion.

To achieve quick responsiveness to network congestion, the key server will decide whether to increase h by checking the number of NACKs (denoted by u) for the last rekey message. In particular, the key server will increase h whenever $u > u_\alpha$, where u_α is defined as $\mathbf{P}\{U > u_\alpha\} \leq 1 - \alpha$ by assuming $E(U) = u^*$. Confidence level α can be specified by the owner of the key server. In this way, whenever event $u > u_\alpha$ happens, we have a confidence level of α to tell that $E(U) \neq u^*$ (and thus $E(U) < u^*$ possibly). To derive u_α, we notice that random variable U follows the binomial distribution with parameters (n, r), as shown in Equation 3.19. The binomial distribution can be approximated as the normal distribution when n is large. In particular, $\frac{U-nr}{\sqrt{nr(1-r)}}$ can be approximated as a standard normal random variable. Then for any specified α, we can derive u_α by solving $\mathbf{P}\{\frac{U-nr}{\sqrt{nr(1-r)}} \leq \frac{u_\alpha - nr}{\sqrt{nr(1-r)}}\} = \alpha$ and $n \cdot r = u^*$ For example, letting $\alpha = 99.9\%$ and $n = 768$, we have $u_\alpha = 11.9$, 19.7, and 33.6 for $u^* = 5$, 10, and 20 respectively.

We next consider when the key server should decrease h. An inappropriate decrease of h may significantly increase the number of NACKs. There-

fore, it is desired to measure $E(U)$ based on several rekey messages before the key server decides to reduce h. We use exponentially weighted average of u to approximate $E(U)$. Let $\bar{u}$ be the estimate value of $E(U)$. The key server executes $\bar{u} \leftarrow v \cdot \bar{u} + (1 - v)u$ whenever a new sample value u is available for current (h, T) pair. In our simulations, we use $v = 0.8$ and the average value should be based on at least three sample values for each updated (h, T) pair.

We further specify a lower bound (denoted by h_l) on h. That is, the value of h should be larger than or equal to h_l at any time. This prevents the key server from reducing h to a very small value due to inaccurate estimation of $E(U)$. In fact, as can be seen from Figure 3.7, it is possible for h' to reach 0 while h^* is 2. Given u^*, the value of h_l can be determined by

$$h_l = \min\{h \mid p \cdot \sum_{i=0}^{k-1} \binom{k+h-1}{i} (1-p)^i p^{k+h-1-i} \leq u^*/n\}$$

where the value of p can be chosen based on experience. Intuitively, h_l is the minimum h that makes the value of r computed by the Bernoulli loss formula in Equation 3.3 no less than $r^* = u^*/n$. Here we consider only the Bernoulli loss model since packets of the same block will likely experience independent losses, as observed in Section 3.2. The Markov loss model can also be considered if T is very small.

In our simulations, we set $p = 6\%$, and then get $h_l = 3$, 3, and 2 for $u^* = 5$, 10, and 20, respectively.

3.3.4 Determining Δh

From our previous discussions, we know that the key server should increase h by Δh whenever $u > u_\alpha$. In our approach, the key server uses the heuristic presented in Section 2.4 to determine the value of Δh. For completeness, we describe the heuristic here again.

After multicast, the key server collects NACKs from users. In its NACK, a user specifies the number of parity packets it needs to recover the block to which its specific rekey packet belongs. Let a^{u^*+1} be the (u^*+1)th (starting from 1) largest one among collected NACKs. Then the key server sets $\Delta h = a^{u^*+1}$.

3.3.5 Proposed A-FEC scheme

We are now ready to present our adaptive FEC scheme named A-FEC, as specified in Figure 3.10. The key server runs the routine after it collects NACKs from users. Initially *counter* is 0.

3.3.6 Performance evaluation

We use simulations to evaluate the performance of the A-FEC scheme. We run our simulations using network simulator *ns*-2 [34]. To simulate the Internet topology, we use Georgia Tech Internetwork Topology Models (GT-ITM) [10] to generate a Transit-Stub graph with 10 Mbps of link bandwidth. The graph contains 592 stub domains. We let the key server reside in one stub domain, and then create 591 edge networks in each of the remaining stub

A-FEC (u^*, A)
▷ u^*: target number of NACKs
▷ A: each item in A is the number of parity packets requested by a user
for its required block
1. $u \leftarrow size(A)$ ▷ number of NACKs received
2. $a^{u^*+1} \leftarrow (u^*+1)$th (starting from 1) largest item in A
3. $counter \leftarrow counter + 1$
4. **if** $counter = 1$ **then** $\bar{u} \leftarrow u$
5. **else** $\bar{u} \leftarrow v \cdot \bar{u} + (1-v)u$
6. **if** $(u > u_\alpha)$ **or** $(counter \geq 3$ **and** $\bar{u} > u^*)$ **then**
7. $h \leftarrow h + a^{u^*+1}$
8. $T \leftarrow \frac{BW_m(h)+BW_u(u^*/n)}{b(t)}$
9. $counter \leftarrow 0$
10. **if** $counter \geq 3$ **and** $\bar{u} \leq u^*$ **and** $h > h_l$ **then**
11. $h \leftarrow h - 1$
12. $T \leftarrow \frac{BW_m(h)+BW_u(u^*/n)}{b(t)}$
13. $counter \leftarrow 0$

Figure 3.10: Our adaptive FEC scheme A-FEC.

domains. Each edge network has an access link connected to the internetwork. For simplicity, we did not simulate data traffic for our group communication application. Instead, we set the bandwidth of each access link to a relatively small value. More specifically, the bandwidth of each access link is uniformly distributed between 0.1 and 1 Mbps. To simulate the background traffic, we let each of 514 edge networks have 30 outgoing and 30 incoming FTP flows, and each of the remaining 77 edge networks have 40 outgoing and 40 incoming FTP flows. For simplicity, we assume that when the key server updates h during one rekey interval, the updated h will be applied for the next rekey message. We assume that at the beginning of each rekey interval, the key tree (with degree 4) is balanced with 768 users. During each rekey interval, 192 join and 192 leave requests are processed. We set block size as $k = 14$. Our

simulations show that the average packet loss rate observed by each user is about 6%. Therefore, we set $u_\alpha = 11.9$, 19.7, and 33.6, and $h_l = 3$, 3, and 2 for $u^* = 5$, 10, and 20 respectively, as calculated earlier.

For comparison, we define two additional heuristics to compare with the A-FEC scheme:

- Heuristic 1. The key server increases h by a^{u^*+1} whenever $u > u^*$, and decrease h whenever $u \leq u^*$. No lower bound is specified for h.
- Heuristic 2. It is the same as Heuristic 1 except that h should be larger than or equal to a lower bound h_l at any time.

Figures 3.11 to 3.13 demonstrate traces of the number of NACKs for Heuristic 1, 2, and the A-FEC scheme. Each curve in the figure (and all remaining figures in this subsection) is a trace obtained from one typical simulation run. For Heuristic 1, the system starts with $h = 0$. For Heuristic 2 and the A-FEC scheme, the initial value of h is h_l. From the figures, we have the following observations:

- An increase of h by a^{u^*+1} upon $u > u_\alpha$ (or $u > u^*$) can effectively control the number of NACKs. In particular, whenever the number of NACKs is larger than u_α (or u^*), it usually takes only one rekey message for the key server to make $u \leq u_\alpha$ (or $u \leq u^*$).
- With the lower bound h_l specified, we can effectively prevent h being reduced to a very small value. As a result, specifying h_l can significantly

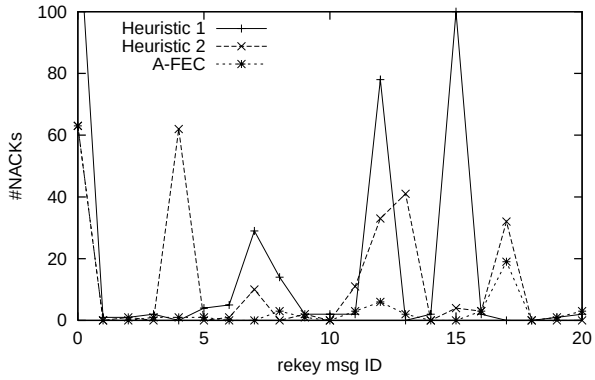


Figure 3.11: Traces of the number of NACKs for $u^* = 5$.

reduce the peak point values on the curves of the number of NACKs.

- Compared to Heuristics 1 and 2, the A-FEC scheme can further reduce the fluctuations of the number of NACKs by making the conditions to update h more strict. This is achieved by trading our protocol's responsiveness to network traffic change.

- When u^* is large, it is hard to control the fluctuations of the number of NACKs, as seen in Figure 3.13. Therefore, it is desired to specify a small u^* in practice.

We further evaluate the responsiveness of the A-FEC scheme to network traffic change. To simulate a changing network, we increase the number of background FTP flows until the total number is doubled. Each added FTP flow starts randomly during the rekey intervals of rekey messages 13 to 23. Figures 3.14 and 3.15 demonstrate the traces of h and the number of NACKs.

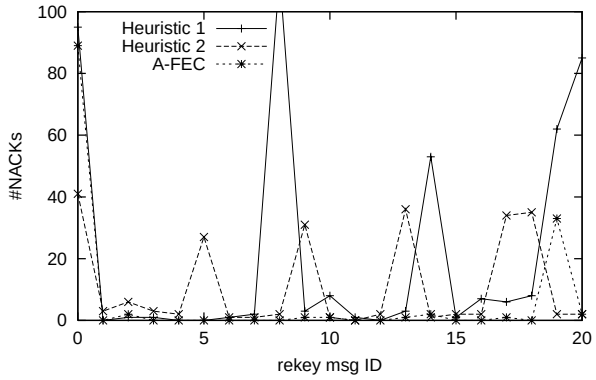


Figure 3.12: Traces of the number of NACKs for $u^* = 10$.

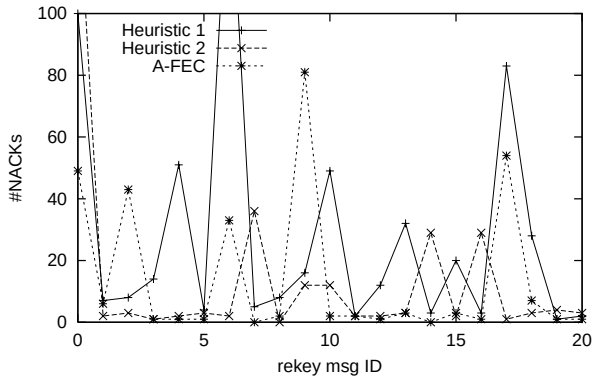


Figure 3.13: Traces of the number of NACKs for $u^* = 20$.

As can be seen, when the network becomes loaded, shared loss causes a lot of users to send NACKs. With the A-FEC scheme, the key server can quickly increase h upon network congestion, thus significantly reducing the number of NACKs for the next rekey message.

To simulate a network with decreasing traffic, we let each added FTP flow stop randomly during the rekey intervals of rekey messages 38 to 48.

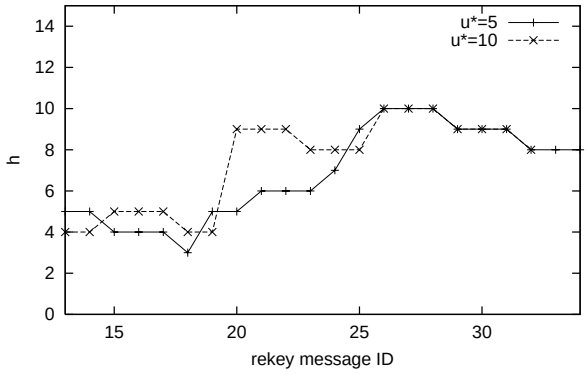


Figure 3.14: Traces of h when background traffic is doubled.

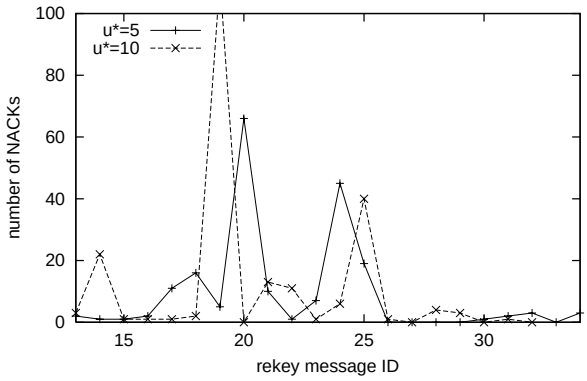


Figure 3.15: Traces of the number of NACKs when background traffic is doubled.

As seen from Figures 3.16 and 3.17, with the A-FEC scheme, the key server gradually reduces h (and T) to adapt to the improved network condition.

3.4 Related work

Following the key tree approach [50, 52], several other group key management systems have been proposed [4, 8, 12, 30, 36]. Some of these [4, 12]

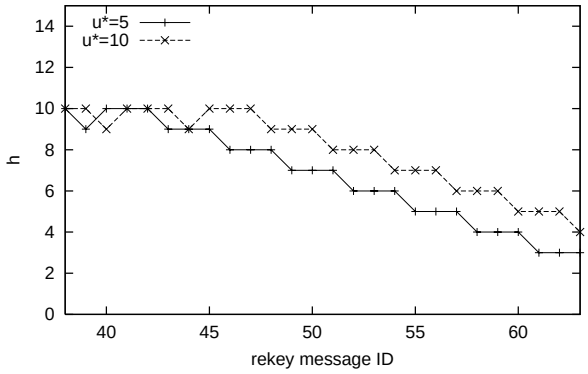


Figure 3.16: Traces of h when background traffic is reduced.

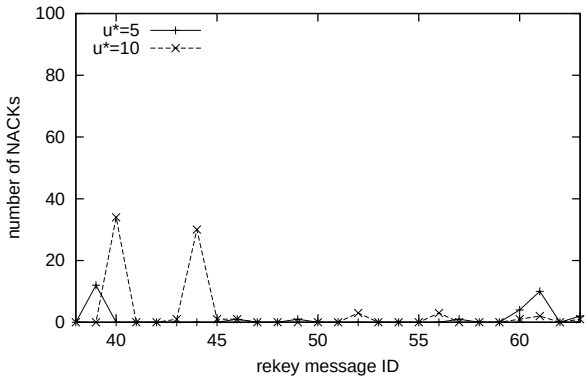


Figure 3.17: Traces of the number of NACKs when background traffic is reduced.

also require reliable delivery of rekey messages; however, no reliable group rekeying protocols have been designed for them. Other approaches, such as MARKS [8], ELK [36], and Subset-Difference [30], do not require reliable delivery of rekey messages. Each of them, however, has its limitations. In particular, MARKS assumes the lifetime of each user to be pre-determined before it joins the system. ELK introduces hint information into each data packet

to help users recover a new group key. This approach incurs per-packet bandwidth overhead and imposes significant computation overhead on users. In the Subset-Difference approach, each rekey message contains $2 \cdot e$ keys, where e is the total number of revoked users from the beginning of a session until now. The value of e may become very large as the session progresses.

Recently, the WKA-BKR protocol was proposed [46] which tries to improve server bandwidth overhead compared to our protocol. This protocol, however, does not provide real-time delivery of rekey messages as our protocol does.

Proactive FEC scheme was first proposed by Rubenstein et al. [44] for data transport. The paper observed that an increase of h can exponentially decrease the number of NACKs. However, no detailed algorithm was specified to determine and adjust h. Similar to [44], Yoon et al. also used receiver-initiated proactivity to provide fast delivery of data packets [56]. McKinley et al. proposed an adaptive proactive FEC protocol for a wireless plus wired LAN environment [29].

Bolot et al. proposed an adaptive FEC-based error control scheme for Internet telephony [7]. In their protocol, the sender calculates the amount of redundant information based on a utility function and the loss rate reported by the unicast receiver. We believe that this scheme is hard to apply to multicast because the loss rates of receivers are often unknown to the sender in multicast. At the same time, it is more challenging to determine an appropriate utility function for multicast than unicast.

3.5 Summary

In our group rekeying protocol, we study two performance metrics: r and T. Metric r measures the fraction of users who cannot receive the new group key during the initial multicast, while T is a measure of group access control granularity. Ideally, we want to achieve both small r and T. To achieve a smaller r, however, the key server has to increase h, and thus increase T to send increased rekey traffic. Therefore, there are tradeoffs between r, T, and h.

To investigate the tradeoffs between r, T, and h, we analyzed $r = f(h, T)$ under the Bernoulli and Markov loss models. We also examined the impact of rekey bandwidth constraint on the relationship between h and T. The rekey bandwidth constraint arises since we do not want rekey traffic to impact the performance of other flows in the Internet. We observed that with a rekey bandwidth constraint of $b(t) = 100$ Kbps, the value of T needs to be in the order of seconds. Then with our packet spacing scheme, packets of the same block will likely experience independent loss. As a result, an increase of h can effectively reduce r; decreasing r will not significantly increase h and T. In conclusion, we can achieve both small r and T.

We designed an adaptive FEC scheme to determine (h', T') for any specified u^*. We proved and also demonstrated that (h', T') is close to the optimal (h^*, T^*). Our scheme does not require knowledge of the mathematical expression for $r = f(h, T)$. Simulation results from ns-2 show that our protocol can achieve fairly smooth traces of the number of NACKs when group rekeying

is subjected to statistical fluctuations of a fixed set of competing flows. We also found that with the onset of network congestion our adaptive FEC protocol can still achieve the target u^* by adjusting values of h and T.

Chapter 4

Efficient rekeying using application-layer multicast

Application-layer multicast (ALM) offers new opportunities to do naming and routing. In this chapter, we propose to use application-layer multicast to support concurrent rekey and data transport. Rekey traffic is bursty and requires fast delivery. It is desired to reduce rekey bandwidth overhead as much as possible since it competes for available bandwidth with data traffic. Towards this goal, we propose a multicast scheme that exploits proximity in the underlying network. We further propose a rekey message splitting scheme to significantly reduce rekey bandwidth overhead at each user access link and network link. We formulate and prove correctness properties for the multicast scheme and rekey message splitting scheme. We have conducted extensive simulations to evaluate our approach. Our simulation results show that our approach can reduce rekey bandwidth overhead from several thousand encrypted new keys (encryptions, in short) to less than ten encryptions for more than 90% of users in a group of 1024 users. Furthermore, for 78% of users in a group of 226 users, the latency from a sender to each of these users over the multicast paths is less than twice the unicast delay between the sender and such user.

symbol	*description*
B	base of each digit in user ID
D	number of digits in user ID
F-percentile	a joining user computes F-percentile of the RTTs measured for users in its (i, j)-ID subtree
K	maximum number of neighbors in each neighbor table entry
N	number of users in a group
P	a joining user collects P users from each of its (i, j)-ID subtrees
R_i	RTT thresholds, $i = 1, 2, ..., D - 1$
$u.ID$	ID of user u
$u.ID[i]$	ith digit of $u.ID$, $0 \leq i \leq D - 1$
$u.ID[0 : i]$	first $i + 1$ digits of $u.ID$; it is a null string if $i < 0$

Table 4.1: Notation used in Chapter 4.

Notation used in this chapter is defined in Table 4.1.

4.1 System design

In this section, we present our system design. We assume a fixed group of N users in this section. User joins and leaves are discussed in Section 4.2.

4.1.1 ID tree

Each user, which is an end host, in the group is assigned a unique ID that is a string of D digits of base B, where $D > 0$ and $B > 0$. We count digits from left to right and call the leftmost digit the 0th digit. We use $D = 5$ and $B = 256$ in the simulations presented in this chapter. All user IDs and their prefixes are organized into a tree structure, referred to as ID tree, as defined below. Note that an ID is a prefix of itself, and a null string is a prefix of any ID.

Definition 4.1.1. *Given a group of users, the corresponding* ***ID tree*** *is defined as follows:*

- *At level 0, there is a single node, the tree root, whose ID is a null string, denoted by "[ ]".*
- *At level i, $1 \leq i \leq D$, each node has a unique ID that is a string of i digits. A node with ID x exists at level i if there exists a user u in the group such that x is a prefix of $u.ID$. The node with ID x at level i is a child of the node at level $i-1$ whose ID is a prefix of x.*

In an ID tree, a subtree is said to be a **level-i ID subtree** if it is rooted at a node of level i, $0 \leq i \leq D$. The ID of a subtree is defined to be the ID of the subtree root. Hereafter, we say that *a user belongs to an ID subtree* if the ID subtree has the leaf node whose ID equals the user's ID.

Definition 4.1.2. *Given a user u and an ID tree, a level-$(i+1)$ ID subtree is said to be the (i,j)-**ID subtree** of u if the parent node (at level i) of the subtree root is an ancestor of the leaf node whose ID equals $u.ID$, and the last digit of the subtree's ID is j, $0 \leq i \leq D-1$ and $0 \leq j \leq B-1$.*

By definition 4.1.2, for each user w that belongs to u's (i,j)-ID subtree, $w.ID$ must share the the first i digits with $u.ID$, and the ith digit of $w.ID$ (that is, $w.ID[i]$) is j.

Figure 4.1 illustrates the ID tree for a group of five users with the IDs "[0,0]", "[0,1]", "[2,0]", "[2,1]", and "[2,2]", respectively. In the ID tree, users

u_3, u_4, and u_5 belong to u_1's $(0,2)$-ID subtree, and u_2 belongs to u_1's $(1,1)$-ID subtree. Note that an ID tree is not a data structure maintained by the key server or any user. It is defined as a conceptual structure to guide us in protocol design.

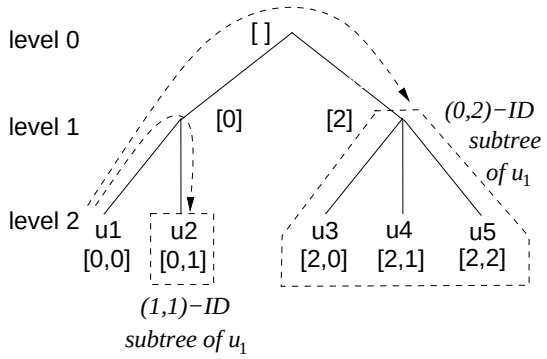


Figure 4.1: Example ID tree.

Our user ID assignment scheme exploits proximity in the underlying network. More specifically, user IDs are assigned such that the round-trip-time (RTT) between any two users belonging to the same level-i ID subtree tends to be less than or equal to a delay threshold R_i, for $i = 1, 2, ..., D-2$. As a result, all of the users belonging to the same level-i ID subtree tend to be in the same topological region with one-way delay diameter $R_i/2$. These users are partitioned into multiple child level-$(i+1)$ ID subtrees of the level-i ID subtree, such that all of the users belonging to the same level-$(i+1)$ ID subtree tend to be in the same topological sub-region with delay diameter $R_{i+1}/2$, where $R_{i+1} < R_i$. In Section 4.2.1, we discuss how a joining user determines its ID.

We further define the ID of the key server to be a null string, denoted by "[ ]". By definition, the key server belongs to the level-0 ID subtree.

4.1.2 Neighbor tables

Each user in the group maintains a neighbor table. Similar neighbor tables were used to support hypercube routing [24, 25, 28, 37, 42, 63].

A neighbor table has D rows and each row has B entries. The jth entry at the ith row is referred to as (i, j)**-entry**, $0 \leq i \leq D-1$ and $0 \leq j \leq B-1$. The (i, j)-entry of a user's neighbor table contains user records and performance measures of some other users, referred to as (i, j)-neighbors. Each (i, j)**-neighbor** of user u must be a user that belongs to the (i, j)-ID subtree of u. The first neighbor in each entry is referred to as the **primary neighbor** of that entry. Each **user record** contains the IP address, ID, and some other information of a particular neighbor. For rekey transport, the performance measure of a neighbor is the RTT between the neighbor and the owner of the table. All neighbors in the same entry are arranged in increasing order of their RTTs.

Definition 4.1.3. *Given a group of users, each with a unique ID of D digits, their neighbor tables are said to be K-**consistent**, $K \geq 1$, if for any user u in the group, each (i, j)-entry, $0 \leq i \leq D-1$ and $0 \leq j \leq B-1$, in its neighbor table satisfies the following conditions:*

(1) If $j = u.ID[i]$, then the (i, j)-entry is empty.

(2) If $j \neq u.ID[i]$, then the (i,j)-entry contains $\min\{K, m\}$ (i,j)-neighbors, where m denotes the total number of users belonging to the (i,j)-ID subtree of u.

The concept of K-consistency was proposed in [24, 25]. K-consistency implies 1-consistency. If all users in the group maintain 1-consistent neighbor tables, then a message is guaranteed to reach every user via multicast, as proved in Section 4.1.3. It is desired to let $K > 1$ for resilience [24, 25].

The key server also maintains a neighbor table, which has a single row. The row contains B entries, each referred to as $(0,j)$-entry, $j = 0, 1, ..., B-1$. Among all of the users whose IDs have the prefix "$[j]$", the key server chooses the K (or all, if the total number of such users is less than K) users who have the smallest RTTs to the key server as its $(0,j)$-neighbors.

4.1.3 Multicast scheme: T-mesh

Given a group of users with their neighbor tables, the neighbor tables embed multicast trees rooted at the key server and each user. Therefore, the key server or any user can send a message to every one else via multicast by using their neighbor tables. A multicast session consists of a sender, a set of receivers, and a message to multicast. The sender is the multicast source. In a multicast session for rekey transport, the key server is the sender, and all users in the group are receivers. In a multicast session for data transport, a particular user who has data to multicast is the sender, and all other users are

receivers. Hereafter, we use "member" to refer to the key server or a user in the group.

We propose a multicast scheme, referred to as *T-mesh*, for both rekey and data transport. In the multicast scheme, each message to multicast contains a `forward_level` field. This field specifies the forwarding level of each user, as defined below. Each user is at a unique forwarding level in a multicast session since each one receives a single copy of the multicast message, as stated in Theorem 4.1.3.

Definition 4.1.4. *In a multicast session, the sender's* ***forwarding level*** *is defined to be 0. A user u is said to be at forwarding level i if it receives a message with the* `forward_level` *field equal to i, $1 \leq i \leq D$.*

To multicast a message, the sender first sets the message's `forward_level` field to be 0, and then executes the routine FORWARD specified in Figure 4.2. When a user receives the message, it also executes this routine. We can see that each member can determine who are the next hops by looking up its neighbor table according to the `forward_level` field of the multicast message.

Figure 4.3 illustrates an example rekey multicast tree for the group of five users defined in Figure 4.1. Intuitively, a copy of the multicast message first enters each level-1 ID subtree, and then enters each level-2 ID subtree, and so on. It is not surprising to find out that the IDs of a member and its downstream users satisfy a specific relationship, as stated below.

```
FORWARD (msg)
▷ The sender should set msg.forward_level to be 0
  before calling this routine.
▷ msg: the message to multicast if the caller (who calls the routine) is
       the sender; otherwise, it is the message received by the caller.
1  level ← msg.forward_level
2  if level = D then return
3  if the caller is the key server then        ▷ level = 0 in this case
4     msg.forward_level ← level + 1
5     send a copy of msg to each (0, j)-primary neighbor, 0 ≤ j < B
6  else for i ← level to D − 1 do
8     msg.forward_level ← i + 1
9     send a copy of msg to each (i, j)-primary neighbor, 0 ≤ j < B
```

Figure 4.2: Routine that the sender or each forwarder executes to send or forward a message.

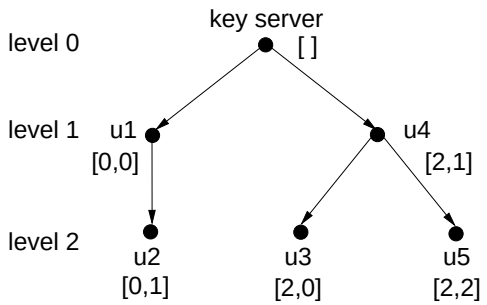


Figure 4.3: Example multicast tree for rekey transport.

Lemma 4.1.1. *In a multicast session, suppose member u is at forwarding level i, $0 \leq i \leq D$. Then the IDs of u and all of its downstream users have the common prefix $u.ID[0 : i-1]$. Furthermore, u and all of its downstream users belong to the same level-i ID subtree.*

Recall that all of the users belonging to the same ID subtree tend to be in the same topological region by virtue of our user ID assignment scheme.

Lemma 4.1.2. *In a multicast session, suppose member u is at forwarding level i, $0 \leq i \leq D$. Then for any other member w whose ID has the prefix $u.ID[0:i-1]$, w can only be a downstream user of u.*

A direct implication of Lemmas 4.1.1 and 4.1.2 is that each multicast tree embedded in the neighbor tables tends to be topology-aware. That is, in a multicast session, only a single copy of the multicast message is forwarded to each topological region; once the message with `forward_level` $= i$ enters a region (which corresponds to a level-i ID subtree), it is forwarded only to its sub-regions (each corresponds to a child level-$(i+1)$ ID subtree), and not be sent out of the region anymore. As a result, the message goes through each long-latency link that connects remote regions only once. This helps to reduce delivery latency as well as link stress. Here, stress of a physical link is defined as the number of identical copies of the message carried by a physical link during multicast.

The correctness of the multicast scheme is stated below.

Theorem 4.1.3. *In a multicast session, assume that every user in the group has 1-consistent neighbor table and no message is lost. Then following the multicast scheme specified in Figure 4.2, each member (except the sender) will receive a single copy of the multicast message.*

T-mesh also provides fast failure recovery and quick adaptation to network dynamics if $K > 1$. Once a member detects the failure of a next hop, or detects congestion on the path to a next hop by observing burst losses, it can

simply forward messages to another neighbor in the same table entry as the failed or congested neighbor. At the same time, the member needs to look for another neighbor to replace the failed or congested one.

4.1.4 Modified key tree

The key server maintains a key tree. To support efficient rekey message splitting, the key tree used in this chapter is different from the original approach proposed in [50, 52] and used in Chapter 2 and 3. The original key tree has a fixed tree degree, and the tree grows vertically when users join. Our modified key tree has a fixed height, and it grows in a horizontal direction when users join. Hereafter, unless otherwise stated, we use "key tree" to refer to the modified key tree.

A key tree is a rooted tree with the group key as root. It has two types of nodes: **u-nodes** and *k-nodes*. Each u-node corresponds to a particular user, and it contains the individual key of the user. A user's **individual key** is known only by the user and the key server. A **k-node** contains either the group key or an auxiliary key. In the key tree approach, *each user is given its individual key as well as all of the keys contained in the k-nodes along the path from its corresponding u-node to the root node* [52].

To facilitate rekey message splitting, the key server makes the structure of the key tree match exactly that of the ID tree. More specifically, for each user u, the u-node in the key tree that contains u's individual key corresponds to the leaf node in the ID tree whose ID equals $u.ID$. Figure 4.4 shows the

key tree that corresponds to the example ID tree shown in Figure 4.1. In this example, user u_5 is given the three keys on the path from its u-node to the root: k_5, k_{345}, and k_{1-5}. Key k_5 is the individual key of u_5, key k_{1-5} is the group key that is shared by all of the five users, and k_{345} is an auxiliary key shared by u_3, u_4, and u_5.[1]

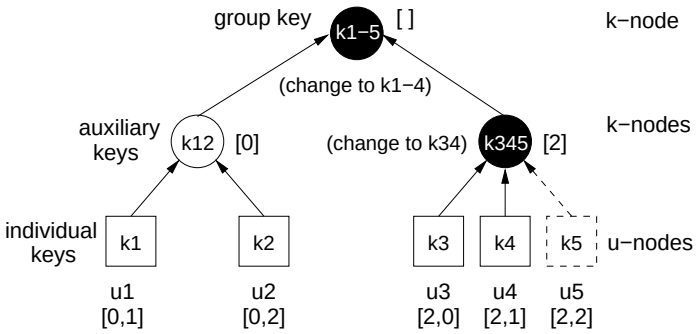


Figure 4.4: Example modified key tree.

Suppose that a single user, say u_5, leaves the group in a rekey interval. Then at the beginning of the next rekey interval, the key server needs to change the keys that u_5 knows: change k_{1-5} to k_{1-4}, and change k_{345} to k_{34}. To securely distribute the new keys to the remaining users, the key server uses the key in each child node of the updated k-node to encrypt the new key in the updated k-node, and generates the following encrypted new keys: $\{k_{1-4}\}_{k_{12}}$, $\{k_{1-4}\}_{k_{34}}$, $\{k_{34}\}_{k_3}$, and $\{k_{34}\}_{k_4}$. Here $\{k'\}_k$ denotes key k' encrypted by key k, and is referred to as an **encryption**. All of the encryptions are put in a single **rekey message**, which is multicasted to all remaining users. Each user,

[1]The key server knows every key.

however, does not need to receive the entire rekey message since it needs only a small subset of all encryptions. For example, u_1 needs only $\{k_{1-4}\}_{k_{12}}$.

In general, the key server performs the following operations in each rekey interval. For each joining user u, the key server adds into the key tree a u-node with ID $u.ID$. At each level i, $i = D-1, D-2, ..., 0$, a k-node with ID $u.ID[0:i-1]$ is added if such a k-node does not exist. For each leaving user w, the key server deletes from the key tree the u-node whose ID equals $w.ID$. At each level i, $i = D-1, D-2, ..., 0$, the k-node whose ID equals $w.ID[0:i-1]$ is deleted if the k-node does not have any descendants. At the beginning of the next rekey interval, the key server updates all of the keys (if the corresponding node exists in the updated key tree) on the path from each newly joined or departed u-node to the root, and then generate encryptions.

We propose an *identification scheme* to identify each key and encryption. We define the ID of a key in the key tree to be the ID of its corresponding node in the ID tree. The ID of an encryption is defined to be the ID of the encrypting key. The ID is attached to each encryption. With this identification scheme, a user can easily determine whether it needs a given encryption by checking the encryption's ID, as stated below.

Lemma 4.1.4. *Given an encryption, a user needs the key encrypted in the encryption if and only if the ID of the encryption is a prefix of the user's ID.*

The correctness of the lemma is due to the fact that a user needs only the keys on the path from its corresponding u-node to the root in the key tree.

4.1.5 Rekey message splitting scheme

To send new keys to users after rekeying, a straightforward approach is to multicast all encryptions to each user, and let each user extract the encryptions that it needs. The bursty rekey traffic, however, may cause congestion at bandwidth-limited links, especially at user access links. Congestion at an access link causes rekey and data message losses for all of its downstream users. Therefore, it is desired to reduce rekey bandwidth overhead as much as possible.

```
REKEY-MESSAGE-SPLIT (msg, w_{s,j}, s)
▷ msg: it is the original rekey message if the caller is the key
       server; otherwise, it is the message received by the caller.
▷ w_{s,j}: the (s, j)-primary neighbor of the caller, 0 ≤ j < B.
▷ s: it equals 0 if the caller is the key server; otherwise, we have
     msg.forward_level ≤ s < D.
1  msg' ← an empty message with forward_level = s + 1
2  for each encryption e contained in msg do
3      if e.ID is a prefix of w.ID[0 : s] or w_{s,j}.ID[0 : s] is a prefix of e.ID then
4          copy e into msg'
5  send msg' to w_{s,j} via unicast
```

Figure 4.5: Routine that the sender or each forwarder executes to compose a separate rekey message for a particular next hop.

To reduce rekey bandwidth overhead, we propose a rekey message splitting scheme. In this scheme, each member sends or forwards an encryption to its downstream users if and only if the encryption is needed by at least one downstream user. To achieve this goal, the key server composes a separate message for each $(0, j)$-primary neighbor by executing the routine REKEY-MESSAGE-SPLIT specified in Figure 4.5, $j = 0, 1, ..., B - 1$. Each user at forwarding level i, $0 \leq i \leq D - 1$, also composes a separate message for each

(s, j)-primary neighbor by executing the routine, $s = i, i+1, ..., D-1$ and $j = 1, 2, ..., B-1$. The routine in Figure 4.5 is called at lines 5 and 9 of the routine FORWARD specified in Figure 4.2. The correctness of the rekey message splitting scheme is stated below.

Theorem 4.1.5. *In a multicast session for rekey transport, suppose that member u is at forwarding level i, $0 \leq i \leq D-1$. Let w be any (s, j)-primary neighbor of u, where $s = 0$ if u is the key server, $s = i, i+1, ..., D-1$ if u is a user, and $j = 0, 1, ..., B-1$. Let set V contain w and all of the downstream users of w. Then given an encryption e, the encryption is required by at least one user in V if and only if $e.ID$ is a prefix of $w.ID[0:s]$, or $w.ID[0:s]$ is a prefix of $e.ID$.*

By Theorem 4.1.5, each member can determine whether to forward each received encryption to its downstream users by checking the encryption's ID. This is accomplished easily because a coherent identification strategy is used to identify each user, key, and encryption throughout the design of the T-mesh, the multicast scheme, and the key tree.

Corollary 4.1.6. *In a multicast session for rekey transport, assume that every user in the group has 1-consistent neighbor table and no message is lost. Following the multicast scheme and the rekey message splitting scheme specified in Figures 4.2 and 4.5, respectively, for any user u in the group and any encryption e that is generated by the key server, user u receives a single copy of e if and only if e is needed by u or by at least one downstream user of u.*

In the rekey message splitting scheme specified in Fig. 4.5, a rekey message is split in units of encryptions and then re-composed during multicast. An alternative way is to split and re-compose the rekey message at packet level, instead of encryption level. In this case, the rekey bandwidth overhead would be larger than what is presented in Section 4.3.3.

4.1.6 Discussion

In our rekey message splitting scheme, each user can easily determine whether an encryption is needed by its downstream users by checking the encryption's ID. Therefore, there is no need for each user to maintain states for its downstream users. However, if we use an existing ALM scheme such as the ones in [6, 13, 23, 38, 43, 64] to replace T-mesh, or use the original key tree [50, 52, 55, 61] to replace the modified key tree, then in order to perform rekey message splitting, each user has to keep track of who are its downstream users and which encryptions are needed by them. In the original key tree approach, the IDs of a user's required keys keep changing for each rekey interval even when no downstream users join or leave. Therefore, each user has to keep track of such changes for itself and all of its downstream users. As a result, it incurs a large maintenance cost for the users who are close to the root of the ALM tree since each of them has $O(N)$ downstream users.

Furthermore, our splitting scheme is more effective in reducing rekey bandwidth overhead than what could be achieved with the existing ALM schemes. In T-mesh, because of the exact structure match between the mod-

ified key tree and the ID tree, all of the users sharing a common encryption belong to the same level-i ID subtree, where i is the number of digits contained in the encryption's ID. As a result, only a single copy of the encryption is forwarded when the forwarding level is less than or equal to i. It is then duplicated to users who need it at subsequent forwarding levels. In contrast, if we use an existing ALM scheme to replace T-mesh, it becomes hard to make the structure of the key tree match that of the ALM tree. As a result, users sharing a common encryption have random positions in the ALM tree. In this case, the shared encryption may have to be duplicated at early forwarding levels.

The efficiency of our splitting scheme also benefits from our topology-aware user ID assignment scheme. Since all of the users sharing a common encryption belong to the same ID subtree, they tend to be in the same topological region by virtue of the user ID assignment scheme. As a result, only a single copy of the shared encryption is forwarded until it enters the region. It is then duplicated and forwarded to multiple sub-regions. In contrast, if each user randomly chooses its ID, then each user has a random position in the ID tree. For example, users from the same LAN could belong to different level-0 ID subtrees. In this case, their shared encryptions have to be duplicated once the multicast starts, and multiple copies of the shared encryptions traverse the Internet and enter the same LAN.

In short, the efficiency of our rekey message splitting scheme comes from a careful integration of the other system components, that is, the user ID

assignment scheme, the multicast scheme T-mesh, and the modified key tree. If any of these components is replaced by an existing scheme, the efficiency of the splitting scheme would be reduced. This is confirmed by our simulation results presented in Section 4.3.

4.2 Protocol description

In this section, we present the protocol for a joining user to determine its ID. We also discuss the issues related to a user's join, leave, and recovery from neighbor failures.

4.2.1 User ID assignment

To join a group, a user, say u, first contacts the key server (or a separate registrar server [53]). They mutually authenticate each other using a protocol such as SSL. If authenticated and accepted into the group, user u receives its individual key and the current group key. From now on, all of the communications between u and the key server are encrypted with the individual key, and all of the communications between u and other users in the group are encrypted with the group key.[2]

If u is the first join in the group, the key server assigns its user ID as D digits of "0". The key server then sends u a message via unicast that contains u's ID and all of the keys on the path from u's corresponding u-node to the

[2]The key server needs to send u the new group key via unicast if u cannot finish constructing its neighbor table before the end of the current rekey interval.

root in the key tree.

If u is not the first join, the key server gives u the user record of another user already in the group. Then u needs to determine its ID digit by digit, starting with the 0th digit. To determine the ith digit, $0 \leq i \leq D-2$, user u's actions consist of four steps. (We assume i is fixed in the following discussion and in Sections 4.2.1.1, 4.2.1.2, and 4.2.1.3.)

In the first step, u collects the records of users who belong to its (i, j)-ID subtree (see Definition 4.1.2), for $j = 0, 1, ..., B-1$. These users tend to be in the same topological region, and each one's ID shares the first i digits with u's ID. (User u has already determined the first i digits, $u.ID[0:i-1]$, of its ID so far.) In the second step, u measures the RTTs between itself and the users it collected. According to the measurement results, u determines the value of $u.ID[i]$ in the third step. More specifically, if u predicts that it is "close" to the users belonging to a particular ID subtree, say (i, b)-ID subtree, then u sets $u.ID[i]$ to be b, $0 \leq b \leq B-1$. As a result, u's ID shares one more digit with the users in the (i, b)-ID subtree, and u itself becomes a user belonging to this ID subtree. We thus achieve the effect that users close to each other belong to the same ID subtree. In the last step, u notifies the key server its determined ID digits. We describe each step in detail below.

4.2.1.1 Step 1: collecting user records

For u to know which users belong to its (i, j)-ID subtree, $j = 0, 1, ..., B-1$, a straightforward approach is to let the key server provide such information.

This however increases the key server's bandwidth overhead. Therefore, we let u collect the information by querying other users.

For $i = 0$, user u sends a query to the user whose record is provided to u by the key server. For $i > 0$, since u has already determined the first i digits of its ID so far, it knows at least one user that belongs to u's $(i-1,0)$-ID subtree, $(i-1,1)$-ID subtree, ..., or $(i-1,B-1)$-ID subtree. User u sends a query to such a user. The query specifies a target ID prefix as $u.ID[0:i-1]$. Upon receiving the query, the receiver looks up its neighbor table, and returns the user records of all of the neighbors whose IDs have the target ID prefix. In this way, u collects one or more users from its (i,j)-ID subtree if the subtree is not empty, for $j = 0, 1, ..., B-1$.

For each j, $j = 0, 1, ..., B-1$, to collect more users from its (i,j)-ID subtree, u keeps querying the users it collected from the ID subtree until it collects P users from the subtree, or it has queried all of the users it collected from the subtree. In each query, u specifies the target ID prefix as $u.ID[0:i-1]$ appended with digit j. We set $P = 10$ for all of the simulations in this chapter.

4.2.1.2 Step 2: measuring RTTs

In this step, u estimates whether it is close to the users it collected from its (i,j)-ID subtree, for $j = 0, 1, ..., B-1$. For this purpose, u measures the RTT between the first-hop and last-hop routers (referred to as gateway routers) on the path from u to w, for each user w it collected in the ID subtrees. Let $r(u,w)$ denote the RTT between u and w's gateway routers. Let $h(u,w)$

denote the RTT between the two end hosts u and w. In our protocol, u uses $r(u, w)$ instead of $h(u, w)$ to estimate whether it is close to w topologically. The rational is that two end hosts tend to be topologically close to each other even if their access links have long latency.[3]

User u can easily derive $r(u, w)$ if it knows $h(u, w)$, the RTT between u and its gateway router, and the RTT between w and its gateway router. For this purpose, u estimates $h(u, w)$ by using ping messages. And each user measures the RTT between itself and its gateway router using the `traceroute` utility. The value of the RTT between a user and its gateway router is stored in each copy of the user's corresponding user records so that others can know it.

4.2.1.3 Step 3: determining $u.ID[i]$

In this step, for each j, $j = 0, 1, ..., B - 1$, user u computes the F-percentile of the RTTs measured for all of the users it collected from its (i, j)-ID subtree. (Each RTT used in this step is the one between two gateway routers.) Here F is a system parameter. In order to tolerate the estimation error of RTTs, we did not use 100-percentile. Instead, 70-percentile is used in all of the simulations in this chapter. Suppose the RTTs of the users that u collected from its (i, b)-ID subtree, $0 \leq b \leq B - 1$, produces the smallest F-percentile value, denoted by $f_{i,b}$. User u then compares $f_{i,b}$ with the delay

[3]Note that the latency stored for each neighbor in a neighbor table is the RTT between two end hosts.

threshold R_{i+1}, and the comparison results in two cases.

In the first case, $f_{i,b}$ is less than or equal to R_{i+1}. User u then predicts that it is topologically close to the users belonging to its (i, b)-ID subtree, and thus assigns $u.ID[i]$ as b. User u then continues to determine the next digit $u.ID[i+1]$ of its ID if the next digit is not the last digit. That is, u increases the value of i by 1, and goes back to step 1. If the next digit is the last one, u goes to step 4 and asks the key server to assign the last digit to make sure that every user in the group has a unique ID.

In the second case, $f_{i,b}$ is larger than R_{i+1}. User u then predicts that it is not close enough to the users in any (i, j)-ID subtree, $j = 0, 1, ..., B-1$. In this case, u goes to step 4 and asks the key server to assign digits for $u.ID[i]$, $u.ID[i+1]$, ..., and $u.ID[D-1]$.

4.2.1.4 Step 4: notifying the key server

In this step, u sends the key server a message that contains its determined ID digits. Suppose u already determines the first l digits, $u.ID[0:l-1]$, of its ID, $0 \leq l \leq D-1$. The key server then assigns the lth digit to the last digit of u's ID, such that none of the other users in the group shares the first $l+1$ digits with u.[4] Consequently, in the ID tree, u becomes a user in a new

[4]In an extreme case, the key server may not be able to find a unique value for $u.ID[0:l]$ such that none of the existing users in the group shares the first $l+1$ digits with u. In this case, the key server will try to modify $u.ID[l-1]$ to make $u.ID[0:l-1]$ unique among the IDs of all existing users. If this attempt fails, the key server will try to modify $u.ID[l-2]$, $u.ID[l-3]$,..., and so forth. If all of the attempts fail, the key server will force u to join a level-1 ID subtree.

level-$(l+1)$ subtree to which none of the other users in the group belong. After that, the key server sends u a message that contains u's complete ID and all of the keys on the path from u's corresponding u-node to the root in the key tree.

To analyze the communication cost for a joining user to determine its ID, we observe that if each non-leaf node in the ID tree has the same outgoing degree, then the total number of messages exchanged while a joining user determines its ID is $O(P \cdot D \cdot N^{1/D})$ on average. The cost function is minimized as $O(P \cdot e \cdot \ln N)$ for $D = \ln N$. Here e refers to the base of the natural logarithm.

4.2.2 Join, leave, and failure recovery

After its ID is determined, u needs to build its neighbor table.[5] It also needs to contact some other users to have its user record inserted in their neighbor tables. The join protocol presented in the Silk system [24, 28] is used to accomplish this task. The join protocol is proved to construct consistent neighbor tables after an arbitrary number of joins if messages are delivered reliably and there are no user leaves or failures. After its joining process terminates, u sends the key server a notification message.

When u decides to leave the group, it needs to contact other users to have its user record deleted from their neighbor tables. The leave protocol

[5] All user records collected by u while it determines its ID could be used to fill its neighbor table.

presented in Silk is used to accomplish this task. After that, user u sends a leave request to the key server.

User u detects the failure of a neighbor if the neighbor does not respond to consecutive ping messages. Upon detecting the failure of a neighbor, u sends the key server a notification message. It also needs to contact some other users to look for appropriate users to replace the failed one. We refer interested readers to [25] for effective failure recovery strategies.

4.3 Performance evaluation

We evaluate the performance of our approach in this section. We first study whether T-mesh can provide low delivery latency. We then study the modified key tree by the size of the rekey message. Next, we examine whether the rekey message splitting scheme can significantly reduce rekey bandwidth overhead. Finally, we investigate the impact of different values of the delay thresholds R_i, $i = 1, 2, ..., D-1$, on the latency performance of T-mesh.

For efficiency, we wrote our own discrete event-driven simulator. We simulate the sending and the reception of a message as events. The following two topologies were used in the simulations:

- PlanetLab topology – We measured the RTT between each pair of 227 hosts on the PlanetLab infrastructure [1] using a single probe message on August 12, 2004.[6] These hosts spread in North America, Europe, Asia,

[6]We also used the smallest value of 20 RTT samples measured for each pair of PlanetLab

and Australia, and belong to various domains including .edu, .com, .net, and .org. In our simulator, we let each member (a user or the key server) correspond to a PlanetLab host, and set the RTT between each pair of members to be the same as the RTT between the corresponding two PlanetLab hosts. We set one-way delay between two members to be half of their RTT.

- GT-ITM topology – This is a transit-stub topology based on the GT-ITM topology models [10]. The topology consists of 5000 routers and 13000 network links. Each member is attached to a randomly selected router. We abstract away queueing delays in the simulations. We set the two-way propagation delay for each link in the following way. For each link within a stub domain, its delay is uniformly distributed between 0.1 and 1 millisecond. For each link connecting a stub router and a transit router, its delay is between 2 and 3 milliseconds. For each link connecting two transit routers of the same transit domain, its delay is between 10 and 15 milliseconds. For each link connecting two transit domains, its delay is between 75 and 85 milliseconds. With these settings, the relative latency performance of T-mesh to NICE [6], one of the state-of-the-art ALM schemes that we choose for comparison purpose, changes little as we change the simulation topology from PlanetLab to GT-ITM. As to the evaluation of rekey message size and rekey bandwidth overhead,

hosts, and repeated each simulation presented in Section 4.3.1. The relative performance of T-mesh to NICE (the multicast scheme for comparison) does not change.

simulation results presented in Section 4.3.2 and 4.3.3 are not sensitive to the delay settings in the GT-ITM topology.

In the simulations, we compare the performance of T-mesh with NICE [6].[7] We simulate the NICE protocol based on its protocol description [6] and the authors' simulation code.[8] In our simulation of NICE, a user will not join or leave the group until the previous join or leave terminates. In NICE, the ALM tree constructed by such sequential joins and leaves is expected to have better (at least not worse) performance than the tree constructed by concurrent joins. In all of the simulations (except the ones in Section 4.3.2) for T-mesh, we use concurrent joins and leaves. The join and leave protocols of T-mesh are based on the Silk protocols, but simplified to improve simulation efficiency. For each run of a simulation, users follow the same join and leave order in T-mesh and NICE. In all of the simulations for T-mesh, we set $D = 5$, $(R_1, R_2, R_3, R_4) = (150, 30, 9, 3)$ milliseconds, $B = 256$, and $K = 4$, unless otherwise stated. In all of the simulations for NICE, each cluster contains three to eight users [6].

[7]We did not choose Narada [13] for comparison because the structure of Narada mesh keeps changing for self-improving purposes even when there are no user joins or leaves. This incurs significant communication cost for each user to keep track of its downstream users in order to perform rekey message splitting.

[8]The NICE simulation code can be found at http://www.cs.wisc.edu/∼suman/. We did not use the code because it requires specific configurations.

4.3.1 Delivery latency

We evaluate the delivery latency of a rekey message when the key server multicasts the message in T-mesh and NICE, respectively. Given a particular user, we define three performance metrics:

- **User stress** – The total number of messages the user forwards in a multicast session.
- **Application-layer delay** (in seconds) – The latency from the time that the sender sends a message to the time that the user receives a copy of the message.
- **Relative delay penalty** (**RDP**) – The ratio of the user's application-layer delay to the one-way unicast delay from the sender to the user.

4.3.1.1 Rekeying path latency

Note that there is no notion of a key server in the original design of NICE [6]. In our simulations, to multicast a rekey message using NICE, we let the key server unicast the message to the root of the NICE tree, which is the topological center of all of the users in a group [6]. The message then traverses the tree in a top-down fashion.

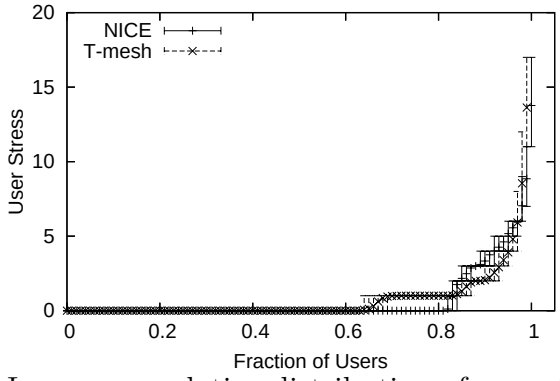
We ran simulations on the PlanetLab topology with 226 user joins. In every run of our simulations, each user join the group at a random time between 0 and 452 seconds. After all of the joins terminate, the key server multicasts a message.

Figure 4.6 plots the inverse cumulative distribution of user stress, application-layer delay, and RDP. Each curve is obtained from 100 simulation runs. For each run in Figure 4.6 (a), we changed user joining times, and started a rekey multicast session in T-mesh and NICE, respectively. We then ranked the users in increasing order of their stresses. For each rank, which corresponds to a point on x-axis, we computed the average user stress (shown as a point in the figure) of the users with this particular rank across all runs, as well as the 5 to 95-percentile value (shown as a vertical bar). Therefore, each point with coordinates (x, y) in Figure 4.6 (a) can be interpreted as: x fraction of users have an average user stress less than or equal to y. Figures 4.6 (b) and (c) can be interpreted similarly.

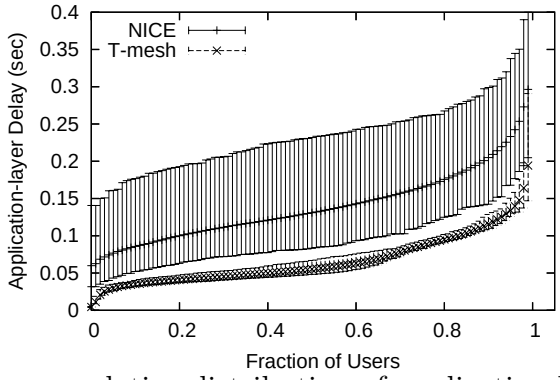
From Figure 4.6, we observe that the distributions of user stress in T-mesh and NICE are comparable; however, the users have much smaller application-layer delay and RDP in T-mesh than those in NICE. The application-layer delay in T-mesh is about half of that in NICE for the majority of users. In T-mesh, 78% of users have an RDP less than 2, and 95% of users less than 3. In NICE only 23% of users have an RDP less than 2, and 47% of users less than 3.

From Figure 4.6, we also observe that in different runs the distributions of application-layer delay and RDP have much smaller variations in T-mesh than those in NICE. This implies that the latency performance of T-mesh is less sensitive to different user joining orders than that of NICE.

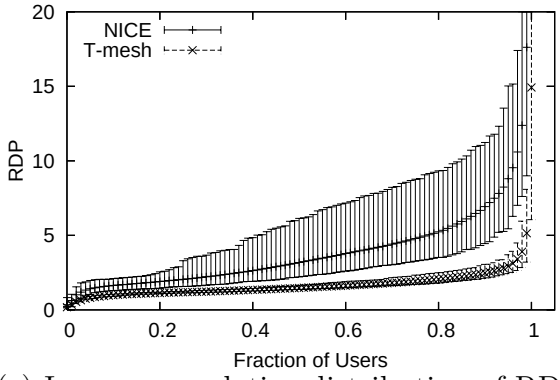
We repeated these simulations on the GT-ITM topology for 256 and



(a) Inverse cumulative distribution of user stress.



(b) Inverse cumulative distribution of application-layer delay.



(c) Inverse cumulative distribution of RDP.

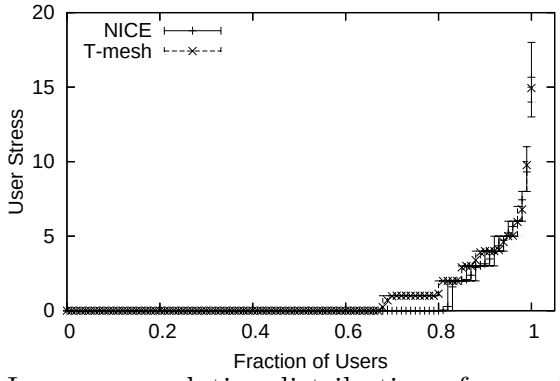
Figure 4.6: Rekey path latency on the PlanetLab topology.

1024 user joins, respectively, as shown in Figures 4.7 and 4.8. Compared with Figure 4.6, we observe that the relative performance of T-mesh to NICE has no significant change as the simulation topology changes from PlanetLab to GT-ITM.

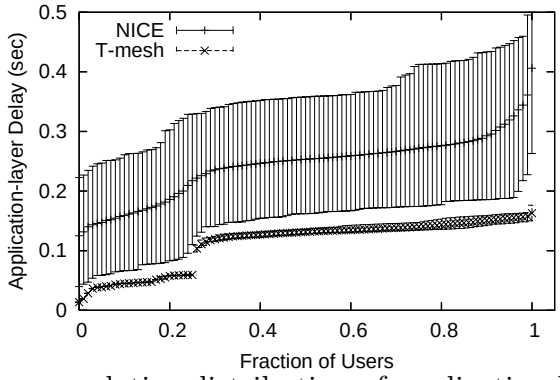
Note that it is not appropriate to conclude that T-mesh is better than NICE for data transport in general. NICE is designed for scalable group communications, and has no notion of a key server. In NICE, to determine its position in the tree, each joining user probes a smaller number of users than a joining user in T-mesh does.

4.3.1.2 Data path latency

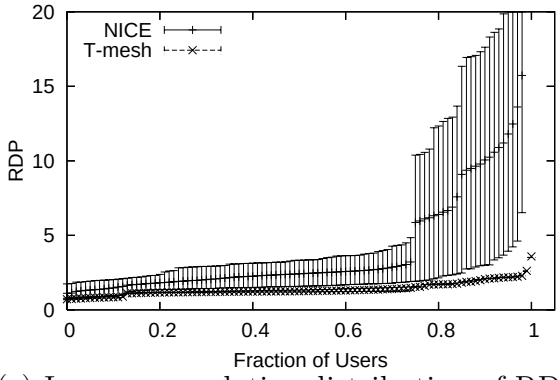
We also conducted simulations on both the PlanetLab and GT-ITM topologies to evaluate delivery latency of a data message in T-mesh and NICE, respectively, as shown in Figs 4.9 to 4.11. A random user is chosen as the sender. The multicast scheme in T-mesh is specified in Section 4.1.3. In NICE, to multicast a data message, the sender unicasts the message to the leader of its local cluster. Then the message traverses the ALM tree in a bottom-up and then top-down fashion [6]. From Figs 4.9 to 4.11, we observe that the relative performance of T-mesh to NICE is similar in data and rekey transports.



(a) Inverse cumulative distribution of user stress.

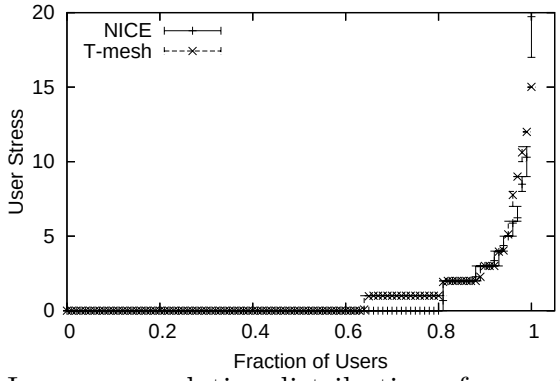


(b) Inverse cumulative distribution of application-layer delay.

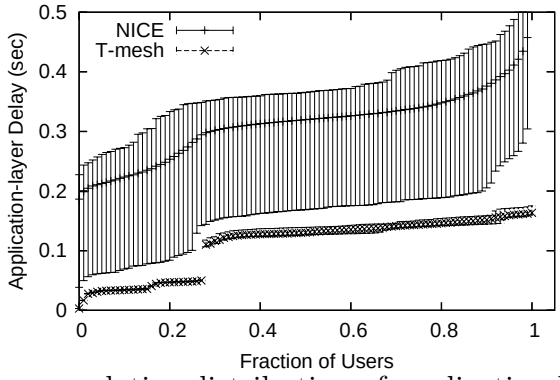


(c) Inverse cumulative distribution of RDP.

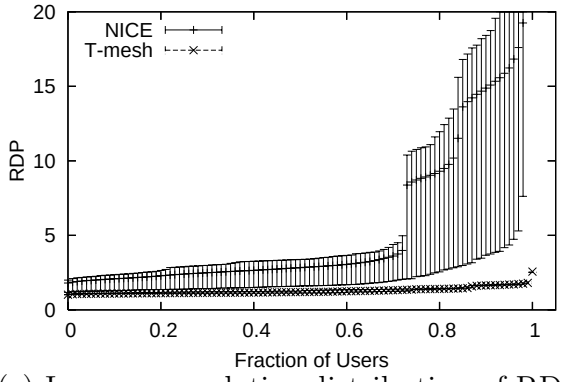
Figure 4.7: Rekey path latency on the GT-ITM topology with 256 user joins.



(a) Inverse cumulative distribution of user stress.

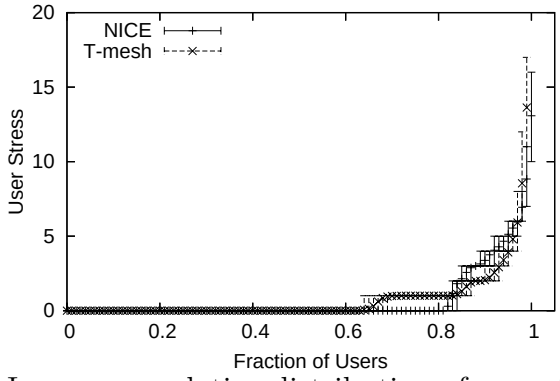


(b) Inverse cumulative distribution of application-layer delay.

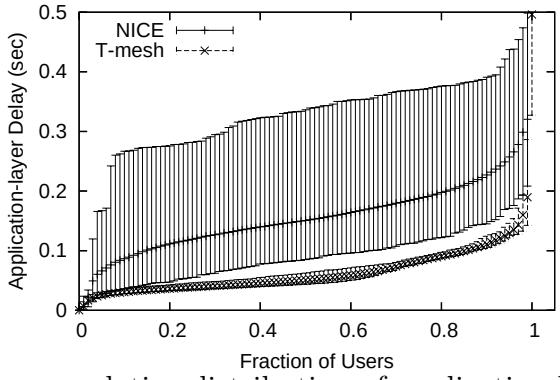


(c) Inverse cumulative distribution of RDP.

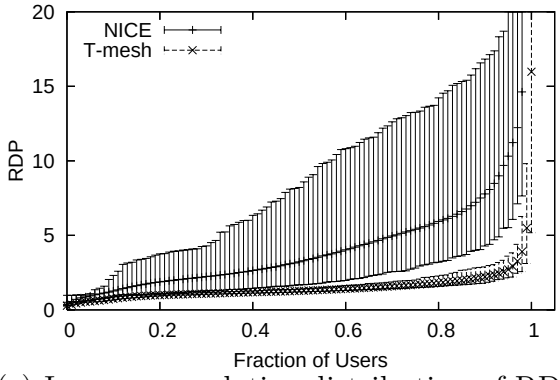
Figure 4.8: Rekey path latency on the GT-ITM topology with 1024 user joins.



(a) Inverse cumulative distribution of user stress.

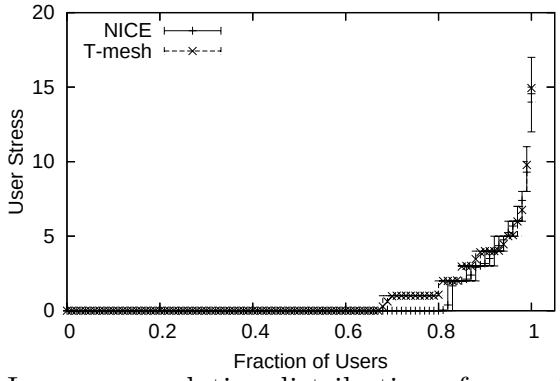


(b) Inverse cumulative distribution of application-layer delay.

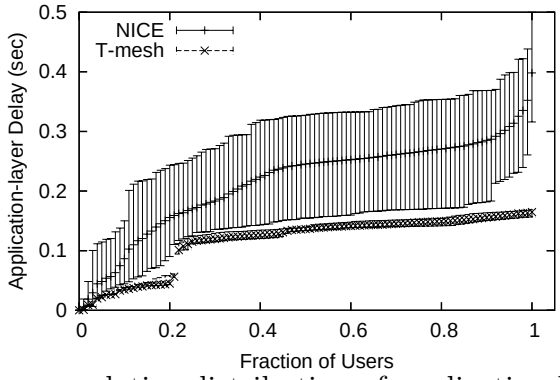


(c) Inverse cumulative distribution of RDP.

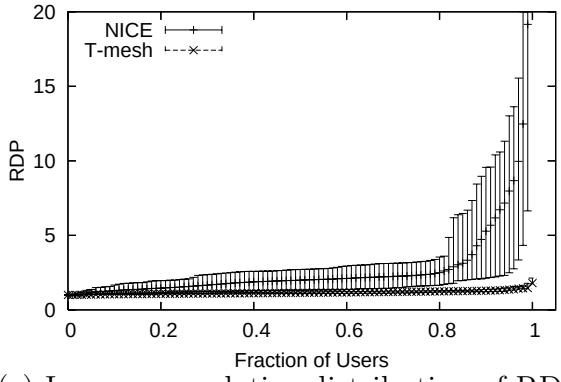
Figure 4.9: Data path latency on the PlanetLab topology.



(a) Inverse cumulative distribution of user stress.

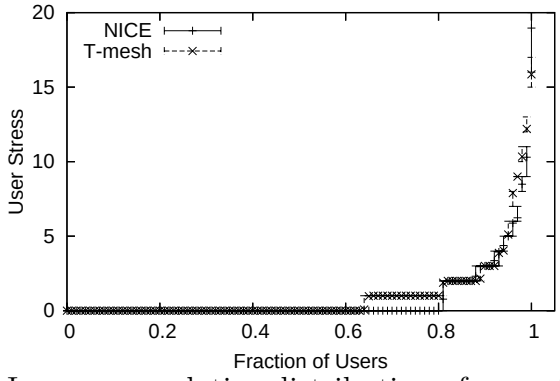


(b) Inverse cumulative distribution of application-layer delay.

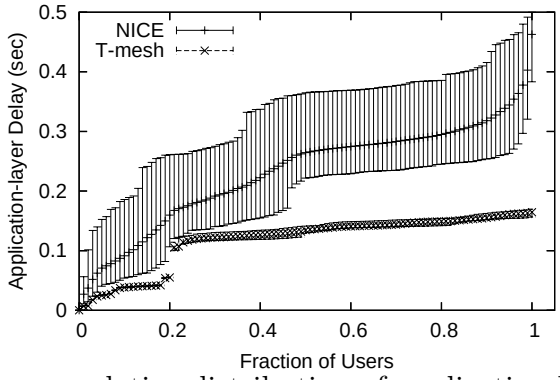


(c) Inverse cumulative distribution of RDP.

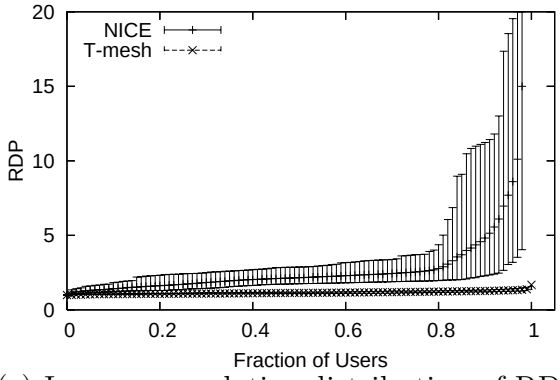
Figure 4.10: Data path latency on the GT-ITM topology with 256 user joins.



(a) Inverse cumulative distribution of user stress.



(b) Inverse cumulative distribution of application-layer delay.



(c) Inverse cumulative distribution of RDP.

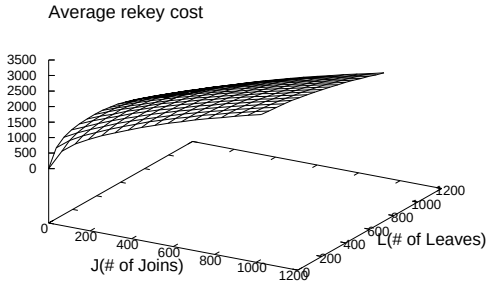
Figure 4.11: Data path latency on the GT-ITM topology with 1024 joins.

4.3.2 Rekey message size

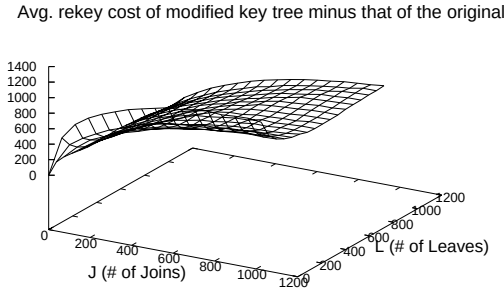
In this subsection, we study the modified key tree by the size of the rekey message. We define **rekey cost** as the number of encryptions contained in a rekey message. All of the simulations in this subsection are performed on the GT-ITM topology. In each simulation, 1024 users join the group each at a random time between 0 and 2048 seconds. After all of the joins terminate, the key server processes J join and L leave requests, $0 \leq J, L \leq 1024$, in one rekey interval, and generates one rekey message. For efficiency, we use a centralized controller to simulate the J joins and L leaves in that rekey interval.

Figure 4.12 (a) plots the average rekey cost of the modified key tree as a function of number of joins and leaves. Each average value is computed based on 20 simulation runs. Figure 4.12 (b) plots the rekey cost of the modified key tree minus that of the original key tree. The original key tree is based on the Wong-Gouda-Lam key tree [52] with degree 4 and the batch rekeying algorithm presented in Chapter 2. A degree of 4 is proved to be optimal in terms of rekey cost per join or leave [52]. After the initial 1024 users join the group, we assume that the original key tree is full and balanced. Then J joins and L leaves are processed in a rekey interval.

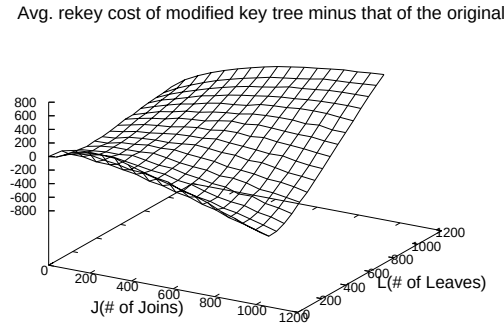
From Figure 4.12 (b), we observe that the modified key tree has a larger rekey cost than the original one for the same number of joins and leaves. This is because in the original key tree, a joining u-node can take the position of a departed u-node (see the marking algorithm presented in Appendix B.3), while in the modified key tree a joining u-node cannot replace a departed one



(a) Rekey cost of the modified key tree.



(b) Rekey cost of the modified key tree minus that of the original key tree.



(c) Rekey cost after applying the cluster rekying heuristic to the modified key tree minus that of the original key tree.

Figure 4.12: Rekey cost as a function of number of joins and leaves in the modified and the original key trees.

unless their IDs share the first $D-1$ digits. As a result, the modified key tree tends to update more keys than the original one for the same number of joins and leaves.

We propose a cluster rekeying heuristic to reduce the rekey cost of the modified key tree. In the heuristic, all of the users belonging to the same level-$(D-1)$ ID subtree are referred to as a bottom cluster. For each bottom cluster a user is selected as the leader. The leader has all of the keys on the path from its corresponding u-node to the root in the modified key tree. A non-leader user has only three keys: the group key, the user's individual key, and a pairwise key shared with its cluster leader. When a leader receives a new group key, it unicasts a copy of the group key to each user in its cluster by first encrypting the group key with the receiving user's pairwise key. With this heuristic, only the join and leave of a leader incurs group rekeying. Appendix C presents a detailed description of this heuristic.

Figure 4.12 (c) plots the average rekey cost of the modified key tree with the cluster rekeying heuristic applied minus that of the original key tree. We observe that with the heuristic, the rekey cost of the modified key tree becomes even smaller than that of the original key tree when the fraction of leaving users is small.

4.3.3 Rekey bandwidth overhead

We now evaluate whether the rekey message splitting scheme can significantly reduce rekey bandwidth overhead. We use the GT-ITM topology

protocol	*key tree approach*	*multicast scheme*	*cluster rekeying*	*rekey message splitting*
P_1	original	NICE	n/a	no
P_1'	original	NICE	n/a	yes
P_2	modified	T-mesh	no	no
P_2'	modified	T-mesh	no	yes
P_3	modified	T-mesh	yes	no
P_3'	modified	T-mesh	yes	yes
P_4	original	IP multicast	n/a	no

Table 4.2: Seven rekey protocols.

for all of the simulations in this subsection. In each simulation, 1024 users join the group each at a random time between 0 and 2048 seconds. After all of the joins terminate, the key server processes 256 joins and 256 leaves in one rekey interval of 512 seconds, and generates one rekey message. Each of the 256 joins and 256 leaves starts at a random time of the rekey interval. Such a large number of joins and leaves is not typical in practice; however, it represents a challenging scenario. If the splitting scheme works well in this scenario, then we expect that rekey transport has little interference with data transport when users join and leave less frequently.

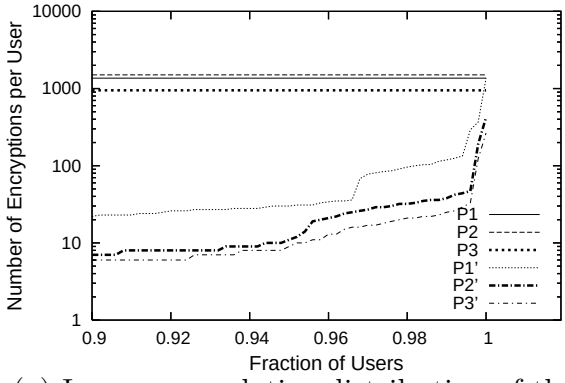
For comparison, we define seven rekey transport protocols, as specified in Table 4.2. The IP multicast scheme used in P_4 is based on the DVMRP multicast routing algorithm [14, 49]. As pointed out in Section 4.1.6, to allow rekey message splitting in P_1', users need to maintain states for $O(N)$ downstream users. In our evaluation of NICE, we did not count such maintenance cost because the cost depends on the particular maintenance protocol.

Figures 4.13 (a), (b), and (c) plot the inverse cumulative distribution

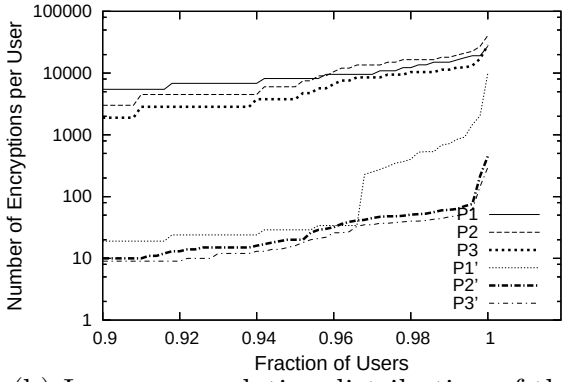
of the number of encryptions received per user, forwarded per user, and going through each of the 13000 network links, respectively. Each curve in the figure is obtained from a typical simulation run where one rekey message is distributed. Note that the y-axis is in log scale, and the x-axis starts from 0.9 or 0.96 since we are concerned with the most loaded users and links.

In Figure 4.13, by comparing P_1' to P_1, P_2' to P_2, and P_3' to P_3, we observe that rekey message splitting is very effective in reducing rekey bandwidth overhead. In particular, in P_2' and P_3' (using T-mesh), the rekey message splitting can reduce rekey bandwidth overhead for more than 90% of users and links from several thousand encryptions to less than ten encryptions. No users receive or forward more than 350 encryptions in P_2' and P_3' (see Figures 4.13 (a) and (b)). And only a few links receive up to 1500 encryptions (see Figure 4.13 (c)). These links are on the paths from the key server to its $(0, j)$-primary neighbors, $j = 0, 1, ..., B-1$. Since rekey transport and data transport choose different multicast trees in T-mesh, we expect that in P_2' and P_3' rekey transport does not affect data transport as long as the rekey bandwidth overhead at most users and most links is very small.

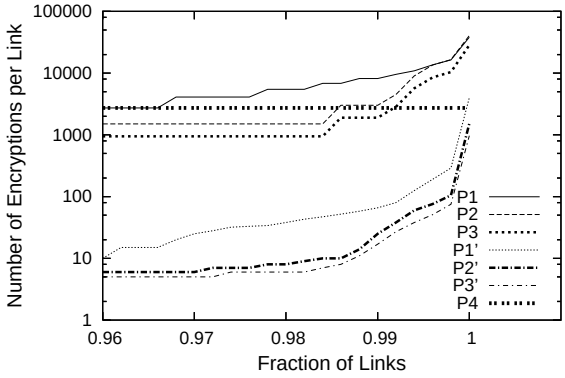
In P_1' (using NICE), however, a few users still need to forward 1000 to 10000 encryptions, and some links need to transfer up to 4000 encryptions, as shown in Figures 4.13 (b) and (c), respectively. These users and links are close to the root of the NICE tree. Congestion at these users or links can cause data and rekey message losses for many downstream users. Therefore, in P_1' the rekey bandwidth overhead of the most loaded users and links is a



(a) Inverse cumulative distribution of the number of encryptions received per user.



(b) Inverse cumulative distribution of the number of encryptions forwarded per user.



(c) Inverse cumulative distribution of the number of encryptions going through each network link.

Figure 4.13: Rekey bandwidth overhead.

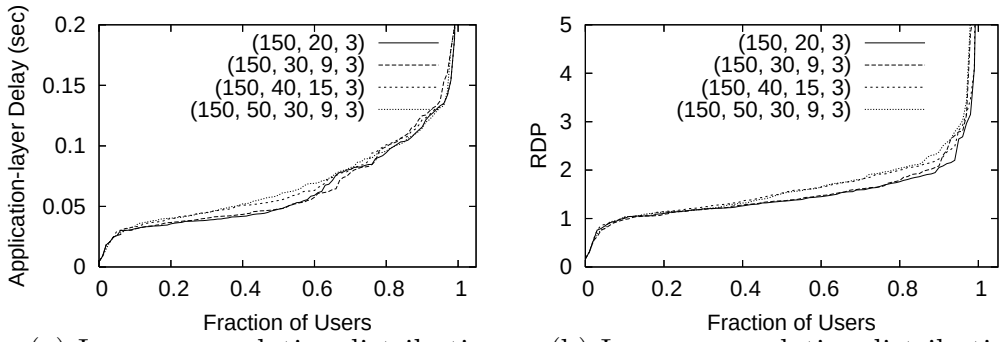
big concern.

We conclude that rekey message splitting is very effective in reducing rekey bandwidth overhead. Furthermore, it is more effective to perform message splitting in P_2' and P_3' (using T-mesh) than in P_1' (using NICE), especially for the most loaded users and links. In addition, in P_2' and P_3' each user does not need to maintain states for its downstream users to perform message splitting.

4.3.4 Delay thresholds

To determine its ID, a joining user needs to compare the RTTs between itself and the users it collected with the delay thresholds R_i, $i = 1, 2, ..., D-1$. To choose appropriate values for R_i, we use the following heuristic. First, we set R_1 around one hundred milliseconds so that all users from the same continent could belong to the same level-0 ID subtree. Second, we set R_{D-1} to be in the order of several milliseconds, so that all users in a few closely located LANs could belong to the same level-$(D-1)$ ID subtree. Last, we make the ratio of R_i/R_{i+1} larger than or equal to 2, so that each level-i ID subtree contains several level-$(i+1)$ ID subtrees.

Figure 4.14 plots the inverse cumulative distributions of application-layer delay and RDP for various values of D and $(R_1, R_2, ..., R_{D-1})$ when the key server multicasts a rekey message. The PlanetLab topology with 226 joins is used in the simulations. Each curve in the figure is obtained from a typical simulation run. From the figure, we observe that the latency performance of



(a) Inverse cumulative distribution of application-layer delay.

(b) Inverse cumulative distribution of RDP.

Figure 4.14: Rekey path latency in T-mesh for various values of D and delay thresholds $(R_1, R_2, ..., R_{D-1})$.

T-mesh is not sensitive to the various values of delay thresholds that we chose.

4.4 Related work

In addition to the rekey transport protocol presented in Chapters 2 and 3, several other rekey transport protocols were proposed recently [5, 46]. These protocols, however, are designed for IP multicast, which has not been widely deployed.

Many ALM schemes were proposed for data transport in the literature. Some schemes such as [6, 13, 23, 38, 39, 43, 64] construct topology-aware ALM trees and provide low delivery latency. These schemes work well for their target applications; however, they are not sufficient to support concurrent rekey and data transport because of the following reasons. First, it incurs $O(N)$ maintenance cost at users to allow rekey message splitting (see Section 4.1.6).

Second, the message splitting scheme that could be achieved in these ALM schemes is not as efficient as ours (see Section 4.1.6). Third, most of the ALM schemes maintains a single ALM tree [6, 23, 43, 64]. As a result, rekey traffic will further increase the load of the users that are close to the root of the ALM tree. Lastly, failure recovery in some of these schemes could be slow since it takes time to recover an explicit tree structure upon host failures.

Hypercube routing was first proposed by Plaxton, Rajaraman, and Richa (PRR) [37]. It was further explored in Pastry [42] and Tapestry [63] to provide efficient object lookup operations in distributed hash tables (DHT). In PRR, Pastry, and Tapestry, each user randomly selects its ID, which is location-independent. Random user IDs are perfect for lookup operations, but not desired for multicast, as explained in Section 4.1.6.

Two multicast schemes, namely, Scribe [43] and Bayeux [64], were proposed on top of the Pastry and Tapestry infrastructures. Scribe and Bayeux were designed to support many small multicast groups, and a single ALM tree is constructed for each multicast group. Therefore, Scribe and Bayeux are different from our multicast scheme.

Ng and Zhang proposed a global network positioning (GNP) scheme [31]. With this scheme, the delay between two hosts can be estimated using their GNP coordinates. This scheme can be used in our system to reduce the probing cost of each joining user. For example, if the key server knows the GNP coordinates of all users, it can determine the ID for a joining user by centralized computing.

4.5 Summary

In this chapter, we proposed an application-layer multicast approach that supports concurrent rekey and data transport. Our goal is to provide fast delivery of rekey messages and reduce rekey bandwidth overhead as much as possible. Our approach consists of a multicast scheme using neighbor tables, a modified key tree, and a rekey message splitting scheme. These system components are integrated with a coherent scheme to identify each user, key, and encryption. By virtue of the identification scheme, each user can determine who are the next hops by looking up its neighbor tables in a multicast session. Also each user can determine whether an encryption is needed by its downstream users by checking the encryption's ID. Furthermore, our user ID assignment scheme exploits proximity in the underlying network such that each multicast tree embedded in the neighbor tables tends to be topology-aware. Our simulation results showed that our approach can achieve much smaller delivery latency and rekey bandwidth overhead for almost all users (and links) than a representative existing ALM scheme.

Chapter 5

Future work

In the dissertation, we have studied how to use ALM to support both rekey and data transport for secure group communications. We have proposed an ALM framework using hypercube neighbor tables. The ALM framework has the properties of topology-awareness and dynamic routing. More specifically, the neighbor tables embed many multicast trees rooted at the key server and each user. Each of the multicast trees tends to be topology-aware. Furthermore, for each multicast session, users do not maintain parent and child states to form a static multicast tree. Instead, each user can determine who are the next hops by looking up its neighbor table according to the `forward_level` field of the multicast message.

With the Topology-awareness and dynamic routing properties, our ALM framework has the potential to provide the following benefits for data transport.

- Low delivery latency. In secure group communications applications such as teleconference and multi-party games, data traffic requires fast delivery. The topology-awareness property of our ALM framework helps to achieve this purpose.

- High throughput. As observed in SplitStream [11], the capacity of outgoing user access links is usually the bottleneck that limits overall system throughput. In a single ALM tree, the outgoing bandwidth of leaf users of the tree are not utilized. Since our ALM framework embeds many multicast trees rooted at each user, we can build multiple ALM trees for each multicast source, and use each tree to transfer part of data traffic. This helps to utilize each user's outgoing link capacity and thus improve overall system throughput.

- Fast failure recovery and quick adaption to network dynamics. The dynamic routing property of our ALM frameworks helps to provide the benefit if each user stores multiple neighbors in each neighbor table entry. Once a user detects the failure of a next hop, or detects congestion on the path to a next hop by observing burst losses, it can simply forward messages to another neighbor in the same table entry as the failed or congested neighbor. At the same time, the user needs to look for another neighbor to replace the failed or congested one.

- Supporting user heterogeneity. Users have various incoming and outgoing capacities for their access links. To support heterogeneous outgoing link capacities, we observe that a user can forward a received message with `forward_level`$= i$ only to some of its (i, j)-ID subtrees, and ask its next hops to take care of the remaining ID subtrees. To support heterogeneous incoming link capacities, it is desired to let those users

with low incoming link capacities close to leaf level in multicast trees. For this purpose, we can include the information of available incoming bandwidth for each neighbor in neighbor tables. In this way, a user can consider both RTT and bandwidth measures when it selects its next hop from multiple candidates.

Although many ALM schemes have been proposed in the literature, they do not offer all of the benefits discussed above. Existing topology-aware ALM schemes such as [6, 13, 23, 38, 39, 43, 64] do not provide high throughput since they build a single ALM tree for each multicast source. SplitStream [11] uses multiple trees to increase overall system throughput, but it does not have the properties of topology-awareness and dynamic routing.

As our future work, we will explore the topology-awareness and dynamic routing properties of our ALM framework, and investigate how to improve our multicast scheme for data transport. In particular, we plan to implement a prototype of the ALM framework, and then conduct extensive experiments in the PlanetLab infrastructure to evaluate the topology-awareness property of the framework. After that, we plan to explore the dynamic routing property of the framework, and investigate how to provide high throughput and quick adaptation to network dynamics for data transport. In a long term, we plan to extend the framework to a general-purpose overlay network, and study how to support a range of applications including grid computing and distribution of streaming media.

Chapter 6

Conclusion

In the dissertation, we have studied how to provide scalable, reliable, and real-time rekey transport for secure group communications. We have identified the special properties and requirements of rekey transport, and investigated various technical issues involved. In particular, We have proposed two efficient rekey transport protocols. One runs on top of IP multicast or other multicast schemes; the other is designed especially for application-layer multicast. The efficiency of both protocols benefits from the special properties of rekey transport.

Our first rekey transport protocol runs in two steps: a multicast step followed by a unicast recovery step. We propose the use of proactive forward error correction (FEC) in multicast to reduce delivery latency and limit the number of users who need unicast recovery. The unicast recovery step provides eventual reliability; it also reduces the worst-case delivery latency as well as user bandwidth overhead. To make proactive FEC effective for rekey transport, we investigate how to choose FEC block size to achieve both small processing and bandwidth overheads, and how to space the sending times of packets to make proactive FEC resilient to burst loss. We further propose

an adaptive FEC scheme that makes the number of users who need unicast recovery controlled around a small target value under dynamic network conditions. The adaptive FEC scheme also makes rekey interval small to achieve tight group access control.

Since the key server changes the key tree structure frequently, we need an efficient method for each user to identify the small subset of new keys it needs after rekeying. In particular, we design a new marking algorithm for updating the key tree, together with a key identification scheme, key assignment algorithm, and block ID estimation algorithm, such that each user can identify its new keys with a low bandwidth overhead.

Our second rekey transport protocol explores the naming and routing opportunities offered by application-layer multicast. Our approach consists of a multicast scheme using hypercube neighbor tables, a modified key tree, and a rekey message splitting scheme. These system components are integrated with a coherent scheme to identify each user, key, and encryption. By virtue of the identification scheme, each user can determine who are the next hops by looking up its neighbor tables in a multicast session. Also each user can determine whether an encryption is needed by its downstream users by checking the encryption's ID. In our protocol, each user receives only the encryptions needed by itself or its downstream users. Furthermore, we propose a user ID assignment scheme that exploits proximity in the underlying network. As a result, each multicast tree embedded in neighbor tables tends to be topology-aware.

We formulate and prove correctness properties for the multicast scheme

and rekey message splitting scheme. Our simulation results have showed that our approach can achieve much smaller delivery latency and rekey bandwidth overhead for almost all users (and links) than a representative existing application-layer multicast scheme.

Appendices

Appendix A

Proofs of lemmas and theorems

Proof of Lemma 2.2.2: 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.

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

1. 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.

2. 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 Lemma 2.2.1: Obviously hold according to the marking algorithm specification. See Figure B.7. □

Proof of Lemma 2.2.3:

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

1. 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 2.2.2, $n_k < m'$ since m' is a u-node.

2. 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 $m' \leq d \cdot n_k + d$.

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:

1. 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.

2. 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 2.2.2 since m' is a u-node.

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

Proof of Theorem 3.3.1: We first prove $h' \leq h^*$. Suppose $h' > h^*$. Then we have $h' - 1 \geq h^*$, and also $T' \geq T^*$ by the definition of T^*. Thus we have $f(h'-1, T') \leq f(h^*, T^*)$ since $f(h,T)$ is a non-increasing function of T and h. Therefore we have $f(h^*, T^*) > r^*$ by Inequality 3.18. This contradicts the definitions of h^* and T^*.

We next prove $T' - T^* \leq \frac{BW_u(r^*) - BW_u(f(h^*,T^*))}{b(t)}$. In our group rekeying protocol, multicast traffic is an increasing function of h. Hence by Equation 3.16 we have

$$\begin{aligned} T' \cdot b(t) &= BW_m(h') + BW_u(r^*) \\ &\leq BW_m(h^*) + BW_u(r^*) \\ &= (BW_m(h^*) + BW_u(f(h^*,T^*))) + \\ &\quad (BW_u(r^*) - BW_u(f(h^*,T^*))) \\ &\leq T^* \cdot b(t) + BW_u(r^*) - BW_u(f(h^*,T^*)). \end{aligned}$$

Therefore, we have $T' - T^* \leq \frac{BW_u(r^*) - BW_u(f(h^*,T^*))}{b(t)}$. □

Proof of Lemma 4.1.1: By Definition 4.1.1, all of the members sharing the common prefix $u.ID[0:i-1]$ belong to the same level-i ID subtree. So we only need to prove that the IDs of u and all of its downstream members have the common prefix $u.ID[0:i-1]$. We prove by induction on the forwarding level.

1) At forwarding level i, member u's ID has the prefix $u.ID[0:i-1]$.

2) Assume that at forwarding levels i, $i+1$, ..., and s, the ID of any

downstream member of u at these levels has the prefix $u.ID[0:i-1]$, where $s \geq i$.

3) Now consider the forwarding level $s+1$. Let w be any downstream member of u at this level. We observe that w must be a $(s, w.ID[s])$-neighbor of its previous hop (say z). By Definition 4.1.3, the IDs of w and z have the common prefix $z.ID[0:s-1]$. By the induction assumption, $z.ID$ has the prefix $u.ID[0:i-1]$ since z's forwarding level is between i and s (inclusively). It follows that that $w.ID$ has the prefix $u.ID[0:i-1]$ since $s \geq i$. □

Lemma A.0.1. *In a multicast session, given any two distinct positions at forwarding levels i and j respectively in the multicast tree, let u_i and w_j be the corresponding member(s) at these two positions, $0 \leq i \leq D$, $0 \leq j \leq D$, and $j \leq i$. Then we have $u_i.ID[0:i-1] \neq w_j.ID[0:i-1]$. Furthermore, if w_j is not an upstream member of u_i, then we have $u_i.ID[0:j-1] \neq w_j.ID[0:j-1]$.*

Proof of Lemma A.0.1: Let V_m be the set of all of the members at forwarding level m, where $0 \leq m \leq D$. Note that V_0 contains only a single element, the sender. Since u_i and w_j are in two distinct positions and $j \leq i$, u_i's forwarding level must be larger than or equal to 1. Consider two cases.

Case 1: w_j is an upstream member of u_i. Let $u_{i'}$, $u_{i'} \in V_{i'}$, be the upstream member of u_i whose previous hop is w_j. (Note that $u_{i'}$ and u_i refer to the same member if the previous hop of u_i is w_j.) Then $u_{i'}$ is a neighbor at the $(i'-1)$th row of w_j's neighbor table. Thus we have $u_{i'}.ID[0:i'-1] \neq$

$w_j.ID[0 : i' - 1]$. By Lemma 4.1.1, $u_{i'}.ID[0 : i' - 1]$ is a prefix of $u_i.ID$. So we have $u_i.ID[0 : i - 1] \neq w_j.ID[0 : i - 1]$ since $i' \leq i$.

Case 2: w_j is not an upstream member of u_i. Let v be the common upstream member of u_i and w_j who is at the largest forwarding level. That is, for any v' who is a common upstream member of u_i and w_j, the forwarding level of v' is smaller than or equal to that of v. Let $u_{i'}$, $u_{i'} \in V_{i'}$, be the upstream member of u_i whose previous hop is v. Let $w_{j'}$, $w_{j'} \in V_{j'}$, be the upstream member of w_j whose previous hop is v. Note that $u_{i'}$ and u_i refer to the same member if the previous hop of u_i is v, and $w_{j'}$ and w_j refer to the same member if the previous hop of w_j is v. Then $u_{i'}$ and $w_{j'}$ are two distinct primary neighbors at the $(i' - 1)$th and $(j' - 1)$th row of v's neighbor table. So we have $u_{i'}.ID[0 : i' - 1] \neq w_{j'}.ID[0 : i' - 1]$ and $u_{i'}.ID[0 : j' - 1] \neq w_{j'}.ID[0 : j' - 1]$. By Lemma 4.1.1, $u_{i'}.ID[0 : i' - 1]$ is a prefix of $u_i.ID$, and $w_{j'}.ID[0 : j' - 1]$ is a prefix of $w_j.ID$. Since $i' \leq i$ and $j' \leq j$, we have $u_i.ID[0 : i - 1] \neq w_j.ID[0 : i - 1]$ and $u_i.ID[0 : j - 1] \neq w_j.ID[0 : j - 1]$. □

Proof of Lemma 4.1.2: Let j be w's forwarding level. By Lemma A.0.1, we have $i < j$. Furthermore, u must be an upstream member of w since $u.ID[0 : i - 1] = w.ID[0 : i - 1]$. □

Proof of Theorem 4.1.3: Since no message is lost and group membership is static, each member appears in at most one position in the multicast tree. So we only need to prove that each member appears in at least one position the multicast tree. Prove by contradiction.

Suppose member w is not in the multicast tree. Let V_i, $i = 0, 1, ..., D$, be the set of members who are at forwarding level i and $v_i[0 : i-1] = w.ID[0 : i-1]$, for any member v_i, $v_i \in V_i$. Obviously, the sender is in V_0. Let V_j be the last non-empty set among $V_0, V_1, ..., V_D$, that is, V_j is non-empty, and V_{j+1}, V_{j+2}, ..., V_D are all empty. Let z_j be a member in V_j and let s be the number of digits contained in the longest common prefix of the IDs $w.ID$ and $z_j.ID$. Then we have $s \geq j$ by the definition of V_j.

Member w is a potential $(s, w.ID[s])$-neighbor of z_j. Since all neighbor tables are 1-consistent, the $(s, w.ID[s])$-entry of z_j's neighbor table is not empty by Definition 4.1.3. Then the primary $(s, w.ID[s])$-neighbor of z_j must be at forwarding level $s+1$ in the multicast tree since z_j is at forwarding level j and $s \geq j$. As a result, the primary $(s, w.ID[s])$-neighbor of z_j is a member of set V_{s+1}. This contradicts the assumption that V_j is the last non-empty set since $s \geq j$. □

Proof of Theorem 4.1.5: We first prove that encryption e is needed by at least one member in V if $e.ID$ is a prefix of $w.ID[0 : s]$, or $w.ID[0 : s]$ is a prefix of $e.ID$. Note that w must be at forwarding level $s + 1$, so all of the members in V have the common prefix $w.ID[0 : s]$ by Lemma 4.1.1.

Case 1: $e.ID$ is a prefix of $w.ID[0 : s]$. In this case, all of the members in V need this encryption by Lemma 4.1.4.

Case 2: $w.ID[0 : s]$ is a prefix of $e.ID$. In this case, only the members whose IDs have the prefix $e.ID$ need this encryption. Such a member must

exist in the group; otherwise, the key server will not generate e. Furthermore, by Lemma 4.1.2, such a member must belong to V since the ID of such a member has the prefix $w.ID[0:s]$.

Next we prove that if e is needed by at least one member in V, then $e.ID$ is a prefix of $w.ID[0:s]$, or $w.ID[0:s]$ is a prefix of $e.ID$.

If e is needed by at least one member (say z) in V, then by Lemma 4.1.4, $e.ID$ is a prefix of $z.ID$. Since $w.ID[0:s]$ is also a prefix of $z.ID$ (by Lemma 4.1.1), we have that either $e.ID$ is a prefix of $w.ID[0:s]$, or $w.ID[0:s]$ is a prefix of $e.ID$. □

Appendix B

Protocol Specification

B.1 Server protocol and user protocol

The protocol for the key server is specified in Figure B.1, and the protocol for a user is specified in Figure B.2. In both protocols, we consider only one rekey message.

1. run a marking algorithm to generate encryptions
2. run a key assignment algorithm to construct rekey packets
3. partition the sequence of rekey packets into blocks such that each block contains k rekey packets
4. generate h parity packets for each block
5. status $\leftarrow$ MULTICAST
6. **for** each block **do** multicast k rekey packets and h parity packets
7. $R \leftarrow$ empty set $\triangleright$ R is the set of users who send NACKs
8. $A \leftarrow$ empty list $\triangleright$ A contains NACK information
9. start a timer
10. **when** receiving a NACK (a list of $< a_i, i >$) from user m **do**
    $\triangleright < a_i, i >$: user m requests a_i parity packets for block i
13. **if** (status = MULTICAST) **then**
14. $R \leftarrow R + \{m\}$
15. $i_m \leftarrow$ ID of the block containing the specific rekey packet of user m
16. $a_{i_m} \leftarrow$ number of parity packets that user m requests for block i_m
17. put a_{i_m} into A
18. **else** send unicast recovery packets to user m
19. **when** timeout **do**
20. run the algorithm AdjustRho (A)
21. status $\leftarrow$ UNICAST
22. send unicast recovery packets to each user in R

Figure B.1: Key server protocol for one rekey message.

In the key server protocol specified in Figure B.1, the key server starts to send unicast recovery packets at the end of the multicast round. However, to provide fast recovery, the key server can send unicast recovery packets to a user once it receives a NACK from the user.

In group rekeying, each user needs to know when a new group key has been distributed. Then the user can set a timer as shown in line 1 of Figure B.2. A user can figure out that a new group key is being or has been distributed in the following two cases. In the first case, the user receives any rekey or parity packet with a new rekey message ID. In the second case, the user receives some data packets that are encrypted by a new group key that has a new version number.

B.2 Packet format

Figures B.3, B.4, B.5 and B.6 specify the formats of rekey, parity, unicast recovery, and $NACK$ packets, respectively. Each number in parentheses is the suggested field length, in number of bits. In a unicast recovery packet, the encryption IDs are optional if we arrange the encryptions in increasing order of ID.

B.3 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

```
1.  start timer1
2.  for each block ID i do counter[i] ← 0
3.  when receiving a packet pkt do
4.     if pkt is a unicast recovery packet then
5.        update my user ID according to the new user ID contained in pkt
6.        retrieve required encryptions from the packet
7.        cancel all timers
8.        return
9.     else
10.       if pkt is a rekey packet then
11.          m ← new user ID derived
12.          set my user ID as m
13.          if (pkt.frmID ≤ m ≤ pkt.toID) then
                ▷ this packet contains my required encryptions
14.             retrieve required encryptions from the packet, and cancel all timers
15.             return
16.          else
17.             if pkt is not a duplicate then
18.                execute the algorithm EstimateBlkID (m, high, low, pkt)
19.             increase counter[pkt.blkID] by 1
20. when timer1 timeout do
21.    if (high = low) and (counter[high] ≥ k) then
22.       decode the block with block ID high, and retrieve required encryptions
          from the decoded rekey packet
23.       return
24.    else
25.       for each block ID i ∈ [low, ..., high] do
26.          if (counter[i] ≥ k) then
27.             decode the block
28.             if my specific rekey packet is in the block then
29.                retrieve required encryptions
30.                return
31.          else put <k − counter[i], i> into a NACK packet
32.       unicast the NACK packet to the key server, and start timer2
33. when timer2 timeout do
34.       unicast the NACK packet to the key server, and start timer2
```

Figure B.2: User protocol for one rekey message.

1. Type: rekey packet(3)	2. Duplication flag (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)	

Figure B.3: Format of a rekey packet.

1. Type: parity packet (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 rekey packets	

Figure B.4: Format of a parity packet.

1. Type: unicast recovery packet(3)	2. Reserved (1)
3. Rekey message ID (12)	4. New user ID (16)
5. A list of <encryption, ID> (variable)	

Figure B.5: Format of a unicast recovery packet.

1. Type: NACK packet(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)	

Figure B.6: Format of a NACK packet.

the following marking algorithm to update the key tree and construct a rekey subtree. The marking algorithm is different from those presented in [27, 55].

The marking algorithm consists of two steps. In the first step, the algorithm modifies the structure of the key tree to satisfy the join and leave requests. The operations for this step are specified in Figure B.7. The n-node and ID information used in the algorithm are explained in Section 2.2.

In the second step, the marking algorithm constructs a rekey subtree. The operations are specified in Figure B.8. The input to the second step of

the algorithm is a copy of the updated key tree. The algorithm will label all nodes and then prune the tree. We call the remaining subtree **rekey subtree**. Each edge in the rekey subtree corresponds to an encryption. 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, and thus generating a sequence of encryptions. The key server then runs the key assignment algorithm to assign encryptions into rekey packets.

```
▷ input: key tree, J join and L leave requests
▷ output: updated key tree
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 of 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 maxKID + 1 and
       d · maxKID + d (inclusive) in order of from low to high
12.    ▷ maxKID is the largest k-node ID
13.    while there are still extra joins do
          ▷ split the u-node maxKID + 1 to accommodate extra joins
14.       move the u-node maxKID + 1 to a new position d · maxKID + 1
          ▷ now this u-node becomes a left-most child of its old position
15.       add a new k-node with ID maxKID + 1
16.       add, in increasing order of ID, up to d − 1 new joins as child u-nodes
          of the k-node maxKID + 1
17.       update the value of maxKID
18. for each n-node do
19.    if the n-node has a descendant u-node then
20.       change the n-node to k-node
```

Figure B.7: Marking algorithm step 1: updating the structure of the key tree.

```
▷ input: a copy of updated key tree
▷ 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 of 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
19. change the key in each k-node in the rekey subtree
```

Figure B.8: Marking algorithm step 2: constructing a rekey subtree.

B.4 Estimating Block ID

In our protocol, the key server partitions rekey packets into multiple blocks. If a user lost its specific rekey packet, the user cannot know directly to which block its specific rekey packet belongs. We address this issue here.

A user can estimate the ID of the block to which its specific rekey packet belongs from the ID information contained in the received rekey packets. Suppose a user's specific rekey packet is the j^{th} packet in block i, where $i \geq 0$ and $0 \leq j \leq k-1$. Let $<i, j>$ denote the <block ID, sequence number within a block> pair. Whenever a user receives a rekey packet, it first derives its new user ID, denoted by m, and then it refines its estimation of the block

ID i. For example, if the received rekey packet is not a duplicate and m is larger than $toID$ field of this packet, then i should be larger than or equal to the block ID of the received packet. This is because the received rekey packet must be generated earlier than the user's specific rekey packet. In this way, if the user can receive any one rekey packet whose <block ID, sequence number> pair is in the set $S_l = \{<i-1,\ k-1>,\ <i,\ 0>,...,\ <i,\ j-1>\}$, and receive any one rekey packet in the set $S_u = \{<i,\ j+1>,\ ...,\ <i,\ k-1>,\ <i+1,\ 0>\}$, then it can determine the precise value of i even if the packet $<i,\ j>$ is lost. Figure B.9 illustrates the idea of block ID estimation. The detailed algorithm to estimate block ID is specified in Figure B.10.

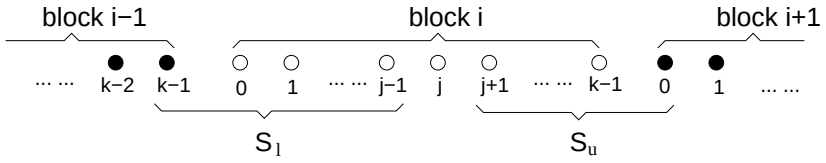


Figure B.9: Illustration of block ID estimation.

A user can determine the precise value of the ID of its required block with high probability. The probability of such failure is as low as $p^{j+2} + p^{k-j+1} - p^{k+2}$, as stated in Lemma B.4.1. In the worst case when $j = 0$ or $j = k-1$, the probability is about p^2.

Lemma B.4.1. *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 rekey packet belongs is about $p^{j+2} + p^{k-j+1} - p^{k+2} = O(p^2)$, where j is the sequence*

number of its specific rekey packet, and k is the block size, $0 \leq j \leq k-1$ and $k \geq 1$.

Proof of Lemma B.4.1: As illustrated in Figure B.9, only if all of the rekey packets whose <block ID, sequence number> pairs are in the set $S_l + \{<\text{i}, j>\}$ are lost, or when all of the rekey packets in the set $S_u + \{<\text{i}, j>\}$ are lost, the user cannot determine the precise value of the ID of its required block. 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 the ID of its required block, 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 considers only the number of parity packets requested for the user's required block when it adjusts the proactivity factor.

The algorithm EstimateBlkID specified in Figure B.10 has four inputs. Integer m is the update-to-date ID of the user who runs this routine. Variable *low* is the current estimate of the lower bound of the ID of the user's required block, and *high* is the current estimate of the upper bound. Packet *pkt* is the rekey packet that the user just received.

In the algorithm, a user sets the initial values of the lower bound *low* and upper bound *high* as 0 and ∞, respectively. However, the **if** statements of lines 10-15 guarantee that eventually *high* will not be infinity if the user

```
EstimateBlkID (m, low, high, pkt)
▷ m: update-to-date ID of the user executing this routine
▷ low: current estimate of the lower bound of the ID of the required block
▷ high: current estimate of the upper bound of the ID of the required block
▷ pkt: rekey packet just received
1.  if pkt is a duplicate then     ▷ duplicate packets are only in the last block
2.      high ← min{high, pkt.blkID}
3.  else if (pkt.toID ≤ m ≤ pkt.frmID) then
4.      high ← pkt.blkID
5.      low ← pkt.blkID
6.  else if (m > pkt.toID) and (pkt.seqNo = k − 1) then
7.      low ← max{low, pkt.blkID + 1}
8.  else if (m > pkt.toID) and (pkt.seqNo < k − 1) then
9.      low ← max{low, pkt.blkID}
10. else if (m < pkt.frmID) and (pkt.seqNo = 0) then
11.     high ← min{high, pkt.blkID − 1}
12. else if (m < pkt.frmID) and (pkt.seqNo > 0) then
13.     high ← min{high, pkt.blkID}
14. else if (m > pkt.toID) then
15.     high ← min{high, pkt.blkID + ⌈(d·(pkt.maxKID+1) − pkt.toID − (k−1−pkt.seqNo)) / k⌉}
```

Figure B.10: Estimating the ID of the required block.

receives any rekey packet, say pkt. Consider two cases. In the first case, the packet pkt is a duplicate, then we have $high \leq pkt.blkID$ since the duplicate packet must belong to the last block. In the second case, the packet pkt is not a duplicate. We know that the $maxKID$ field of the packet specifies the largest ID of current k-nodes. Therefore, all user IDs must be in the range of $maxKID + 1$ and $d \cdot (pkt.maxKID + 1)$, inclusively. In the worst case, one rekey packet contains encryptions for only one user; then there are at most $(d \cdot (pkt.maxKID + 1) - pkt.toID)$ rekey packets each with $frmID$ sub-field larger than $pkt.toID$. Among these rekey packets, $k - 1 - pkt.seqNo$ of them are in the block $pkt.blkID$. Therefore, the maximum block ID cannot be larger than $pkt.blkID + \lceil \frac{d \cdot (pkt.maxKID+1) - pkt.toID - (k-1-pkt.seqNo)}{k} \rceil$.

Appendix C

Cluster rekeying heuristic for T-mesh

In the heuristic, all of the users belonging to the same level-$(D-1)$ ID subtree are referred to as a bottom cluster. For each bottom cluster, the user with the earliest joining time among all of the users in the cluster is selected as the cluster leader. A user's joining time is the time that the key server assigns the user's ID, and it is based on the key server's local clock. Each user record in neighbor tables contains the joining time and public key of a neighbor, in addition to the neighbor's IP address and ID.

A leader has all of the keys on the path from its corresponding u-node to the root in the modified key tree. It also shares a pairwise key with each of the other users in its cluster. A non-leader user has only three keys: the group key, the user's individual key, and a pairwise key shared with its cluster leader.

In the heuristic, a joining user determines its user ID and constructs its neighbor table in the same way as described in the main text. The message multicast process is as usual when forwarding level is less than $D-1$. At forwarding level $D-1$, when a non-leader user receives a rekey message with `forward_level` $= D-1$, it forwards the message to its cluster leader. When a

leader receives a rekey message with `forward_level` $\geq D-1$, it first extracts the new group key, and then unicasts a copy of the group key to each user in its cluster by first encrypting the group key with the receiving user's pairwise key. [1]

A non-leader user's join or leave does not incur group rekeying. To join a bottom cluster, the user (say u) first gets from the key server the user record of the cluster leader (say w) and a joining certificate. The joining certificate is u's user record signed by w's individual key. User u then sends the certificate to w. After verifying the certificate, w establishes a pairwise key with u using SSL. To leave a cluster, u first requests w to sign a leaving certificate with w's individual key. The leaving certificate contains u's user record and a timestamp. User u then presents the certificate to the key server.

A cluster leader's join or leave incurs group rekeying. A cluster leader (say w) is always the first join in its cluster. The key server follows the regular rekeying procedure to process its join. To leave the group, w sends the new leader (if it exists), say v, the following information: all of the keys on the path from w's corresponding u-node to the root in the key tree, and user records of all of the other users in the cluster. After receiving from v a leaving certificate signed by v's individual key, w presents the certificate to the key server. Meanwhile, v establishes a pairwise key with each remaining user in

[1] As we can see, it is desired to let cluster leaders, instead of non-leader users, receive rekey messages at forwarding level D, For this purpose, in every table entry at the $(D-2)$th row of each neighbor table, the neighbor with the earliest joining time should be chosen as the primary neighbor.

the cluster.

Bibliography

[1] *PlanetLab project*. http://www.planet-lab.org/.

[2] Omar Ait-Hellal, Eitan Altman, Alain Jean-Marie, and Irina A. Kurkova. On loss probabilities in presence of redundant packets and several traffic sources. *Performance Evaluation*, 36–37:486–518, 1999.

[3] Eitan Altman, Chadi Barakat, and Victor Ramos. Queueing analysis of simple fec schemes for ip telephony. In *Proceedings of IEEE INFOCOM 2001*, pages 796–804, Anchorage, Alaska, April 2001.

[4] David Balenson, David McGrew, and Alan Sherman. Key management for large dynamic groups: One-way function trees and amortized initialization, INTERNET-DRAFT. URL: http://www.securemulticast.org/smug-drafts.htm, September 2000.

[5] Suman Banerjee and Bobby Bhattacharjee. Scalable secure group communication over IP multicast. *JSAC Special Issue on Network Support for Group Communication*, 2002.

[6] Suman Banerjee, Bobby Bhattacharjee, and Christopher Kommareddy. Scalable application layer multicast. In *Proceedings of ACM SIGCOMM 2002*, pages 205–217, Pittsburgh, PA, August 2002.

[7] Jean-Chrysostome Bolot, Sacha Fosse-Parisis, and Don Towsley. Adaptive FEC-based error control for Internet Telephony. In *Proceedings of IEEE INFOCOM '99*, pages 1453–1460, March 1999.

[8] Bob Briscoe. Marks: Zero side effect multicast key management using arbitrarily revealed key sequences. In *Proceedings of NGC 1999*, pages 301–320, Pisa, Italy, November 1999.

[9] John W. Byers, Michael Luby, Michael Mitzenmacher, , and Ashu Rege. A digital fountain approach to reliable distribution of bulk data. In *Proceedings of ACM SIGCOMM '98*, pages 56–67, Vancouver, B.C., September 1998.

[10] Ken Calvert, Matt Doar, and Ellen W. Zegura. Modeling Internet topology. *IEEE Communications Magazine*, 35(6):160–163, June 1997.

[11] Miguel Castro, Peter Druschel, Anne-Marie Kermarrec, Animesh Nandi, Antony Rowstron, and Atul Singh. Splitstream: High-bandwidth multicast in a cooperative environment. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP'03)*, Lake George, New York, October 2003.

[12] Isabella Chang, Robert Engel, Dilip Kandlur, Dimitrios Pendarakis, and Debanjan Saha. Key management for secure Internet multicast using boolean function minimization techniques. In *Proceedings of IEEE INFOCOM '99*, volume 2, pages 689–698, March 1999.

[13] Yang-hua Chu, Sanjay G. Rao, and Hui Zhang. A case for end system multicast. In *Proceedings of ACM SIGMETRICS 2000*, pages 1–12, Santa Clara, CA, June 2000.

[14] Stephen E. Deering. Multicast routing in internetworks and extended LANs. In *Proceedings of ACM SIGCOMM '88*, August 1988.

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

[16] Sally Floyd, Mark Handley, Jitendra Padhye, and Jörg Widmer. Equation-based congestion control for unicast applications. In *Proceedings of ACM SIGCOMM 2000*, pages 43–56, August 2000.

[17] Mohamed G. Gouda, Chin-Tser Huang, and E. N. Elnozahy. Key trees and the security of interval multicast. In *Proceedings of the IEEE International Conference on Distributed Computing Systems*, Vienna, Austria, 2002.

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

[19] Internet Research Task Force (IRTF). Reliable Multicast Transport (rmt) Charter. http://www.ietf.org/html.charters/rmt-charter.html.

[20] Internet Research Task Force (IRTF). The secure multicast research group (SMuG). http://www.securemulticast.org/.

[21] Sneha K. Kasera, Jim Kurose, and Don Towsley. A comparison of server-based and receiver-based local recovery approaches for scalable reliable multicast. In *Proceedings of IEEE INFOCOM '98*, pages 988–995, San Francisco, CA, March 1998.

[22] Roger G. Kermode. Scoped Hybrid Automatic Repeat reQuest with Forward Error Correction (SHARQFEC). In *Proceedings of ACM SIGCOMM '98*, pages 278–289, September 1998.

[23] Minseok Kwon and Sonia Fahmy. Topology-aware overlay networks for group communication. In *Proceedings of ACM NOSSDAV*, pages 127–136, Miami, Florida, USA, May 2002.

[24] Simon S. Lam and Huaiyu Liu. Silk: A resilient routing fabric for peer-to-peer networks. Technical Report TR–03–13, Department of Computer Sciences, The University of Texas at Austin, October 2003.

[25] Simon S. Lam and Huaiyu Liu. Failure recovery for structured p2p networks: Protocol design and performance evaluation. In *Proceedings ACM SIGMETRICS 2004*, New York, NY, June 2004.

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

[27] X. Steve Li, Y. Richard Yang, Mohamed G. Gouda, and Simon S. Lam. Batch rekeying for secure group communications. In *Proceedings of Tenth International World Wide Web Conference (WWW10)*, pages 525–534, Hong Kong, China, May 2001.

[28] Huaiyu Liu and Simon S. Lam. Neighbor table construction and update in a dynamic peer-to-peer network. In *Proceedings of IEEE ICDCS 2003*, Providence, RI, May 2003.

[29] Philip K. McKinley and Arun P. Mani. An experimental study of adaptive forward error correction for wireless collaborative computing. In *Proceedings of the 2001 IEEE Symposium on Applications and the Internet (SAINT-2001)*, pages 157–166, San Diego, CA, January 2001.

[30] Dalit Naor, Moni Naor, and Jeff Lotspiech. Revocation and tracing schemes for stateless receivers. *Lecture Notes in Computer Science (Crypto 2001)*, 2139:41–62, 2001.

[31] T. S. Eugene Ng and Hui Zhang. Predicting internet network distance with coordinates-based approaches. In *Proceedings of IEEE INFOCOM 2002*, New York, NY, June 2002.

[32] J. Nonnenmacher, E. Biersack, and D. Towsley. Parity-based loss recovery for reliable multicast transmission. In *Proceedings of ACM SIGCOMM '97*, pages 289–300, September 1997.

[33] J. Nonnenmacher, M. Lacher, M. Jung, E.W. Biersack, and G. Carle. How bad is reliable multicast without local recovery? In *Proceedings of IEEE INFOCOM '98*, pages 972–979, San Francisco, CA, March 1998.

[34] The Network Simulator ns 2. http://www.isi.edu/nsnam/ns/.

[35] S. Paul, K. Sabnani, and D. Kristol. Multicast transport protocols for high speed networks. In *Proceedings of IEEE ICNP '94*, pages 4–14, Boston, MA, October 1994.

[36] Adrian Perrig, Dawn Song, and Doug Tygar. Elk, a new protocol for efficient large-group key distribution. In *Proceedings of IEEE Symposium on Security and Privacy*, pages 247–262, May 2001.

[37] C. Greg Plaxton, Rajmohan Rajaraman, and Andréa W. Richa. Accessing nearby copies of replicated objects in a distributed environment. *Theory of Computing Systems*, pages 241–280, 1999.

[38] Sylvia Ratnasamy, Mark Handley, Richard Karp, and Scott Shenker. Application-level multicast using content-addressable networks. In *Proceedings of NGC 2001*, November 2001.

[39] Sylvia Ratnasamy, Mark Handley, Richard Karp, and Scott Shenker. Topologically-aware overlay construction and server selection. In *Proceedings of IEEE INFOCOM 2002*, June 2002.

[40] Luigi Rizzo. Effective erasure codes for reliable computer communication protocols. *Computer Communication Review*, 27(2):24–36, April 1997.

[41] Luigi Rizzo. pgmcc: A tcp-friendly single-rate multicast congestion control scheme. In *Proceedings of ACM SIGCOMM 2000*, pages 17–28, Stockholm, Sweden, August 2000.

[42] Antony Rowstron and Peter Druschel. Pastry: Scalable, decentralized object location and routing for large-scale peer-to-peer systems. In *Proceedings of 18th IFIP/ACM International Conference on Distributed Systems Platforms (Middleware 2001)*, pages 329–350, Heidelberg, Germany, November 2001.

[43] Antony Rowstron, Anne-Marie Kermarrec, Miguel Castro, and Peter Druschel. SCRIBE: The design of a large-scale event notification infrastructure. In *Proceedings of NGC 2001*, pages 30–43, London, UK, November 2001.

[44] D. Rubenstein, J. Kurose, and D. Towsley. Real-time reliable multicast using proactive forward error correction. In *Proceedings of NOSSDAV '98*, pages 279–293, July 1998.

[45] Sanjeev Setia, Samir Koussih, Sushil Jajodia, and Eric Harder. Kronos: A scalable group re-keying approach for secure multicast. In *Proceedings of IEEE Symposium on Security and Privacy*, pages 215–228, Berkeley, CA, May 2000.

[46] Sanjeev Setia, Sencun Zhu, and Sushil Jajodia. A comparative performance analysis of reliable group rekey transport protocols for secure

multicast. In *Performance Evaluation, special issue on the Proceedings of Performance 2002*, volume 49, pages 21–41, Rome, Italy, September 2002.

[47] Jack Snoeyink, Subhash Suri, and George Varghese. A lower bound for multicast key distribution. In *Proceedings of IEEE INFOCOM 2001*, pages 422–431, Anchorage, Alaska, April 2001.

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

[49] D. Waitzman, C. Partridge, and S. Deering. *Distance Vector Multicast Routing Protocol, RFC 1075*, November 1988.

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

[51] Jorg Widmer and Mark Handley. Extending equation-based congestion control to multicast applications. In *Proceedings of ACM SIGCOMM 2001*, pages 275–285, San Diego, CA, August 2001.

[52] Chung Kei Wong, Mohamed G. Gouda, and Simon S. Lam. Secure group communications using key graphs. In *Proceedings of ACM SIGCOMM '98*, pages 68–79, September 1998.

[53] Chung Kei Wong and Simon S. Lam. Keystone: a group key management system. In *Proceedings of International Conference on Telecommunications*, Acapulco, Mexico, May 2000.

[54] Y. Richard Yang and Simon S. Lam. General AIMD congestion control. In *Proceedings of the 8th International Conference on Network Protocols '00*, pages 187–198, Osaka, Japan, November 2000.

[55] Y. Richard Yang, X. Steve Li, X. Brian Zhang, and Simon S. Lam. Reliable group rekeying: A performance analysis. In *Proceedings of ACM SIGCOMM 2001*, pages 27–38, San Diego, CA, August 2001.

[56] Jaehee Yoon, Azer Bestavros, and Ibrahim Matta. Adaptive reliable multicast. In *International Conference on Communications (ICC)*, volume 3, pages 1542–1546, New Orleans, LA, June 2000.

[57] X. Brian Zhang, Simon S. Lam, and Dong-Young Lee. Group rekeying with limited unicast recovery. In *IEEE ICC 2003 Symposium on Next Generation Internet*, volume 1, pages 707–714, Anchorage, Alaska, May 2003.

[58] X. Brian Zhang, Simon S. Lam, and Dong-Young Lee. Group rekeying with limited unicast recovery. *Computer Networks*, 44(6):855–870, April 2004.

[59] X. Brian Zhang, Simon S. Lam, Dong-Young Lee, and Y. Richard Yang. Protocol design for scalable and reliable group rekeying. In *Proceedings*

of SPIE Conference on Scalability and Traffic Control in IP Networks, volume 4526, pages 87–108, Denver, CO, August 2001.

[60] X. Brian Zhang, Simon S. Lam, Dong-Young Lee, and Y. Richard Yang. Protocol design for scalable and reliable group rekeying. Technical Report TR–02–29, Department of Computer Sciences, The University of Texas at Austin, June 2002 (revised, November 2002).

[61] X. Brian Zhang, Simon S. Lam, Dong-Young Lee, and Y. Richard Yang. Protocol design for scalable and reliable group rekeying. *IEEE/ACM Transactions on Networking*, 11(6):908–922, December 2003.

[62] X. Brian Zhang, Simon S. Lam, and Huaiyu Liu. Efficient group rekeying using application-layer multicast. In *25th IEEE International Conference on Distributed Computing Systems (ICDCS 2005)*, Columbus, Ohio, June 2005.

[63] Ben Y. Zhao, John Kubiatowicz, and Anthony D. Joseph. Tapestry: An infrastructure for fault-tolerant wide-area location and routing. *IEEE Journal on Selected Areas in Communications*, 22(1):41–53, January 2004.

[64] Shelley Q. Zhuang, Ben Y. Zhao, Anthony D. Joseph, Randy H. Katz, and John D. Kubiatowicz. Bayeux: An architecture for scalable and fault-tolerant widearea data dissemination. In *Proceedings of NOSSDAV 2001*, June 2001.

Vita

Xincheng Zhang was born in Hefei, Anhui, China. After completing his work at the No. 1 high school, Hefei, Anhui, he entered Tsinghua University, Beijing. He received the degree of Bachelor of Engineering in Computer Science and Technology from Tsinghua University in 1996. In August 1999, he entered the Graduate School of the University of Texas at Austin. In August 2004, he received the degree of Master of Science in Computer Sciences from the University of Texas of Austin.

Permanent address: 22 JinZhai Street, Bldg 8, Room 405
Hefei, Anhui Province, P.R. China 230022

This dissertation was typeset with LaTeX[†] by Xincheng Zhang.

[†]LaTeX is a document preparation system developed by Leslie Lamport as a special version of Donald Knuth's TeX Program.