Optimized Group Rekey for Group Communication systems

Ohad Rodeh, Ken Birman, Danny Dolev *

July 19, 1999

Abstract

In this paper we describe an efficient algorithm for the management of group-keys. Our algorithm is based on a protocol for secure IP-multicast and is used to manage group-keys in group-communication systems. Unlike prior work, based on centralized key-servers, our solution is completely distributed and fault-tolerant and its performance is comparable to the centralized solution.

1 Introduction

Increasingly many applications require multicast services, for example, teleconferencing, distributed interactive simulation, collaborative work. To protect multicast message content, such applications require secure multicast.

A multicast group can be efficiently protected using a single symmetric encryption key. This key is securely communicated to all group members which subsequentally use it to encrypt/decrypt group messages. The group-key is securely switched whenever the group membership changes, thereby preventing old members from eavesdropping on current group conversations. The challenge is to create an efficient and fast key-switch algorithm that can handle large groups and a high rate of membership changes.

The general case of a multicast group includes scenarios where the group size is very large, up to thousands or millions of members, and where there are few senders and many receivers. It also includes a more specific case where there is symmetry between group members. By this we mean that any member may be a source of multicast messages as well as a recipient. This work focuses on the symmetric case.

IP-multicast is a widespread low-level multicast primitive. IP-multicast security has been extensively discussed in the literature [1, 5, 3, 2, 8, 11] and efficient solutions have been proposed to secure it. These protocols all use centralized servers for key dissemination. The servers are single points of failure. Our solution uses a Group Communication System (GCS), it is completely distributed and fault-tolerant. It achieves low latency in the case of member join/leave, and its performance is on par with the centralized solution. Our solution's drawback is limited scalability, up to the scalability of the GCS. Work is in progress to alleviate this limitation.

Our solution is related to other group-key architectures developed for GCSs. These works begin with seminal research by Reiter [6, 9] and Gong [4], continuing to recent research on Ensemble [7].

*The authors were supported in part by the Israeli Ministry of Science grant number 032-7892.

These results show how group keying can be integrated with a Group Membership Protocol (GMP) to support such functions as securely managing keys at the group members, securely rekeying, supporting *secure channels* between members (discussed below), signing messages and encrypting the data segments of messages.

These real-world solutions, however, are limited by the scalability not just of the GMP itself (in its underlying insecure mode), but also of the keying architecture. Elsewhere we have studied GMP scalability and are arriving at some intriguing results, but there has been little attention by the practical community to scalability of GCS key architectures.

Here we report on the first half of an effort to integrate such a scalable keying architecture with the Ensemble system. This first step involves extending a powerful keying architecture to support high availability. Elsewhere, we plan to report on the second half, which will investigate systems issues and performance considerations arising when such a system is engineered for high performance and studied carefully under controlled test situations.

2 Model

We assume processes in the group can send and receive point-to-point and multicast messages, and have access to trusted authentication and authorization services. We also assume that the authentication service allows processes to open *secure channels*. A secure channel between a pair of processes allows the secure exchange of private information.

We use a GCS [10] to provide reliable communication and group membership services. All processes, inside a group, have knowledge of the set of currently live and accessible members. When some process crashes, or a network partition occurs, processes receive a *view notification* event, describing the current membership. This is also called a *view-change*. The GCS allows only trusted and authorized members into the group. Since all group members have the same view of the membership, we number them lexicographically from 1 to n. When we refer to the group leader, we implicitly refer to member number 1 (denoted m_1).

We use the notations:

$m_x \rightarrow m_{y,z} : M \equiv$ Member m_x sends message M to members m_y, m_z.

$\{X\}_{K_1,K_2} \equiv$ A tuple consisting of message X signed with key K_1, and X signed with key K_2. Note that such a tuple can be signed in a manner that would prevent intruders from tampering with either component of the pair.

Group Members: are denoted by $m_1 \ldots m_n$.

Subgroup keys: are denoted by K_S, where S is the subset of members

2.1 Liveness and Safety

Distributed protocols should be *live* and *safe*. In this section we define these properties formally, and discuss the manner in which our protocols satisfy them.

Liveness: We say that a protocol is live if when the network is stable[1] then the protocol eventually terminates.

[1]Network stability means that no link failures, nor process crashes occur, and all messages arrive on time.

Safety: We say that a protocol is safe if it does not reveal the group key to unauthorized members.

In the article body we describe protocols in a terse fashion. Here we describe the manner in which a protocol is actually executed, and how its liveness is ensured. A protocol is described as a series of send/multicast events between members, and as local computation steps. For example, protocol $\mathcal{P}$ in Figure 1 securely switches the group key.

---

1. The group leader chooses a new key.
2. The leader uses secure channels to send the key securely to the members.

Figure 1: Protocol $\mathcal{P}$

This protocol is safe since it uses secure channels to group members, all of which are trusted. However, as stated above, $\mathcal{P}$ is not live. Notice that the protocol requires all processes to receive the new-key. If some member fails and never recovers during the execution the protocol blocks. To make the protocol fault-tolerant we restart the protocol in case of a view change. Another problem we face is protocol termination. Before starting to use a key, participants need to know that all members of the view have received that key. That is, all group members should be notified that the protocol has terminated. We use a two phase protocol for this purpose:

1. Each group member, once it receives the group-key from the leader, sends an acknowledgment (in the clear) to the leader.
2. The leader, once it receives acknowledgments from all group members, multicasts a *ProtoDone* message.
3. A member that receives a *ProtoDone* message knows that the protocol has terminated and the new key can now be used.

We implicitly add these steps to all protocols thereby ensuring their liveness. This allows us to specify protocols in a more succinct fashion.

3 The centralized solution ($\mathcal{C}$)

Here we describe a protocol by Wong, Gouda and Lam [2]. A keygraph is defined as a directed tree where the leafs are the group members and the nodes are keys. A member knows all the keys on the way from itself to the root. The keys are distributed using a key-server. In Figure 2 we see a typical key-graph for a group of 8 members.

Each member m_i shares a key with the server, K_i, and also shares keys with subgroups in the tree. For example, member m_1 knows keys $K_1, K_{12}, K_{14}, K_{18}$. It shares K_1 with the server, K_{12} with member m_2, K_{14} with members m_2, m_3, m_4, and K_{18} with members $m_1, \ldots m_8$.

The tree is built by the key server, it initially has *secure channels* with each of the members. It uses these channels to create the higher level keys. For example, in order to create key K_{12} it

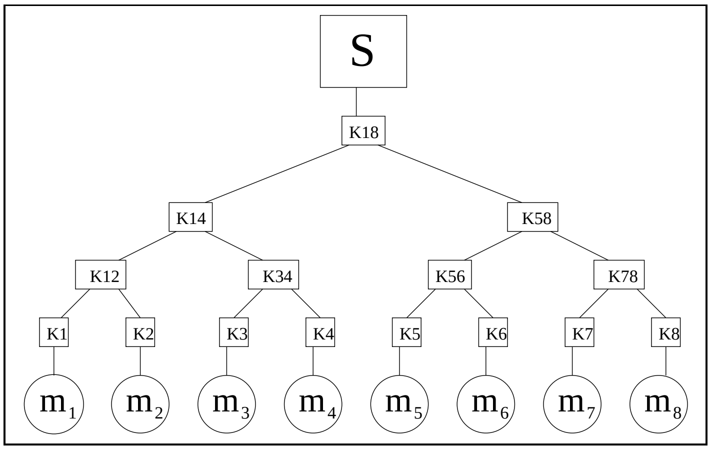

Figure 2: A keygraph for a group of 8 members.

encrypts K_{12} with keys K_1 and K_2, and sends $\{K_{12}\}_{K_1,K_2}$ to members m_1, m_2. Only members m_1 and m_2 will be able to decrypt this message and retrieve K_{12}. In the same manner keys K_{34}, K_{56}, K_{78} are established. To establish K_{14} the server chooses K_{14} encrypts it with K_{12} and K_{34} and sends $\{K_{14}\}_{K_{12},K_{34}}$ to members $m_1, \ldots m_4$. In similar manner K_{58} is established. Key K_{18} is then encrypted with K_{14}, K_{58} and multicast.

Figure 3 describes this through a time-line diagram. First, keys $K^2 = \{K_{12}, K_{34}, K_{56}, K_{78}\}$ are created. Then, keys $K^4 = \{K_{14}, K_{58}\}$ are created based on K^2. Finally, the group-key K_{18} is established using K^4.

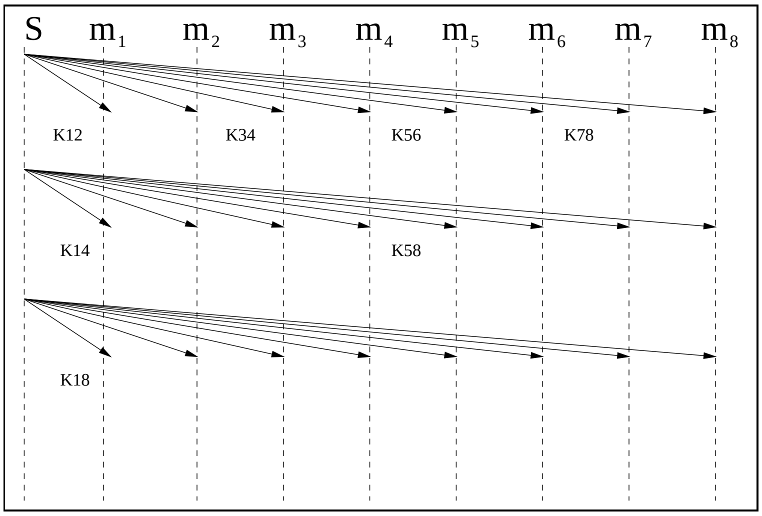

Figure 3: The creation of a keygraph on a time-line. Time flows from top to bottom. First, $K^2 = \{K_{12}, K_{34}, K_{56}, K_{78}\}$ is created, then $K^4 = \{K_{14}, K_{58}\}$, then the group-key.

The group key needs to be replaced if some member joins or leaves. This is performed through key-tree operations.

Join: Assume member m_9 joins the group. S picks a new (random) group-key K_{19} encrypts it with K_9, K_{18} and multicasts it to the group. Member m_9 uses K_9 to decrypt it, and the existing members use K_{18} to decrypt it.

Leave: Assume member m_1 leaves, then the server needs to replace keys K_{14}, and K_{18}. It chooses new key K_{24}, encrypts it with K_2, K_{34} and sends to m_2, m_3, m_4. It then chooses K_{28} and uses K_{24} and K_{58} to disseminate it.

In this scheme each member stores $log_2(n)$ keys, while the server keeps a total of n keys. The server uses n secure channels to communicate with the members. It is possible to create the full tree using a single multicast.

It is possible, and in fact more efficient, to use trees of degree larger than 2. Here and through the paper we discuss binary trees for simplicity. The analysis for trees of degree three or more is essentially the same for the centralized solution as well as our algorithm.

Trees become imbalanced after many additions and deletions, and it becomes necessary to rebalance them. We do not discuss this issue for brevity, but note only that well known tree-algorithms can be used for this purpose.

4 Our solution

The problem with the previous solution is that it is not fault-tolerant, and relies on a centralized server which has knowledge of all the keys. We desire a completely distributed solution. Our protocol uses no centralized server, and members play symmetric roles.

First we describe the basic protocol, denoted $\mathcal{B}$. In order to make protocol $\mathcal{B}$ completely distributed we use the notion of subtrees *agreeing* on a mutual key. Informally, this means that two groups of members, L and R, securely agree on a mutual encryption key. Assume that m_l is L's leader, m_r is R's leader, L has group key K_L, and R has group key K_R. The protocol used to agree on a mutual key is as follows:

1. m_l chooses a new key K_{LR}, and sends it to m_r using a secure channel.

2. m_l encrypts K_{LR} with K_L and multicasts it to L; m_r encrypts K_{LR} with K_R and multicasts to R.

3. All members of $L \cup R$ securely receive the new key.

Formally, *agree* is defined as a protocol by which two subtrees, L and R, possessing secret keys K_L and K_R respectively, choose a new key and securely communicate it to all members $L \cup R$.

We say that m_l is the leader of $L \cup R$, and it is also the leader of L. We denote the *subtree* of which $p \in G$ is the leader by $G(p)$. For example, in the context of $G = \{1, 2, 3, 4, 5, 6, 7, 8\}, G(1) = G, G(2) = \{2\}, G(3) = \{3, 4\}, G(5) = \{5, 6, 7, 8\}$.

We use the agree primitive to obtain our solution. Below is an example for the creation of a completely distributed key-tree for a group of 8 members (see Figure 4):

1. Members 1 and 2 agree on mutual key K_{12}
   Members 3 and 4 agree on mutual key K_{34}
   Members 5 and 6 agree on mutual key K_{56}
   Members 7 and 8 agree on mutual key K_{78}

2. Members 1,2 and 3,4 agree on mutual key K_{14}
   Members 5,6 and 7,8 agree on mutual key K_{58}

3. Members 1,2,3,4 and 5,6,7,8 agree on mutual key K_{18}

Each round's steps occurs concurrently. In this case, the algorithm takes 3 rounds, and each member stores 3 keys. We now describe the algorithm in the general case, where a key-graph should be built for a group of size N. We assume for simplicity that $N = 2^n$. If $N \neq 2^n$ the same algorithm is applicable, but it's description is more complex and harder to understand.

Base case: If the group contains 0 or 1 members, then we are done. If the group contains 2 members then use the agreement primitive to agree on a mutual key.

Recursive step ($N = 2^{n+1}$): Split the group into two subgroups, L and R, containing each 2^n members. Apply the algorithm recursively to L and R. Now, each subgroup possess a key, K_L and K_R respectively. Apply the agreement procedure to L and R such that they agree on a group key.

This algorithm takes $log_2 n$ rounds to complete. Each member stores $log_2 n$ keys.

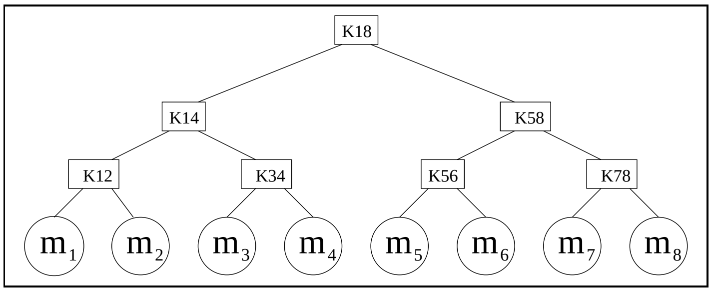

Figure 4: Our solution for a group of 8 members.

In case of join, new nodes are added. For example, if member m_9 joins then key K_{19} is added (see Figures 5,6). The protocol then works as follows:

1. Members $m_1, \ldots, m_8$ and m_9 agree on a mutual key K_{19}

2. K_{19} is the new group key.

In case of failure, some of the tree nodes are replaced. For example, if member m_1 fails then keys K_{14} and K_{18} must be replaced; see Figure 7.

We use a similar strategy to choose new keys:

1. Members m_2 and m_3, m_4 agree on mutual key K_{24}.

2. Members m_2, m_3, m_4 and m_5, m_6, m_7, m_8 agree on mutual key K_{28}

In general this takes $O(log_2 n)$ rounds. With a tree of degree d, it requires $O(log_d n)$ rounds.

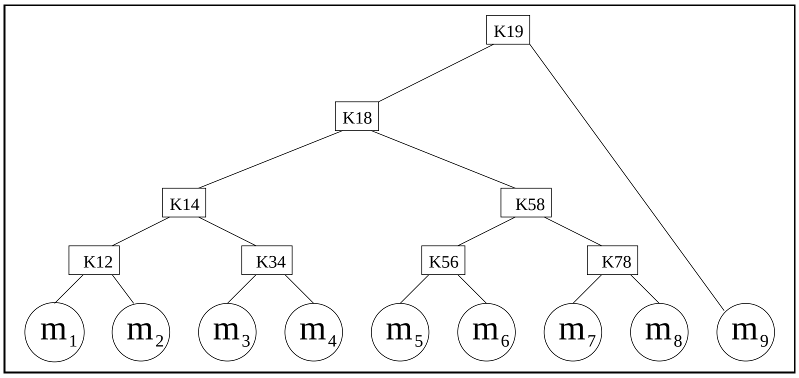

Figure 5: Single member join case.

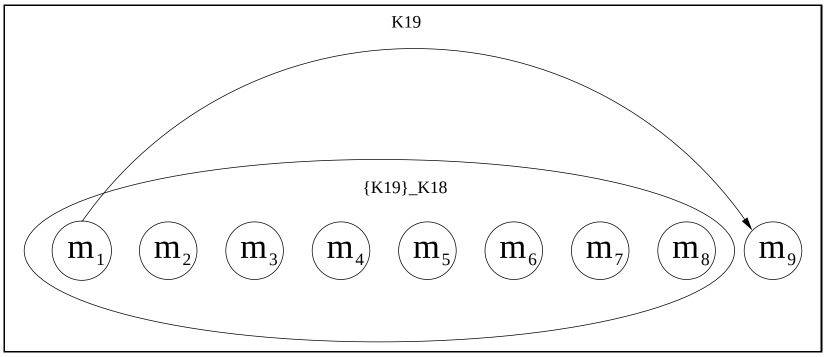

Figure 6: The join algorithm: members m_1 and m_9 agree on key K_{19}. Member m_1 encrypts K_{19} with K_{18} and sends it to members $m_1, \ldots, m_8$.

4.1 Safety

Here we give an informal proof showing why the basic protocol is safe.

First, we need to show why the tree build protocol is safe. During the build we use the agree sequence multiple times. This sequence is safe by induction: It is safe for two members because they simply use a secure channel to communicate. Assume it is safe for subtrees up to size n. Leader m_l passes K_{LR} to m_r using a secure channel. Leaders m_l and m_r then use (safe by induction) subgroup keys K_L, K_R to encrypt K_{LR} prior to dissemination.

The join protocol is safe using a similar argument. The leave protocol is safe since we discard all keys that the leaving member has known, and do not use them to disseminate the new group key.

The optimized solution can be shown to be safe using an analogous argument, though we omit the proof for brevity considerations.

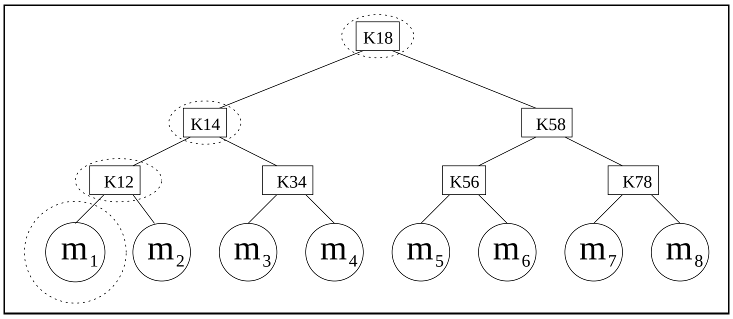

Figure 7: The single member failure case.

4.2 Optimized solution ($\mathcal{O}$)

The basic protocol can be improved to achieve latency of 2 rounds. We state the optimized protocol $\mathcal{O}$, and then provide an example run. Assume for simplicity that $N = 2^n$.

We describe the optimized algorithm recursively:

Base case ($N = 0, 1, 2$): If the group contains 0 or 1 members we are done. If the group contains 2 members then member m_1 is the leader, it chooses key K_{12} and sends it using a secure channel to m_2.

Base case ($N = 4$): In this case, we use the following protocol:

1. (a) Leaders locally choose new keys:
   m_1 chooses K_{14}, K_{12}
   m_3 chooses K_{34}

   (b) Using secure channels:
   $m_1 \to m_2 : K_{14}, K_{12}$
   $m_1 \to m_3 : K_{14}$
   $m_3 \to m_4 : K_{34}$
2. In the clear:
   $m_3 \to m_4 : \{K_{14}\}_{K_{34}}$

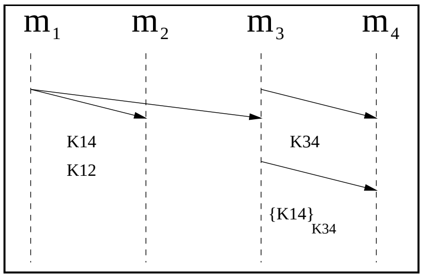

Figure 8: A time-line diagram of the optimized key-graph with 4 members.

We provide this case to show the two stages of the solution: the first stage is the choice of new keys and their dissemination using secure channels, the second stage is the encryption and dissemination of keys received in the first round, using multicast. See Figure 8 for a time-line diagram.

Induction: Assume the group contains 2^{n+1} members, and we can solve the problem in two rounds for the 2^n case. Split the group G into two disjoint subgroups, L and R where $|L|=|R|=2^n$. For each member $p \in G$ Create the list of actions to be performed in stages 1 and 2, $A^n_{p,1}$ and $A^n_{p,2}$ respectively. Mark the leader of L as m_l and the leader of R as m_r.

The new stages are now:

$A^{n+1}_{p,1}$**:** For all members different than m_l, the actions are the same as before, i.e., $A^{n+1}_{p,1} = A^n_{p,1}$. Member m_l initially sets $A^{n+1}_{l,1} = A^n_{l,1}$. It adds another action to $A^{n+1}_{l,1}$:

Choose a new key for the whole group, K_{LR}, and sends it using a secure channel to m_r.

Then, m_l adds K_{LR} as payload to all $A^n_{l,1}$ messages.

$A^{n+1}_{p,2}$**:** Initially, $A^{n+1}_{p,2} = A^n_{p,2}$. All members that receive K_{LR} during $A^{n+1}_{p,1}$ add the following action to $A^{n+1}_{p,2}$:

encrypt K_{LR} with the subtree key and multicast it to $G(p) \setminus \{p\}$.

Our description is recursive, however, we emphasize that it takes only 2 communication phases to perform the action lists by all members of G because the "stages" occur concurrently.

For example, below is an execution with 8 members.

1. (a) Member 1 chooses K_{12}, K_{14}, K_{18}
   Member 3 chooses K_{34}
   Member 5 chooses K_{56}, K_{58}
   Member 7 chooses K_{78}

   (b) Using secure channels, the leaders send the chosen keys as follows:
   $m_1 \rightarrow m_2 : K_{12}, K_{14}, K_{18}$
   $m_1 \rightarrow m_3 : K_{14}, K_{18}$
   $m_1 \rightarrow m_5 : K_{18}$
   $m_3 \rightarrow m_4 : K_{34}$
   $m_5 \rightarrow m_6 : K_{56}, K_{58}$
   $m_5 \rightarrow m_7 : K_{58}$
   $m_7 \rightarrow m_8 : K_{78}$

2. Using regular communication, leaders send:
   $m_3 \rightarrow m_4 : \{K_{14}\}_{K_{34}}, \{K_{18}\}_{K_{14}}$
   $m_5 \rightarrow m_{6,7,8} : \{K_{18}\}_{K_{58}}$
   $m_7 \rightarrow m_8 : \{K_{58}\}_{K_{78}}$

All members can now decrypt and get all keys on route to the root. For example, m_8 receives K_{78} in stage 1, and $\{K_{58}\}_{K_{78}}$ $\{K_{18}\}_{K_{58}}$ in stage 2. It can then decrypt and get K_{18}, K_{58}. This is also shown in a time-line diagram in figure 9. The figure shows how action lists A_1, A_2 are created. Stage S_1 shows all the messages that need to be sent for K_{18} to reach all members. S_2 shows the protocol for subgroups $L = \{m_1, m_2, m_3, m_4\}$ and $R = \{m_5, m_6, m_7, m_8\}$. A_1 and A_2 are then the combination of S_1 and S_2 for the various members.

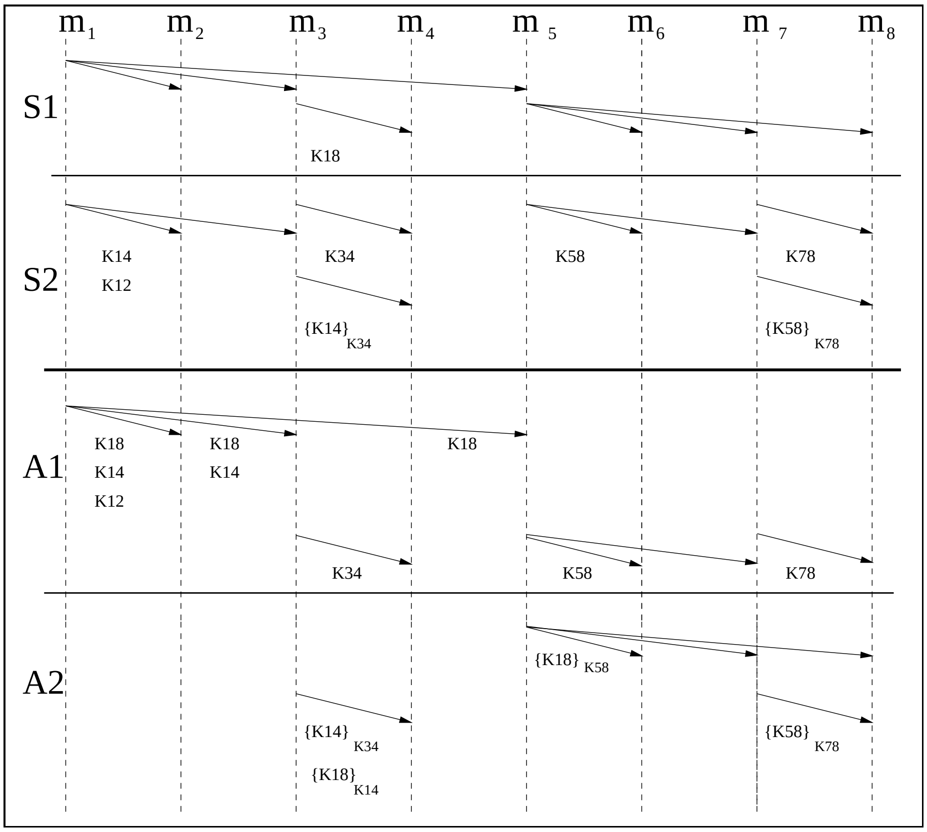

Figure 9: A time-line diagram of the optimized key-graph with 8 members. Time flows from top to bottom.

The join case is efficient using the regular solution, so we make no attempt to improve it.

In case a member leaves, the protocol is simplified. Examine the case where member m_1 left the group. The problem is to choose and distribute keys K_{24}, K_{28} in 2 rounds. This works as follows (Figure 10):

1. (a) m_2 chooses K_{24}, K_{28}.
   (b) Using secure channels:
   $m_2 \rightarrow m_3 : K_{24}, K_{28}$
   $m_2 \rightarrow m_5 : K_{28}$

2. In the clear:
   $m_3 \rightarrow m_4 : \{K_{24}\}_{K_{34}}, \{K_{28}\}_{K_{24}}$
   $m_5 \rightarrow m_{6,7,8} : \{K_{28}\}_{K_{58}}$

m_4 receives $\{K_{24}\}_{K_{34}}$, $\{K_{28}\}_{K_{24}}$ m_3; it already has key K_{34} and it recovers K_{24} and K_{28}. Members 6,7,8 receive $\{K_{28}\}_{K_{58}}$ from m_5; they have key K_{58} and they recover K_{28}.

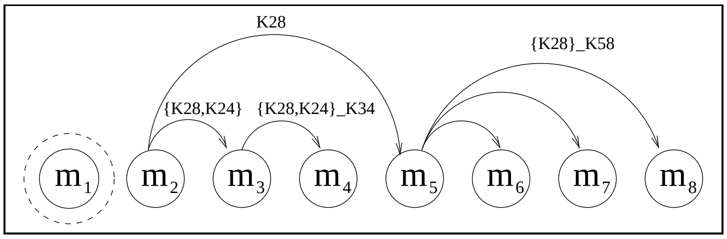

Figure 10: Optimizing the single member failure case.

The leave algorithm costs, in the case of a group of size n, $log_2(n)$ point-to-point messages $O(log_2(n))$ multicasts and the total amount of information passed is $O(log_2 n)^2$ (see the Appendix 7 for a fuller description).

4.3 3 round solution ($\mathcal{O}_3$)

The optimized solution, as stated above, has an efficiency problem with respect to multicast messages. Each subtree-leader sends $log_2(n)$ multicast messages, potentially one for each level of recursion. Since there are $n/2$ such members, we have $O(n * log_2(n))$ multicast messages sent in a typical run of the protocol.

Here we improve $\mathcal{O}$ and create protocol $\mathcal{O}_3$. Protocol $\mathcal{O}_3$ is equal to $\mathcal{O}$ except for one detail, in each view, a member m_x is chosen. All subtree-leaders, in stage 2, send their multicasts messages point-to-point to m_x. Member m_x concatenates these messages and sends them as one (large) multicast. The other members will unpack this multicast and use the relevant part. This scheme reduces costs to $n/2$ point-to-point messages from subtree leaders to m_x, and one multicast message by m_x. Hence, we add another round to the protocol but reduce multicast traffic.

4.4 Costs

Here, we compare the 3-round solution with the regular centralized solution. There are three comparisons — building the key-tree, the join algorithm and the leave algorithm. Protocol $\mathcal{O}_3$ includes a liveness-ensuring stage discussed in the model Section 2. We do not include this sub-protocol in protocol costs, since the centralized solution does not include an equivalent stage. We use tables to compare the solutions and we use the following notations:

 # pt-2-pt: The number of point-to-point messages sent.

 # multicast: The number of multicast messages sent.

 # bytes: The total number of bytes sent

 # rounds: The number of rounds the algorithm takes to complete

First, we compare the case of building a key-tree for a group of size 2^n where there are no preexisting subtrees.

	# pt-2-pt	# multicast	# bytes	# rounds
regular	0	1	$(O(nlog_2n)$	1
distributed	$O(n)$	1	$(O(nlog_2n)$	3

Protocol $\mathcal{O}_3$ takes $O(n)$ more point-to-point messages. These messages are sent concurrently, and are fairly small, hence, their impact is minor. Both protocols require a single (large) multicast. $\mathcal{O}_3$ takes 2 more rounds of communication. Protocol $\mathcal{C}$ requires the key-server to know all the key-tree, a total of n keys. The other members need keep $log_2(n)$ keys. $\mathcal{O}_3$ requires only knowledge of log_2n keys from all members.

The leave algorithm costs:

	# pt-2-pt	# multicast	# bytes	# rounds
regular	0	1	$O(log_2n)$	1
distributed	log_2n	log_2n	$O((log_2n)^2)$	2

The join algorithm costs:

	# pt-2-pt	# multicast	# bytes	# rounds
regular	0	1	$O(1)$	1
distributed	1	2	$O(1)$	2

5 Performance

Here we describe the performance of our algorithm in the context of the Ensemble [10] Group communication system.

We tested our rekeying system on a group of machines, all running versions of the BSD operating system. We used three groups of machines. The first group contained 16 PentiumPro 200Mhz machines connected through 10Mbit/sec switched Ethernet. The connection to the outside is through a 100Mbit/sec connection. The second group included 8 PentiumPro 200Mhz machines connected through a Myrinet LAN, and to the rest of the network through a gateway. The third group contained 2 Pentium 120Mhz connected through a shared 10Mbit/sec Ethernet to the network. The three groups were connected through 10Mbit/sec Ethernet.

Our test program uses a view-size parameter, n. Each run of the program creates a group of size n, each group member on a different machine. Once the group reaches size n, the leader performs a rekey operation. When the rekey is complete, the leader waits until the group stabilizes. The, it removes a random member from the group. When the new view (size $n-1$) stabilizes, it performed another rekey operation. This alternating process is repeated for the duration of several minutes and rekey times are measured.

In order to focus on the running time of the communication protocol, we did not use PGP in this experiment. Therefore "secure channels" used in the second phase of the protocol are not really secure. Since Ensemble caches secure channels, when they are actually used, using real secure channels and would complicate the timed behavior of our protocol.

We performed the above measurement for groups of sizes 3 to 24, and got two times, a *low* and a *high* for each parameter n. This is depicted in Figure 11, where the x-axis denotes group size, and the y-axis denotes time in seconds. There are three time-lines in the diagram. One for the low times, one for the high times, and another for *prompt* times (see below). Note that the low times are higher than the high times. This is because in the lower view change, the leader has to merge $log(n)$ partitions, where in the higher case, it only needs to allow a single member to an existing group.

We also measured the time Ensemble takes to perform a simple view change. This is depicted as the lowest line in the graph, *prompt*. The actual time the rekeying protocol requires, is the difference between the hi,lo and prompt lines.

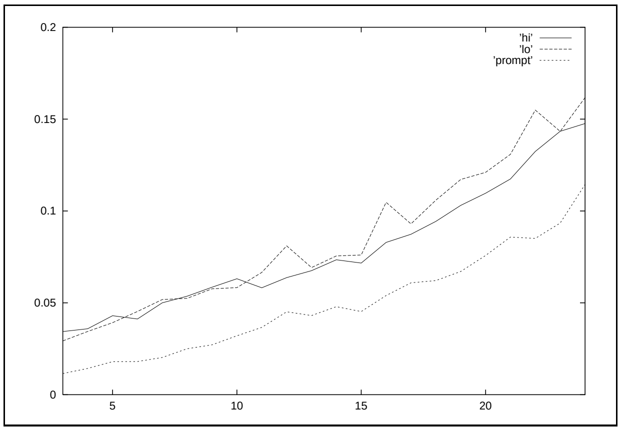

Figure 11: Rekey time, the y-axis denotes time in seconds, the x-axis denotes group size. The time lines are "lo" for the low view, "hi" for high view, and "prompt" for a simple view-change.

The difference between the high,low and prompt lines is more or less constant and is about 40 milliseconds at 24 machines. We believe this is fast enough for most uses of group rekeying, and shows our algorithm as efficient and scalable to large groups.

6 Conclusions

We have shown how to convert a centralized and non-fault-tolerant protocol into one which is decentralized and tolerant of failures, and yet has nearly the same cost as the original protocol.

While $\mathcal{O}_3$ as stated, requires a group communication system it uses a relatively minor part of the provided functionality. It could be run, with relatively minor adjustments, over wide-area Internet applications. We are in the process of examining this approach and will report our findings elsewhere.

References

[1] A. Ballardie. Scalable multicast key distribution. Technical Report 1949, IETF, May 1996.

[2] C.K. Wong, M. Gouda, and S.S. Lam. Secure group communication using key graphs. In *ACM SIGGCOM*. ACM, September 1998.

[3] Debby M. Wallner, Eric J. Harder, and Ryan C. Agee. Key management for multicast: Issues and architectures. Internet Draft draft-wallner-key-arch-01.txt, IETF, Network Working Group, September 1998.

[4] L. Gong. Enclaves: Enabling secure collaboration over the internet. *IEEE Journal on Selected Areas in Communications*, (15(3):567–575), April 1997.

[5] H. Harney and C. Muckenhirn. Group key management protocol architecture. RFC 2094, IETF, 1997.

[6] M.K. Reiter, K.P. Birman, and L. Gong. Integrating security in a group oriented distributed system. TR 92-1269, Department of Conmputer Science, University of Cornell, February 1992.

[7] O. Rodeh, K.P. Birman, M. Hayden, Z. Xiao, and D. Dolev. Ensemble security. Tr, Department of Conmputer Science, University of Cornell, 1998.

[8] R. Canetti, J. Garay, G. Itkis, D. Micciancio, M. Naor, and B. Pinkas. Multicast security: A taxonomy and efficient constructions, March 1999. To appear in Infocom99.

[9] M. Reiter, K.P., Birman, and R. Renesse. A security arcihtecture for fault-tolerant systems. *ACM Transactions on Computer Systems*, 4(12), November 1994.

[10] R.V. Renesse, K. P. Birman, M. Hayden, A. Vaysburd, and D. Karr. Building adaptive systems using ensemble. TR 97-1638, Cornell University, July 1997.

[11] S. Mittra. Iolus: A framework for scalable secure multicasting. In *ACM SIGCOMM*, September 1997.

7 Appendix

7.1 A: The cost of the leave algorithm, optimized case

Examine a tree of depth n. W.l.o.g. m_1 leaves, and m_2 chooses new keys $K = \{K_{23}, K_{24}, K_{28}, \ldots\}$ to replace all the keys from m_1 to the root. Member m_2 sends the new keys to sub-leaders $m_3, m_5, m_9, m_17, m_{2^n+1}, \ldots$ through secure channels. All new keys are sent to m_3, all keys except K_{23} are sent to 5, $K\{K_{23}, K_{24}\}$ are sent to 5 and etc. Typical sub-leader p encrypts the received keys with the key of $G(p)$ and multicasts this information to the members. The communication cost is $log_2(n)$ point-to-point messages and $log_2(n)$ multicasts. The total amount of information passed is $O(log_2 n)^2$.

7.2 B: The complete algorithm

This appendix section aims to clarify and describe how key-trees are managed and how they evolve between group views, when group-views may dynamically change.

For example, examine a group G of 8 members $m_1, \ldots m_8$ with a complete binary key-tree. Assume member m_1 fails and the key-tree is split into three disjoint components: $\{m_2\}, \{m_3, m_4\}$, $\{m_5, m_6, m_7, m_8\}$ (see Figure 12). Each of the components possesses a complete key-tree. The algorithm merges the subtrees together into a single key-tree.

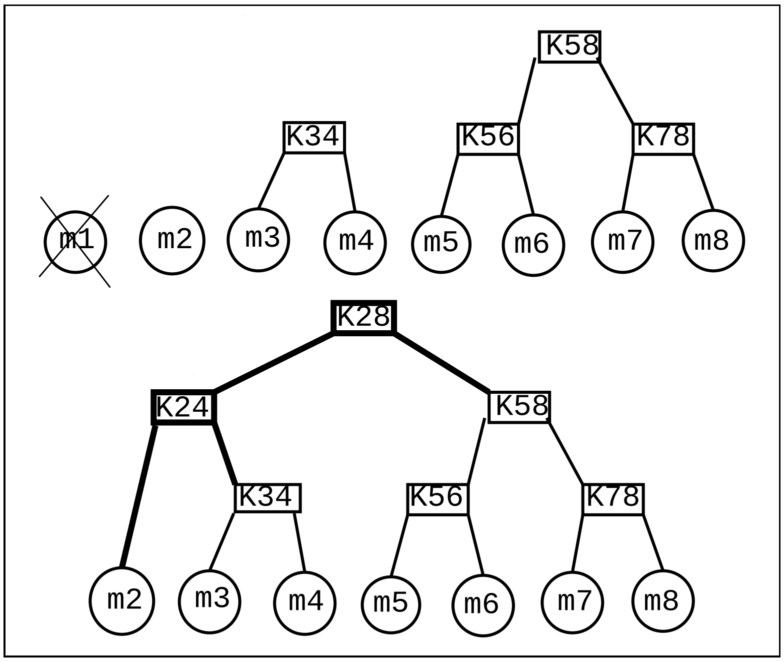

Figure 12: An example of a merge after failure.

An example of a merge sequence is depicted in Figure 13. Three groups G_1, G_2 and G_3 where $G_1 = \{m_1, m_2\}, G_2 = \{m_3, m_4, m_5\}, G_3 = \{m_6, m_7, m_8, m_9\}$ merge together. The algorithm merges the smaller components first, and works its way until all components are merged.

We term any set of members that have a spanning key-tree a *subgroup*. For example, in the first case the subgroups are $\{m_2\}, \{m_3, m_4\}, \{m_5, m_6, m_7, m_8\}$, and in the second, G_1, G_2, G_3 are the subgroups. The leader of a sub-group is its least ranked member. For example, the leader of $\{m_5, m_6, m_7, m_8\}$ is m_5. The leader of sub-group G is called a sub-leader, and is denoted by $L(G)$.

When a new view is initiated, a spanning key-tree for all the view members is created out of existing partial key-trees. To this end each sub-leader sends its key-tree *structure* to the group leader $L(G)$. The structure of the key-tree includes only key names, not actual key values. While the group key-tree structure is known to all group members, actual key values are kept secret. A member may know only the keys on the path from itself to the root. We denote key-tree structures as *S-trees*.

$L(G)$ computes the new key-tree structure and broadcasts it to the view. The sub-leaders follow the protocol steps required of them to build a full key-tree. Once sub-leaders are done, and all group members received all the key-information, they send an acknowledgment back to $L(G)$. When $L(G)$ gets acknowledgments from all group members it knows that the full tree has been created, and it sends a *Done* message to the group.

Some notation: A, B, C, T, T_1, T_2 denote trees. k, k' denote keys, $T(k, A, B)$ is a tree with k is the top key, A is the left subtree, and B is the right subtree. If X is a tree then $h(X)$ is it's height and $K(X)$ is its top key. The algorithm described below merges two trees, A, B, together into $merge(A, B)$.

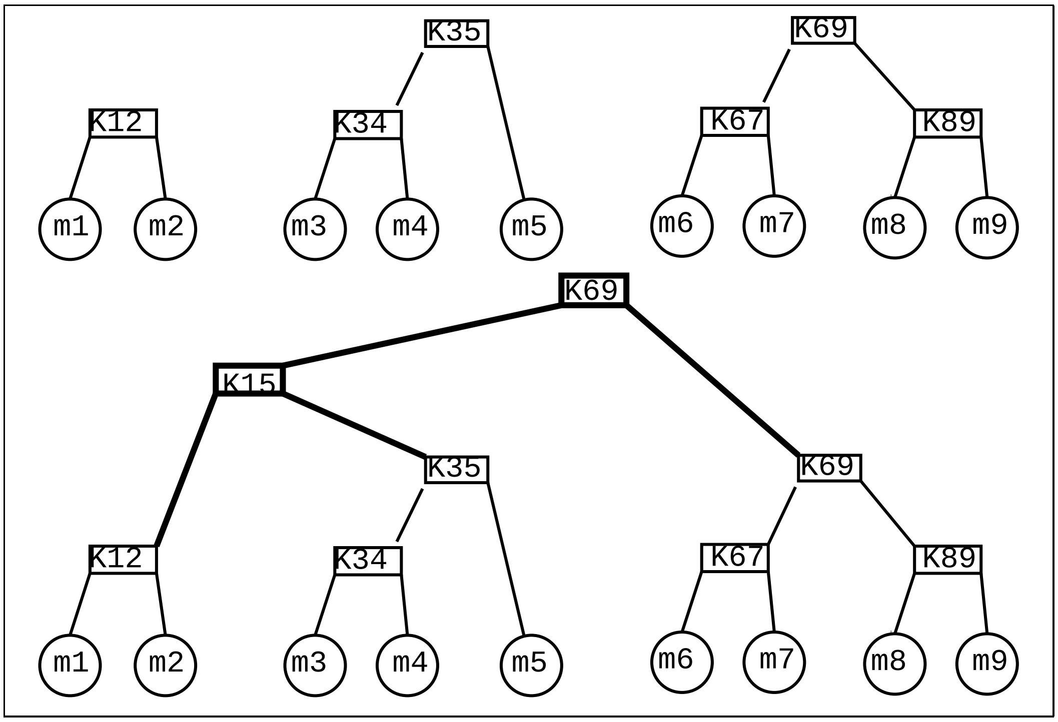

Figure 13: An example of merging several disjoint components.

For example, in the failure case, the protocol would work as follows:

7.3 Local computation by the leader

The leader receives a list of S-trees $\mathcal{L}$. In order to merge them together, it sorts $\mathcal{L}$ according to height into an array. Iteratively, the smallest two S-trees are poped out, merged, and returned to the array. The merge sequence is complete when a single S-tree remains.

Assuming component S-trees are balanced binary AVL trees, the merged S-tree is also AVL. This property guaranties that the route from all members to the root is of length $O(log(n))$. Hence, the failure of any single member will force the system to discard only $O(log(n))$ sub-keys, out of the total $n-1$ sub-keys used in the key-tree.

7.4 Two-way-merge

Here we describe the algorithm by which two S-trees A and B are merged. We denote by $merge(A, B)$ the merged tree.

The base case is when A and B are of equal height. In this case a new key name k is chosen, and $merge(A, B) = T(k, A, B)$.

When $h(A) \neq h(B)$, we assume w.l.o.g. $h(A) > h(B)$, and $A = T(\_, A_1, A_2)$. Match $h(B)$ against the height of A's subtrees A_1, A_2. Recursively merge B with the smaller subtree (if there exists a smaller one). When a subtree A' such that $h(B) = h(A')$ or $h(B) = h(A') + 1$ is found,

1. $m_3 \to m_2 : T(K_{34}, m_3, m_4)$
2. $m_5 \to m_2 : T(K_{58}, T(K_{56}, m_5, m_6), T(K_{78}, m_7, m_8))$
3. $m_2 \to G : T(K_{28}, T(K_{24}, m_2, T(K_{34}, m_3, m_4)),$
   $T(K_{58}, T(K_{56}, m_5, m_6), T(K_{78}, m_7, m_8)))$.
4. m_2 and m_3 create : $T(K_{24}, m_2, T(K_{34}, m_3, m_4))$.
   m_2 and m_5 create :
   $T(K_{28}, T(K_{24}, m_2, T(K_{34}, m_3, m_4)),$
   $T(K_{58}, T(K_{56}, m_5, m_6), T(K_{78}, m_7, m_8)))$.
5. $m_3, m_4, m_5, m_6, m_7, m_8 \to m_2$: Ack
6. $m_2 \to G$: Done

Figure 14: Complete protocol for a simple failure case. $G = m_1 \dots m_8$, member m_1 fails and components $\{m_2\}, \{m_3, m_4\}, \{m_5, m_6, m_7, m_8\}$ merge.

choose a new key name *key* and create a key-tree $A'B = T(key, A', B)$. Now set $merge(A, B) = A$ where A' is replaced by $A'B$.

Claim: if A and B are AVL trees then $merge(A, B)$ is also AVL.

Proof:

- If $h(A) = h(B)$ then $merge(A, B)$ is also AVL.
- Assume correctness all for all trees A, B where $\|h(A) - h(B)\| \leq n$.
- Proof for $h(A) = h(B) + n + 1$. $A = T(key, A_1, A_2)$, hence, tree B is added to the smaller subtree of A. If $h(A_1) = h(A_2)$ then the new tree will be $T(key, A_1, T(key', A_2, B))$ which is AVL since $h(A_1) + 1 = h(T(key', A_2, B)$. If, w.l.o.g., $h(A_1) = h(A_2) + 1$ then the new tree will also be: $T(key, A_1, T(key', A_2, B))$. It is AVL since in this case $h(A_1) = h(T(key', A_2, B))$. $\square$

7.5 Evaluation of the group S-tree

The leader computes the group S-tree and multicasts it to the group. Each sub-leader infers the steps it must complete to create the tree, and agree on key values. A sub-leader may take part in several two-way-merges. For maximal parallelism each two-way-merge is performed as follows:

Assume subtrees A' (A' as part of larger tree A) and B are merged in the group S-tree. The leaders $L(A')$ and $L(B)$ engage in the following protocol:

1. (a) $L(A') \to L(B) : K_{A'B}, key\_val$
   $L(A')$ chooses a new key named $K_{A'B}$ with value *key_val* and sends it using a private connection to $L(B)$.

   (b) $L(A') \to G : \{K_{A'B}\}K_{A'}$
   $L(A')$ sends the group the new key $K_{A'B}$ encrypted with $K_{A'}$. Only members of A' can decrypt this message and extract $K_{A'B}$.

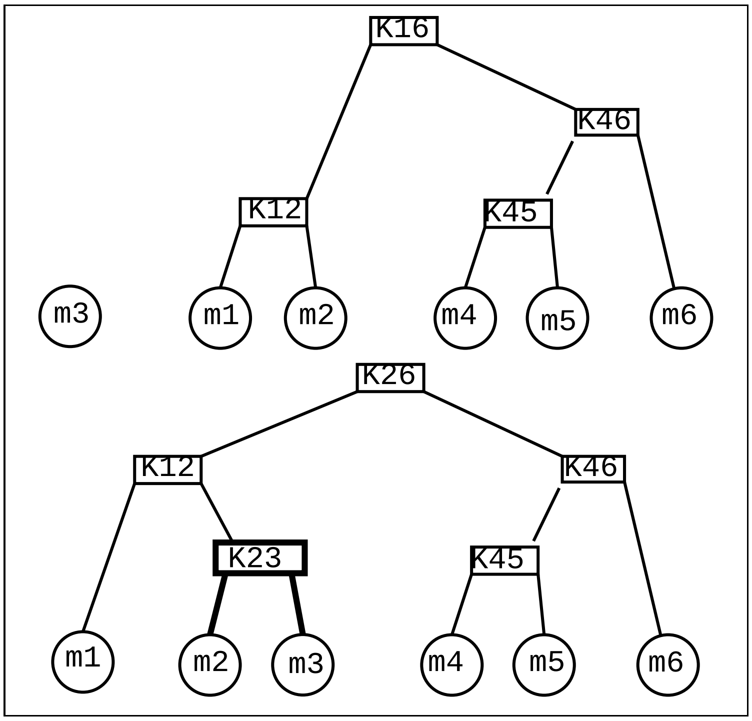

Figure 15: Merge. Two sub-groups are merged, m_3 and $\{m_1, m_2, m_4, m_5, m_6\}$.

   (c) $L(A') \rightarrow G : \{\text{keys above } K_{A'B}\} K_{A'B}$
   $L(A')$ sends the group the keys in A above A'. Only members of $A' \cup B$ can extract these keys.

2. $L(B') \rightarrow G : \{K_{A'B}\} K_B$
   $L(B)$ sends the group the new key of $T(\_, A', B)$. Only members of B can extract it.

For example, in the failure of Figure 14 members m_2 and m_3 create : $T_{24} = T(K_{24}, m_2, T(K_{34}, m_3, m_4))$. Members m_2 and m_5 create $T_{28} = T(K_{28}, T_{24}, T(K_{58}, T(K_{56}, m_5, m_6), T(K_{78}, m_7, m_8)))$. This is performed as follows:

A common tree to $\{m_2\}$ and $\{m_3, m_4\}$ is created by:

1. $m_2 \rightarrow m_3 : K_{24}$

2. $m_3 \rightarrow G : \{K_{24}\} K_{34}$

A common tree to $\{m_2, m_3, m_4\}$ and $\{m_5, m_6, m_7, m_8\}$ is created by:

1. (a) $m_2 \rightarrow m_5 : K_{28}$
   (b) $m_2 \rightarrow G : \{K_{28}\} K_{24}$

2. $m_5 \rightarrow G : \{K_{28}\} K_{58}$

Claim: a complete n-way-merge can be performed in 2 parallel stages. Note that when performing several such merges in one step, step 1.c is problematic. The difficult point is that $L(A')$ need not necessarily know all the keys above it when it performs stage 1.c. Therefore, it sends only those keys it knows at the time. We shall show that inspite of this difficulty, the protocol does indeed complete in 2 parallel stages.

Proof:

- Base case: Two trees can be merged in two steps.
- Induction: Assume n trees can be merged in two phases. Now we need to show for $n+1$ trees. We merge first the smaller n trees into tree T_1. Denote the last tree by T_2, and assume w.l.o.g that $h(T_1) \geq h(T_2)$, and that $h(T_2) = h(T_{1'})$, where $T_{1'}$ is a subtree of T_1. The final tree will be T_1 with $T_{1'}$ replaced with $T_{1'2} = T(k, T_{1'}, T_2)$. The leader of $T_{1'}$ chooses $K(T_{1'2}) = k$ and passes it to $T_{1'}$, hence members of $T_{1'}$ know all the keys on the path to the root in $merge(T_1, T_2)$. Members of T_2 know the keys on the path to the root of T_2 (by assumption). They know k since it is passed from $L(T_{1'})$ to $L(T_2)$ and to T_2. Since $L(T_{1'})$ multicasts all the keys (it initially knows) above $T_{1'}$ encrypted with k then T_2 have knowledge of the same key-set as members of $T_{1'}$. By the induction hypothesis, $T_{1'}$ know all the keys above k when the protocol completes, hence T_2 will know these keys as well. □

7.6 The protocol in 3 stages

In order to turn the protocol from 2 to 3 stages, we use the same trick as we used in chapter 1. We use member $L(G)$ to gather all the would-be-multicasts of of stages 1 and 2. $L(G)$ then multicasts them bundled up in one message. Each member picks the relevant part, and decrypts.

7.7 State kept in the group

Each member needs to keep $log(n)$ keys, namely, all the keys from itself to the root. Also, it needs to keep the structure of the key tree. This is required so that it will be able to recover from group member failures and group partitions.

A member should keep the structure of the subtree of which it is a leader and the list of leaders above it. To show this is enough, examine what happens when there is a failure in the group. The tree is disconnected into several disjoint components. These components should be reconnected into a single tree. To do so, all component leaders should sent their tree structure the leader, which will reconnect the tree. Hence, in the event of failure, a member should know : (1) If it is a sub-leader. (2) What is the tree structure of its sub-tree.

For example, in the scenario in Figure 7, a group of 8 members $m_1, \ldots m_8$ with a complete binary key-tree, splits due to the failure of m_1. Three components are created: $\{m_2\}, \{m_3, m_4\}, \{m_5, m_6, m_7, m_8\}$. Member m_2 need not remember anything. Member m_3 keeps the structure $T(\_, m_3, m_4)$ and member m_5 keeps $T(\_, T(\_, m_5, m_6), T(\_, m_7, m_8))$. This is enough.

Here we account for the total amount of memory required by the key-tree algorithm throughout the group. We do the accounting for a complete binary tree, of n leaves. An actual tree for n leaves is an AVL tree, which has properties not far from a complete binary tree.

First we count the state kept regarding the tree structure. The total state needed is a function, f, with the following properties:

- $f(1) = 1$
- $f(n) = n/2+2*f(n/2)$. Since for a tree $A = T(,A_1, A_2)$, the state kept in A_1, A_2 is $2*f(n/2)$, and $L(A_1)$ should also keep the state of A_2, which is of size $n/2$.

The first equation is derived from the fact that the group tree leader should know all the tree, and besides, each subtree should keep $f(n/2)$ state. Hence, we account the tree leader for keeping the state of right half of the tree (size $n/2$), and add the state kept by each tree half. It can be shown that these recursive equations for f admit the solution $f(n) = O(nlog(n))$.

Each member needs to keep the set of keys on the way to the root, and the set of leaders above it. This amounts to $O(logn)$ information. Summing over all the tree, this amounts to $O(n * logn)$.

To summarize $O(n * logn)$ information is required for this protocol.

7.8 Summary

This section has described the algorithms required to manage the key-tree for a dynamic peer-group. We have shown that our algorithm can recover from any failure or group reconfiguration in 2-3 rounds with $O(n)log(n)$ communication while requiring $O(n * log(n))$ memory for state information. The centralized algorithm, that does not have fault-tolerance properties, requires a single communication round and $O(n * log(n)))$ memory (with a smaller factor however).