Efficient Group Rekeying Using Application-Layer Multicast [*]

X. Brian Zhang, Simon S. Lam, and Huaiyu Liu
Department of Computer Sciences, The University of Texas at Austin, Austin, TX 78712
{zxc, lam, huaiyu}@cs.utexas.edu

Abstract

In secure group communications, there are both rekey and data traffic. 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 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.

1. Introduction

Many emerging Internet applications, such as grid computing, teleconferences, pay-per-view, multi-party games, and distributed interactive simulations will benefit from using a secure group communications model [8]. 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. Each user is an end host. 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*).

There have been extensive research results on the design of group key management in recent years [21, 22, 6, 2, 24, 26]. In particular, the key tree approach [21, 22] 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 [20].

To further reduce server processing and bandwidth overheads, periodic batch rekeying was proposed [18, 24, 12, 26]. In batch rekeying, the key server processes the join and leave requests during a rekey interval as a batch, and generates a single rekey message at the end of the rekey interval. The rekey message is then sent to all users immediately, and it requires fast delivery to achieve tight group access control. As a result, *rekey traffic is bursty*.

Existing rekey transport protocols [24, 3, 19, 26, 25] are based on IP multicast, which has not been widely deployed. In this paper, we propose to use application-layer multicast (ALM) to support concurrent rekey and data transport. To our best knowledge, this paper is the first attempt on how to efficiently support both rekey and data transport over ALM.

Using ALM to support both rekey and data transport 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 many downstream users. Therefore, it is desired to reduce rekey bandwidth overhead as much as possible.

Using ALM to support group rekeying also offers new opportunities to do naming and routing. In our approach, each user in the group is assigned a unique ID that is a string of D digits. All 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 [14, 16, 28, 13, 10, 11]. 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 to exploit proximity in the underlying network. By virtue of this scheme, each multicast tree embedded in the neighbor tables tends to be topology-aware. That is, users in the same multicast subtree

[*] Research sponsored in part by NSF grant ANI-0319168.

tend to be in the same topological region. As a result, when a message is forwarded from its multicast source towards a user during multicast, it tends to be always forwarded 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 in each rekey interval, each user needs only a small subset of encrypted new keys (*encryptions*, in short) generated by the key server [24, 26]. 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. We then propose an identification scheme 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. We further propose a message splitting scheme 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 [7, 4, 29, 17, 15, 9]. If we use an existing ALM scheme to replace our multicast scheme, however, it incurs a large maintenance cost at users, and the efficiency of the splitting scheme would be reduced. In our approach, each user does not need to maintain states for its downstream users to perform rekey message splitting. We defer a detailed discussion of this issue to Section 2.6.

We formulate and prove correctness properties for the multicast scheme and rekey message splitting scheme. We conducted extensive simulations to evaluate our 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 rekey bandwidth overhead from several thousand encryptions to less than ten encryptions.

The rest of this paper is organized as follows. In Section 2, we describe our system design. In Section 3, we present the protocol for each joining user to determine its ID, and discuss user joins, leaves, and failure recovery. We evaluate our approach through simulations in Section 4, and conclude in Section 5.

2. 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 3. Please see our technical report [27] for proofs of the lemmas and theorems. Notation used in this paper is defined in Table 1.

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	total number of users in a group
P	a joining user collects P users from (i, j)-ID subtree
R_i	RTT thresholds, $i = 1, 2, ..., D - 1$
$u.ID$	user u's ID
$u.ID[i]$	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 $i < 0$

2.1. ID tree

Each user 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 paper. All the 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 1 *Given a group of users, the corresponding **ID tree** is defined as follows:*

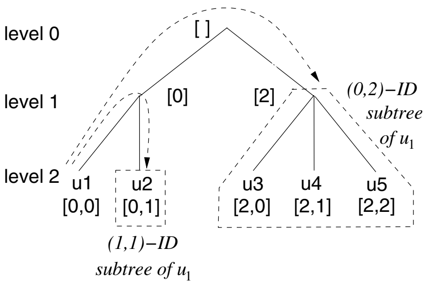
- *At level 0, there is a single node, the tree root, whose ID is a null string, denoted by "[ ]".*
- *At level i, $1 \leq i \leq D$, each node has a unique ID that is a string of i digits. A node with ID x exists at level i if there exists a user u in the group such that x is a prefix of $u.ID$. The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

Fig. 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.

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 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 to multiple child level-$(i+1)$ ID subtrees of the level-i ID subtree, such that all 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 3.1, 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.

2.2. Neighbor tables

Each user in the group maintains a neighbor table. Similar neighbor tables were used to support hypercube routing [14, 16, 28, 13, 10, 11].

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**, $0 \le i \le D-1$ and $0 \le j \le B-1$. 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, 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 the neighbors in the same entry are arranged in increasing order of their RTTs.

Definition 3 *Given a group of users, each with a unique ID of D digits, their neighbor tables are said to be* K**-consistent**, *$K \ge 1$, if for any user u in the group, each (i, j)-entry, $0 \le i \le D-1$ and $0 \le j \le B-1$, in its neighbor table satisfies the following conditions:*

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

(2) If $j \ne u.ID[i]$, 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.

```
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.
$level \leftarrow msg$.forward_level
if $level = D$ then return
if the caller is the key server then        ▷ $level = 0$ in this case
  $msg$.forward_level $\leftarrow level + 1$
  send a copy of $msg$ to each $(0, j)$-primary neighbor, $0 \le j < B$
else for $i \leftarrow level$ to $D - 1$ do
  $msg$.forward_level $\leftarrow i + 1$
  send a copy of $msg$ to each $(i, j)$-primary neighbor, $0 \le j < B$
```

The concept of K-consistency was proposed in [11, 10]. K-consistency implies 1-consistency. If all the users in the group maintain 1-consistent neighbor tables, then a message is guaranteed to reach every user via multicast, as proved in Section 2.3. It is desired to let $K > 1$ for resilience [11, 10].

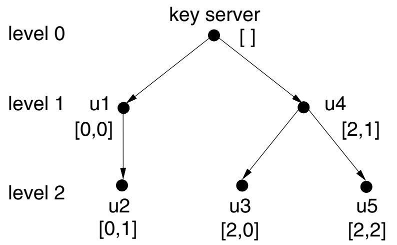
The key server also maintains a neighbor table, which has a single row. The row contains B entries, each referred to as $(0, j)$-entry, $j = 0, 1, ..., B-1$. Among all the users whose IDs have the prefix "[j]", the key server chooses the K (or all, if the total number of such users is less than K) users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

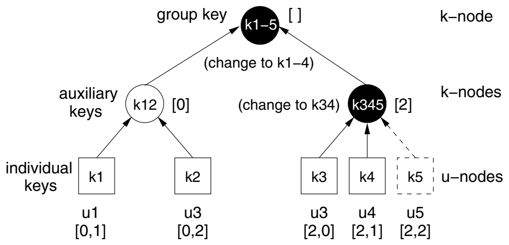
2.3. Multicast scheme: T-mesh

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 the 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 the 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 1.

Definition 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 routine FORWARD specified in Fig. 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.

Fig. 3 illustrates an example rekey multicast tree for the group of five users defined in Fig. 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.

Lemma 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 its downstream users have the common prefix $u.ID[0 : i-1]$. Furthermore, u and its downstream users belong to the same level-i ID subtree.*

Recall that all 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 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 1 and 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 $\texttt{forward\_level} = i$ 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 1 *In a multicast session, assume every user in the group has 1-consistent neighbor table. Then following the multicast scheme specified in Fig. 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 user needs to look for more neighbors to replace the failed or congested one.

2.4. Modified key tree

The key server maintains a key tree. To support efficient rekey message splitting, the key tree used in this paper is different from the original approach [21, 22, 24, 26]. 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. A key tree contains two types of nodes: *u-nodes* and *k-nodes*. Each u-node corresponds to a particular user, and it contains the user's individual key. A user shares its individual key only with the key server. A k-node contains the group key or an auxiliary key. A user in the group is given the individual key contained in its corresponding u-node as well as the keys contained in the k-nodes on the path from its corresponding u-node to the root.

To facilitate rekey message splitting, the key server makes the structure of the key tree match exactly that of the ID tree. Each u-node in the key tree that contains user u's individual key corresponds to the leaf node in the ID tree whose ID equals $u.ID$. Fig. 4 shows the key tree that corresponds to the example ID tree shown in Fig. 1.

At the end of each rekey interval, the key server changes all the keys on the path from each newly joined or departed u-node to the root (see technical report [27] for a detailed description). Then 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. Each encrypted new key is referred to as

```
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, $0 \le j < B$.
▷ $s$: it equals 0 if the caller is the key server; otherwise, we have
    $msg.\texttt{forward\_level} \le s < D$.
$msg' \leftarrow$ an empty message with $\texttt{forward\_level} = s + 1$
for each encryption $e$ contained in $msg$ do
    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
        copy $e$ into $msg'$
send $msg'$ to $w_{s,j}$ via unicast
```

an **encryption**. For example, suppose u_5 leaves the group in Fig. 4. 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}. Then the key server generates four encryptions: $\{k_{1-4}\}_{k_{12}}$, $\{k_{1-4}\}_{k_{34}}$, $\{k_{34}\}_{k_3}$, and $\{k_{34}\}_{k_4}$. Here $\{k'\}_k$ denotes key k' encrypted by key k. All the encryptions are put in a single rekey message. Each user needs only a small subset of encryptions in the rekey message. For example, u_1 needs only $\{k_{1-4}\}_{k_{12}}$.

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 3 *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.*

2.5. Rekey message splitting scheme

To send new keys to users after rekeying, a straightforward approach is to mulitcast all the 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 the downstream users. Therefore, it is desired to reduce rekey bandwidth overhead as much as possible.

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 purpose, the key server composes a separate message for each $(0, j)$-primary neighbor by executing routine REKEY-MESSAGE-SPLIT specified in Fig. 5, $j = 0, 1, ..., B-1$. Each user at forwarding level i, $0 \le i \le D-1$, 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 Fig. 5 is called at lines 5 and 9 of routine FORWARD specified in Fig. 2. The correctness of the rekey message splitting scheme is stated below.

Theorem 2 *In a multicast session for rekey transport, suppose that member u is at forwarding level i, $0 \le i \le D-1$. 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 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 $e.ID$.*

Corollary 1 *In a multicast session for rekey transport, assume every user in the group has 1-consistent neighbor table. Following the multicast scheme and the rekey message splitting scheme specified in Figs. 2 and 5, respectively, for any user u in the group and any encryption e that is generated by the key server, 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.*

2.6. 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 [7, 4, 29, 17, 15, 9] to replace T-mesh, or use the original key tree [21, 22, 24, 26] to replace the modified key, 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 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 modified key tree and the ID tree, all 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 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. Protocol description

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

3.1. User ID assignment

To join a group, a user, say u, first contacts the key server (or a separate registrar server [23]) and mutually authenticates each other using a protocol such as SSL. If authenticated and accepted into the group, u receives its individual key and the current group key. From now on, all the communications between u and the key server are encrypted with the individual key, and all the communications between u and other users in the group are encrypted with the group key.[1]

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 the keys on the path from u's corresponding u-node to the 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, $0 \leq i \leq D-2$, u's actions consist of four steps. (We assume i is fixed in the following discussion and in Sections 3.1.1, 3.1.2, and 3.1.3.)

In the first step, u collects the records of users who belong to its $(i+1, j)$-ID subtree (see Definition 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 already determines 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 $(i+1, b)$-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 $(i+1, b)$-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.

3.1.1. Step 1: collecting user records

For u to know which users belong to its $(i+1, j)$-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 $i = 0$, u sends a query to the user whose record is provided to u by the key server. For $i > 0$, since u already determines the first i digits of its ID so far, it knows at least one user that belongs to u's $(i, 0)$-ID subtree to $(i, B-1)$-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 the neighbors whose IDs have the target ID prefix. In this way, u collects one or more users from its $(i+1, j)$-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 $(i+1, j)$-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 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 the simulations in this paper.

3.1.2. Step 2: measuring RTTs

In this step, u estimates whether it is close to the users it collected from its $(i+1, j)$-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. 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

[1] 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.

end hosts tend to be topologically close to each other even if their access links have long latency.[2]

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.

3.1.3. 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 the users it collected from its $(i + 1, j)$-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 the simulations in this paper. Suppose the RTTs of the users that u collected from its $(i + 1, b)$-ID subtree, $0 \leq b \leq B - 1$, produces the smallest F-percentile value, denoted by $f_{i+1,b}$. User u then compares $f_{i+1,b}$ with the delay threshold R_{i+1}, and the comparison results in two cases.

In the first case, $f_{i+1,b}$ is less than or equal to R_{i+1}. User u then predicts that it is topologically close to the users belonging to its $(i + 1, b)$-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, $f_{i+1,b}$ is larger than R_{i+1}. User u then predicts that it is not close enough to the users in any $(i + 1, j)$-ID subtree, $j = 0, 1, ..., B - 1$. In this case, u goes to step 4 and asks the key server to assign the ith digit to the last digit of its ID, that is, $u.ID[i]$, $u.ID[i+1]$, ..., and $u.ID[D - 1]$.

3.1.4. 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.[3] Consequently, in the ID tree, u becomes a user in a new 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 the keys on the path from u's corresponding u-node to the root in the key tree.

Our technical report [27] presents the analysis of the communication cost for a joining user to determine its ID. The main result is that if each non-leaf node in the ID tree has the same outgoing degree, then the cost is $O(P \cdot \ln N)$ when $D = \ln N$.

3.2. Join, leave, and failure recovery

After its ID is determined, u needs to build its neighbor table.[4] 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 [13, 10] 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 presented in Silk is used to accomplish this task. After that, 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 [11] for sophisticated failure recovery strategies.

4. 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 are 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.[5] In our simulator, we let each member (a user or

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

[3] In an extreme case, the key server may not be able to find a unique value for $u.ID[0 : l]$. See technical report [27] for a solution.

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

[5] We also used the minimum value of 20 RTT samples measured for each pair of PlanetLab hosts, and repeated each simulation presented in Section 4.1. The relative performance of T-mesh and NICE (the multicast

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 [5]. 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. See technical report [27] for the parameter settings of the topology.

In the simulations, we compare the performance of T-mesh with NICE [4], one of the state-of-the-art ALM schemes.[6] 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 the simulations (except the ones in Section 4.2) 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 the simulations of T-mesh, we set $D = 5$, $R_1 = 150$ ms, $R_2 = 30$ ms, $R_3 = 9$ ms, $R_4 = 3$ ms, $B = 256$, and $K = 4$, unless otherwise stated. In all the simulations of NICE, each cluster contains three to eight users [4].

4.1. Delivery latency

We evaluate the delivery latency of a rekey message when the key server multicasts it 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 milliseconds) – 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.

Note that there is no notation of the key server in NICE. To multicast a rekey message in NICE, we let the key server unicast the message to the root of the NICE tree, which is the topological center of all the users in the group [4]. The message then traverses the tree in a top-down fashion.

We ran simulations on the PlanetLab topology with 226 user joins. Each user joins the group at a random time between 0 and 452 seconds. After all the joins terminate, the key server multicasts a message.

Fig. 6 plots the inverse cumulative distribution of user stress, application-layer delay, and RDP. Each point with coordinates (x, y) in Fig. 6 (a) can be interpreted as: x fraction of users have an average user stress less than or equal to y. Each average value is computed based on 100 runs. For each run, 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, we computed the average user stress of the users with this particular rank across all runs. The vertical bars show the 5 to 95-percentile values. Figs. 6 (b) and (c) can be interpreted similarly.

From Fig. 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 Fig. 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 1024 user joins, respectively. We also conducted simulations on both the PlanetLab and GT-ITM topologies to evaluate delivery latency of a data message when it is multicast by a user. The simulation results (see technical report [27]) showed that in these simulations the relative performance of T-mesh and NICE is similar to that in Fig. 6.

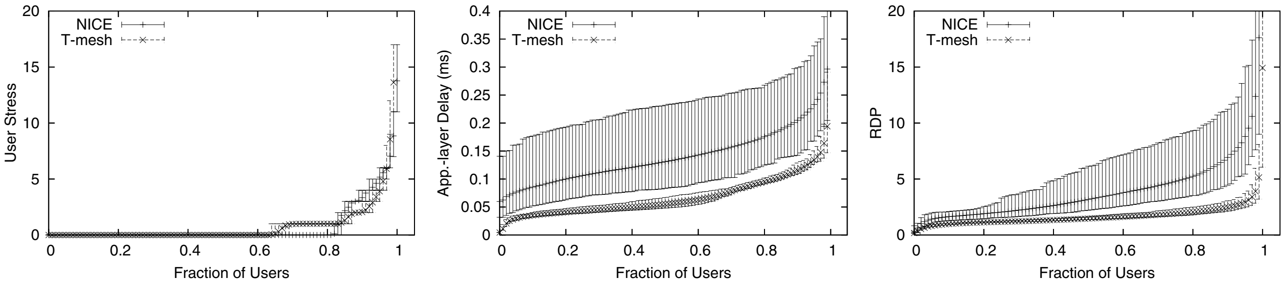
Note that it is not appropriate to conclude that T-mesh is better than NICE for data transport in general. NICE is designed for scalable group communications, and has no notion of the key server. In NICE, to determine its position in the tree, each joining user probes smaller number of users than a joining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

4.2. 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 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 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

scheme for comparison) does not change.

[6] We did not choose Narada [7] 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.



(a) Inverse cumulative distribution of user stress.

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

(c) Inverse cumulative distribution of RDP.

controller to simulate the J joins and L leaves in that rekey interval.

Fig. 7 (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 runs. Fig. 7 (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 [22] with degree 4 and the batch rekeying algorithm proposed in [26]. A degree of 4 is proved to be optimal in terms of rekey cost per join or leave [22].

We propose a cluster rekeying heuristic to reduce the rekey cost of the modified key tree. In the heuristic, all 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 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. Please see technical report [27] for a detailed description of the heuristic.

Fig. 7 (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. Rekey bandwidth overhead

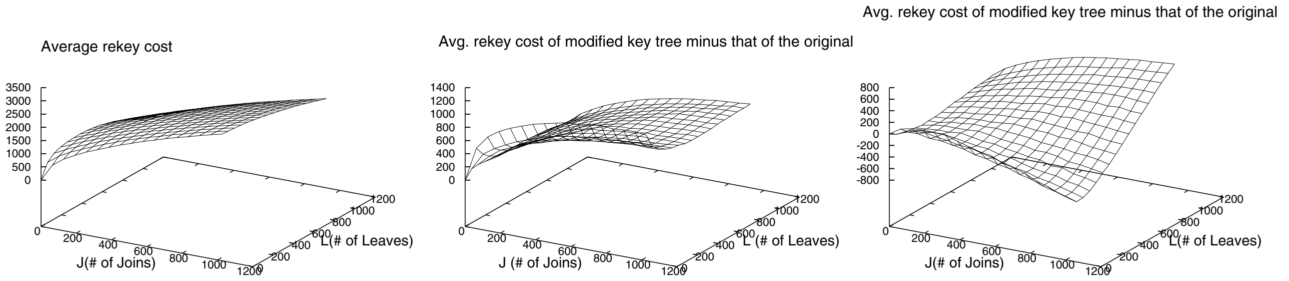
We now evaluate whether the rekey message splitting scheme can significantly reduce rekey bandwidth overhead. We use the GT-ITM topology for all the simulations in this subsection. In each simulation, 1024 users join the group each at a random time between 0 and 2048 seconds. After all the joins terminate, 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.

protocol	*key tree approach*	*multicast scheme*	*cluster rekeying*	*rekey msg 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

For comparison, we define seven rekey transport protocols, as specified in Table 2. The IP multicast scheme used in P_4 is based on the DVMRP multicast routing algorithm. As pointed out in Section 2.6, 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.

Figs. 8 (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 a typical 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.

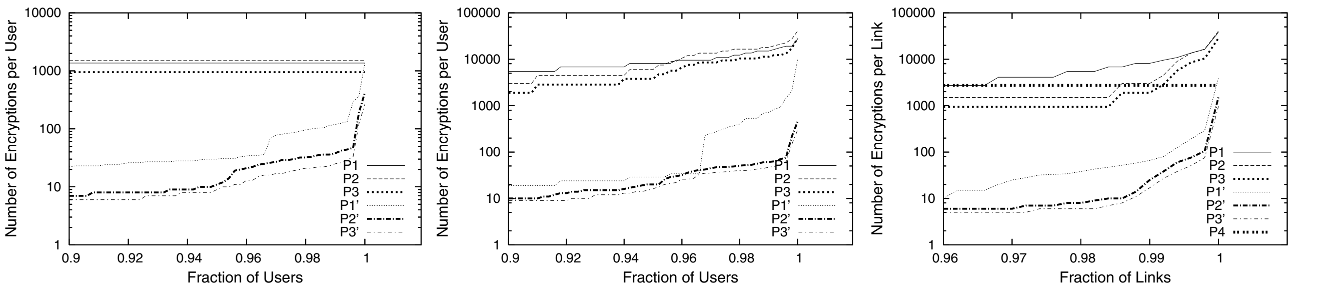
In Fig. 8, by comparing P_1' to P_1, P_2' to P_2, and P_3' to P_3, we observe that rekey message splitting is very effective in reducing rekey bandwidth overhead. In particular, in P_2' and P_3' (using T-mesh), the rekey message splitting can reduce rekey bandwidth overhead for more than 90% of users 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 Figs. 8 (a) and (b)). And only a few links receive up to 1500 encryptions (see Fig. 8 (c)). These links are on the paths from the key server to its $(0, j)$-



(a) Rekey cost of the modified key tree.

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

(c) Rekey cost after applying the cluster rekeying heuristic to the modified key tree.



(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.

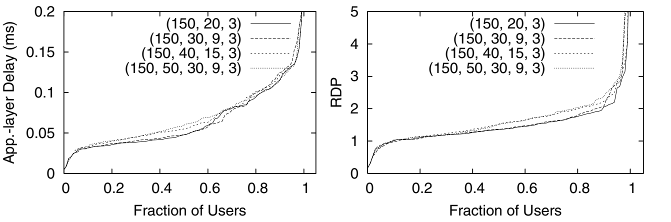
primary neighbors, $j = 0, 1, ..., B-1$. Since rekey transport and data transport choose different multicast trees in T-mesh, we expect that in P_2' and P_3' rekey transport does not affect data transport as long as the rekey bandwidth overhead at most users and most links is very small.

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 Figs. 8 (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 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 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.

4.4. 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



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

(b) Inverse cumulative distribution of RDP.

D

$(R_1, R_2, ..., R_{D-1})$

set R_1 around one hundred milliseconds so that all the 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 the 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.

Fig. 9 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 a typical run.

From the figure, we observe that the latency performance of T-mesh is not sensitive to the various values of delay thresholds that we chose.

5. Conclusion

In this paper, 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 the users (and links) than a representative existing ALM scheme.

References

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

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

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

[4] S. Banerjee, B. Bhattacharjee, and C. Kommareddy. Scalable application layer multicast. In *Proceedings of ACM SIGCOMM 2002*, pages 205–217, Pittsburgh, PA, Aug. 2002.

[5] K. Calvert, M. Doar, and E. W. Zegura. Modeling Internet topology. *IEEE Communications Magazine*, 35(6):160–163, June 1997.

[6] I. Chang, R. Engel, D. Kandlur, D. Pendarakis, and D. Saha. Key management for secure Internet multicast using boolean function minimization techniques. In *Proceedings of IEEE INFOCOM '99*, volume 2, pages 689–698, Mar. 1999.

[7] Y. Chu, S. G. Rao, and H. Zhang. A case for end system multicast. In *Proceedings of ACM SIGMETRICS 2000*, pages 1–12, Santa Clara, CA, June 2000.

[8] I. R. T. F. (IRTF). The secure multicast research group (SMuG). http://www.securemulticast.org/.

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

[10] S. S. Lam and H. 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, Oct. 2003.

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

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

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

[14] C. G. Plaxton, R. Rajaraman, and A. W. Richa. Accessing nearby copies of replicated objects in a distributed environment. *Theory of Computing Systems*, pages 241–280, 1999.

[15] S. Ratnasamy, M. Handley, R. Karp, and S. Shenker. Application-level multicast using content-addressable networks. In *Proceedings of NGC 2001*, Nov. 2001.

[16] A. Rowstron and P. 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, Nov. 2001.

[17] A. Rowstron, A.-M. Kermarrec, M. Castro, and P. Druschel. SCRIBE: The design of a large-scale event notification infrastructure. In *Proceedings of NGC 2001*, pages 30–43, London, UK, Nov. 2001.

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

[19] S. Setia, S. Zhu, and S. 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, Sept. 2002.

[20] J. Snoeyink, S. Suri, and G. Varghese. A lower bound for multicast key distribution. In *Proceedings of IEEE INFOCOM 2001*, pages 422–431, Anchorage, Alaska, Apr. 2001.

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

[22] C. K. Wong, M. G. Gouda, and S. S. Lam. Secure group communications using key graphs. In *Proceedings of ACM SIGCOMM '98*, pages 68–79, Sept. 1998.

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

[24] Y. R. Yang, X. S. Li, X. B. Zhang, and S. S. Lam. Reliable group rekeying: A performance analysis. In *Proceedings of ACM SIGCOMM 2001*, pages 27–38, San Diego, CA, Aug. 2001.

[25] X. B. Zhang, S. S. Lam, and D.-Y. Lee. Group rekeying with limited unicast recovery. *Computer Networks*, 44(6):855–870, Apr. 2004.

[26] X. B. Zhang, S. S. Lam, D.-Y. Lee, and Y. R. Yang. Protocol design for scalable and reliable group rekeying. *IEEE/ACM Transactions on Networking*, 11(6):908–922, Dec. 2003.

[27] X. B. Zhang, S. S. Lam, and H. Liu. Efficient group rekeying using application-layer multicast. Technical Report TR–04–42, Department of Computer Sciences, The University of Texas at Austin, Oct. 2004.

[28] B. Y. Zhao, J. Kubiatowicz, and A. D. Joseph. Tapestry: An infrastructure for fault-tolerant wide-area location and routing. *IEEE Journal on Selected Areas in Communications*, 22(1):41–53, Jan. 2004.

[29] S. Q. Zhuang, B. Y. Zhao, A. D. Joseph, R. H. Katz, and J. D. Kubiatowicz. Bayeux: An architecture for scalable and fault-tolerant widearea data dissemination. In *Proceedings of NOSSDAV 2001*, June 2001.