Optimized Group Rekey for Group Communication Systems

Ohad Rodeh, Kenneth P. Birman, Danny Dolev [1]

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, 7, 4, 3, 21, 26] 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.

A GCS provides reliable multicast and membership services to groups of processes, complying with the the *Virtual Synchrony* (VS) [11] reliability model. 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*. GCSs provide Virtually Synchronous [11] communication to applications in spite of changing network conditions and process crashes. The VS reliability guarantee is, in simple terms, "atomic failure". This means that if a process crashes in a group, then the remaining members view this at "the same time". Remaining processes deliver the same set of messages prior to the view-change, hence, they get the view notification "simultaneously". VS requires that, when progress is made, members maintain agreement upon the membership – a requirement which implies that, under certain patterns of failures, progress may not be possible [5]. Fortunately, such patterns of failure are very unlikely. Consequently, GCS systems live within certain limitations. For example, the Ensemble system membership mechanism may fail to stabilize, if extreme network conditions occur. It may block and unblock the group for extended periods. GCSs guaranty that if the network is stable for a long enough period, then the membership will stabilize. In such cases, the group-leader is easy to choose: it is the process with lexicographically "smallest" name. A family of such systems have been developed in various places in the world. A partial list includes Totem [13], Spread [29], Relacs [17], Timewheel [15] and Phoenix [2]. Ensemble, our GCS, evolved from Isis [9], Horus [25], and Transis [27].

[1] The authors were supported in part by the Israeli Ministry of Science grant number 032-7892, by DARPA/ONR under grant N00014-92-J-1866, and NSF grant EIA 97-03470.

Group communication scalability is inherently limited by the number of members in the group. The VS model requires all members to constantly participate in GCS protocols, send acknowledgments, requests for retransmission, flow control information and more. As the number of processes in the group increases, the probability that some of them will become temporarily unresponsive increases. This is especially true on off-the-shelf OSes (Unix^{TM} and NT^{TM}) that do not guarantee a real-time environment. We have managed to scale Ensemble to run on 100 processes but no more. A study has been published on the VS scalability problem see [10]. Therefore, *scalability*, in this article means up to 100 members.

To overcome the VS scalability problem [10] suggests reducing the reliability model. Other approaches use *light-weight groups* layered on top of a *core-group*. In fact, most GCSs described above have light-weight group extensions. Such an approach uses a client/server architecture where a small number of servers run the actual GCS protocols, providing GCS services to a large number of clients.

Our solution is designed for group communication systems that operate with at most one-hundred members per group, a limit consistent with the scalability currently possible in systems like Ensemble. Work is in progress to develop a new, more scalable, approach to group membership management for Ensemble [10], and in conjunction with this we will revisit the scalability of our security architecture. We note that our algorithm makes use of the virtual synchrony properties offered by Ensemble and hence could only be used in a setting supporting these properties.

A secure GCS ensures that all members in a view are authentic and authorized. It provides a secure key with which all group communication is protected. Members outside the group cannot listen in on group-communications. A partial list of such GCSs include Antigone [20] and Spread [28]. Some GCSs are resistant to Byzantine faults, a partial list includes Totem [12], and Rampart [23].

Our solution is related to other group-key architectures developed for GCSs. These works begin with seminal research by Reiter [16, 24] and Gong [6], continuing to recent research on Ensemble [19]. 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), MAC-ing messages and encrypting the data segments of messages. The term MAC, as used here, is defined in the IETF terms as an HMAC [22]. An HMAC is a Keyed Hash [8] that can be based on any interactive cryptographic hash (e.g., MD5 or SHA-1). It is used protect the integrity of a message.

Here we report on the first half of an effort to integrate 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.

The GCS approach emphasizes the peer-to-peer model, where all members are equally trusted. The group should be able to continue functioning in the event of a partition of member failure. It is possible to create a key-architecture where a centralized key-server, replicated for high availability, chooses and disseminates a group-key to the other members. However, should the key-servers split from the rest of the group — the group would no longer be able to rekey itself. While it is possible to use hybrid approaches, this paper does not use a centralized/replicated server.

2 Model

The “universe” for the purposes of this papers is comprised of a set of machines connected through the Internet. Machines, or processes, can communicate with each other by passing messages through the network. The system is asynchronous: clock drifts are unbounded and messages may be arbitrarily delayed or lost in the network. We do not consider Byzantine failures. The network can split into several disjoint components allowing only machines in the same component to pass messages to each other.

A GCS overcomes these network “inconveniences” and presents the application with a simple interface. As described earlier, a GCS allows the creation of process groups in which reliable ordered multicast and point-to-point messaging is supported. Processes may dynamically join and leave a group, and group components can merge through the GCS protocols and state-transfer.

We assume processes in group have access to trusted authentication and authorization services, as well as to a local key-generation facility. 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.

Ensemble allows the creation of secure-groups where all group members agree on a single symmetric key. Only trusted and authorized members are allowed into the group. Since all members use the same key to MAC and encrypt their messages, no intruder can attack the group or purport to be part of it. 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 MAC-ed with key K_1, and X MAC-ed with key K_2. Note that such a tuple can be protected in a manner that would prevent intruders from tampering with either component of the pair.

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

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

This paper relies strongly on the VS guarantee that all members agree on the group-view. This allows operations such as:

- Agreement on the group leader, just choose m_1
- Split the group in two: if there are n members, split into $[1 \dots n/2]$ and $[n/2+1 \dots n]$.

Hence, we cannot use a reduced reliability model, and we focus on securing GCS abstractions.

Ideally, one would hope that distributed protocols can be proved *live* and *safe*. Key management protocols must also provide *agreement* and *authenticity* properties. Here we define these properties formally, and discuss the degree to which our protocols satisfy them.

Liveness: We say that a protocol is live if, for all possible runs of the protocol, progress occurs. In our work, progress would involve the installation of new membership views with associated group keys, and the successful rekeying of groups.

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

Agreement: the protocol should guarantee that all group-members decide on the same group-key.

Authenticity: An authentic group-key is one chosen by the group-leader.

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 $\mathcal{P}$ (below) securely switches the group key.

Protocol $\mathcal{P}$:

1) The group leader chooses a new key.

2) The leader uses secure channels to send the key securely to the members.

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.

No protocol solving this class of problems can guarantee liveness in an asynchronous networking environment (see FLP [5]). However, our protocol is able to make progress ”most of the time.” The scenarios under which the protocol would fail to make progress are extremely improbable, involving an endless sequence of network partitioning and remerge events, or of timing failures that mimic process crashes. Theoretically, such things can happen, but in any real network, these sequences of events would not occur, hence our protocol should make progress.

We implicitly add the above steps to all protocols, thereby improving their tolerance to failure. However, doing so does not “overcome” the FLP result, nonetheless, our protocols would be live in a realistic settings, and the approach allows us to specify protocols in a more succinct fashion.

The protocol described above ensures agreement because a single member acts as leader. The leader decides on a key and disseminates it to the rest of the group. Hence, agreement is satisfied trivially. All the protocols we use, in essence, use agreed leader or sub-leaders that choose and disseminate keys. Hence, agreement is easily satisfied, and we do not discuss it further, nor provide agreement proofs. Authenticity is satisfied from the GCS security properties. Only an authorized authenticated process can become a group-member. Hence, any key accepted by a member is sent by another member. The originator by virtue of being a group member is authenticated and authorized. Thus, we do not discuss authenticity requirements further.

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

Here we describe a protocol by Wong, Gouda and Lam [3]. 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 3 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 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.

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. As, simplisticly, described so far the protocol takes centralized protocol can take $logN$ messages to complete. In fact, all protocol messages can be combined into a single multicast message sent from the key-server to the clients.

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. Discussion of tree-rebalancing is out of the scope of this paper. Some work has been done to rebalance key-trees in [14].

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.

We describe our solution here with some simple examples. The actual algorithm is much more complex and cannot be presented here for lack of space. The interested reader is referred to [18]. Specifically, we do not discuss how the distributed key-tree is balanced and rebalanced after join-leave sequences.

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:

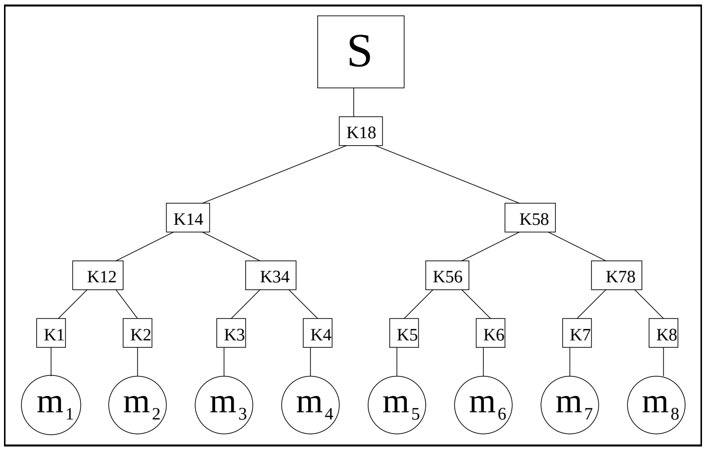
1. m_l chooses a new key K_{LR}, and sends it to m_r using a secure channel.
2. m_l encrypts K_{LR} with K_L and multicasts it to L; m_r encrypts K_{LR} with K_R and multicasts to R.
3. All members of $L \cup R$ securely receive the new key.

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 $L \cup R$. This costs one point-to-point and two multicast messages. We say that *all* members of L and R participate in *agree*, in spite of the fact that actually m_l decides on the key, because all members in $L \cup R$ have some role (active or passive) in the protocol.

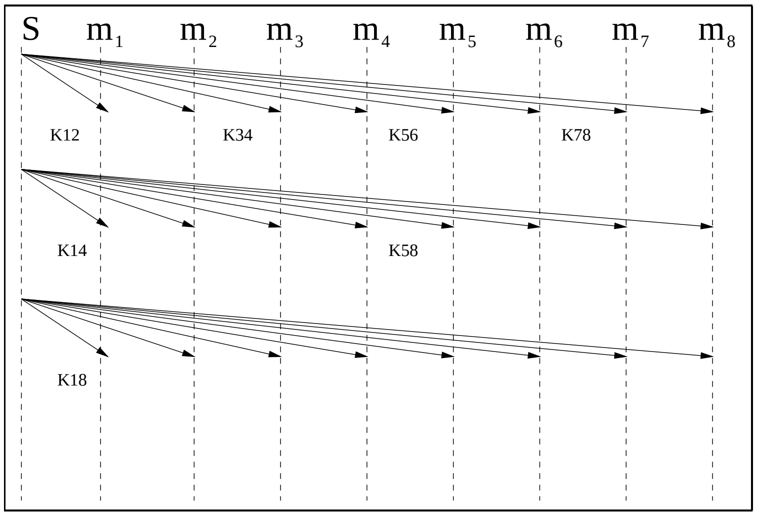
We say that m_l is the leader of $L \cup R$, and it is also the leader of L. We denote the *subtree* of which $p \in G$ is the leader by $G(p)$. For example, in the context of $G = \{1,2,3,4,5,6,7,8\}, G(1) = G, G(2) = \{2\}, G(3) = \{3,4\}, G(5) = \{5,6,7,8\}$.

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}



A keygraph for a group of 8 members.



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.

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.

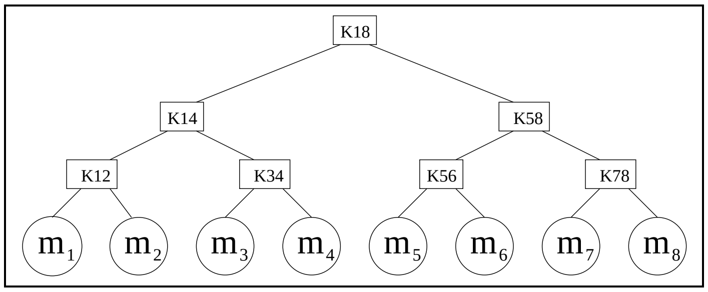
This algorithm takes $log_2 n$ rounds to complete. Each member stores $log_2 n$ keys.

In case of join, new nodes are added. For example, if member m_9 joins then key K_{19} is added (see Figures 4,4). The protocol then works as follows:

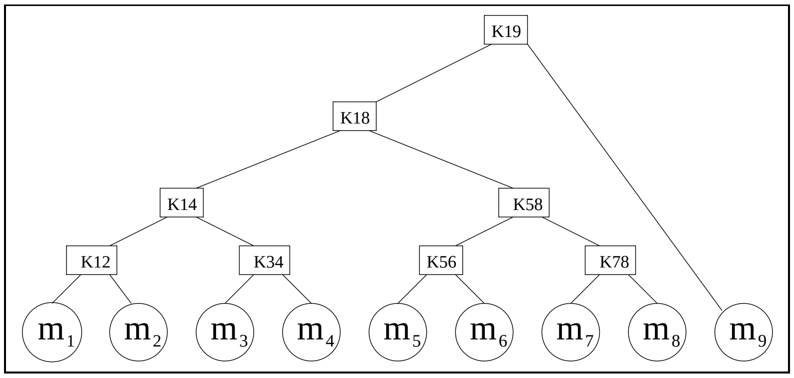
1. Members $m_1, \ldots, m_8$ and m_9 agree on a mutual key K_{19}
2. K_{19} is the new group key.

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 4.

We use a similar strategy to choose new keys:



Our solution for a group of 8 members.



Single member join case.

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.

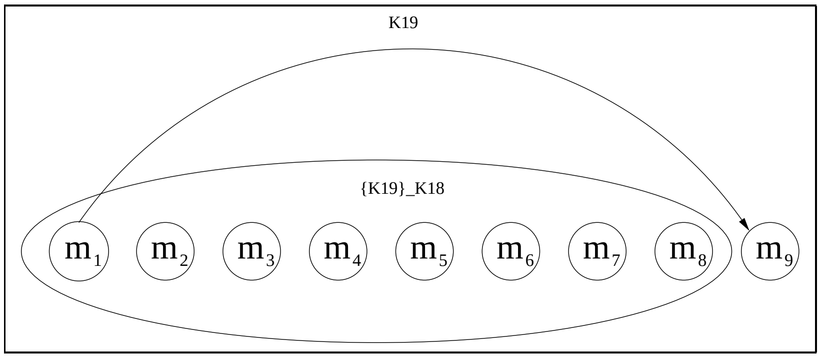
4.1 Safety

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

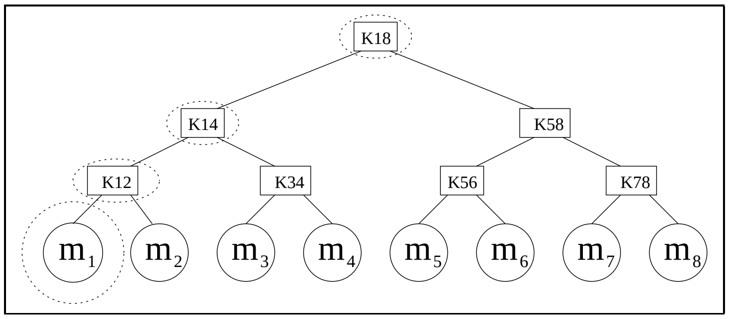
First, we need to show why the tree build protocol is safe. During the build we use the agree sequence multiple times. This sequence is safe by induction: It is safe for two members because they simply use a secure channel to communicate. Assume it is safe for subtrees up to size n. Leader m_l passes 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 is essentially, a reordering of the communication in $\mathcal{B}$. Hence, it is safe by virtue of the basic protocol being safe.



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 $m_1, \ldots, m_8$.



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:
   $m_1 \rightarrow m_2 : K_{14}, K_{12}$
   $m_1 \rightarrow m_3 : K_{14}$
   $m_3 \rightarrow m_4 : K_{34}$
2. In the clear:
   $m_3 \rightarrow m_4 : \{K_{14}\}_{K_{34}}$

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 4.2 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 $\mid L \mid = \mid R \mid = 2^n$. 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 $G(p) \setminus \{p\}$.

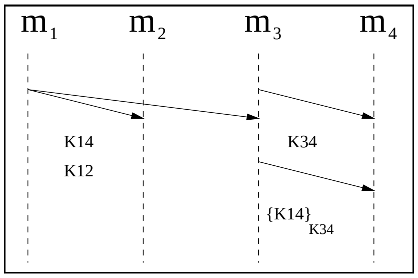
Our description is recursive, however, we emphasize that it takes only 2 communication phases to perform the action lists by all members of G because the "stages" occur concurrently.

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:
   $m_3 \rightarrow m_4 : \{K_{14}\}_{K_{34}}, \{K_{18}\}_{K_{14}}$
   $m_5 \rightarrow m_{6,7,8} : \{K_{18}\}_{K_{58}}$
   $m_7 \rightarrow m_8 : \{K_{58}\}_{K_{78}}$



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

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 4.2. 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.

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 4.2):

1. (a) m_2 chooses K_{24}, K_{28}.
   (b) Using secure channels:
   $m_2 \to m_3 : K_{24}, K_{28}$
   $m_2 \to m_5 : K_{28}$
2. In the clear:
   $m_3 \to m_4 : \{K_{24}\}_{K_{34}}, \{K_{28}\}_{K_{24}}$
   $m_5 \to m_{6,7,8} : \{K_{28}\}_{K_{58}}$

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}.

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 $O(log_2 n)^2$.

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

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

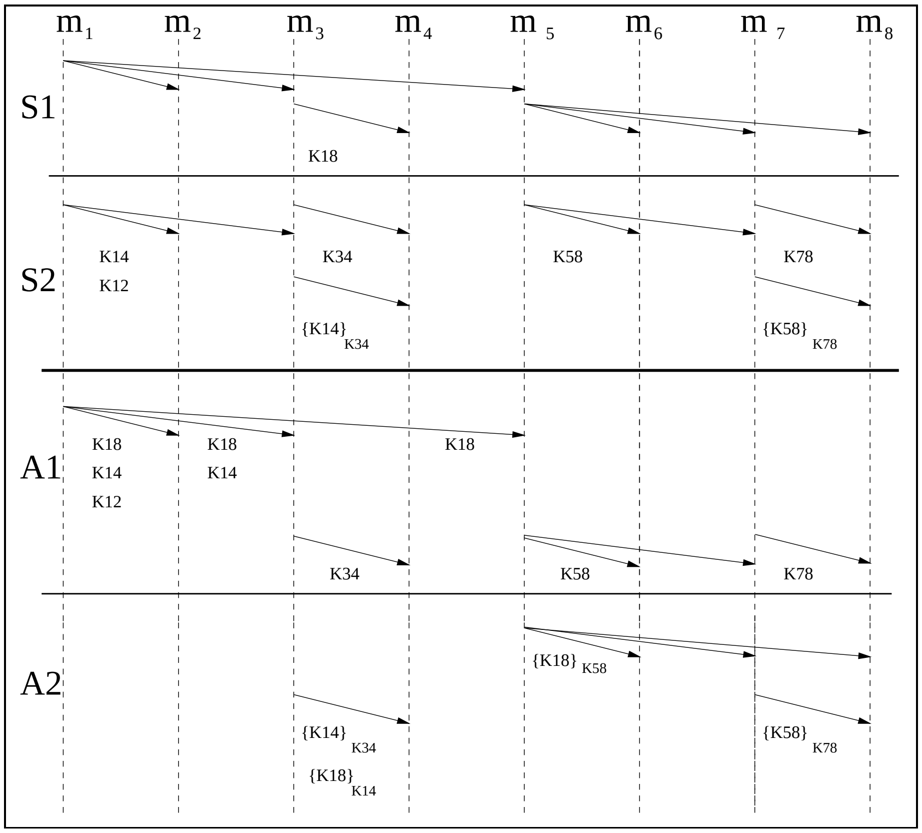
Here, we compare the 3-round solution with the regular centralized solution. There are three comparisons — building the key-tree, the join algorithm and the leave algorithm. Protocol $\mathcal{O}_3$ includes a liveness-ensuring stage discussed in the model Section 2. We do not include this sub-protocol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

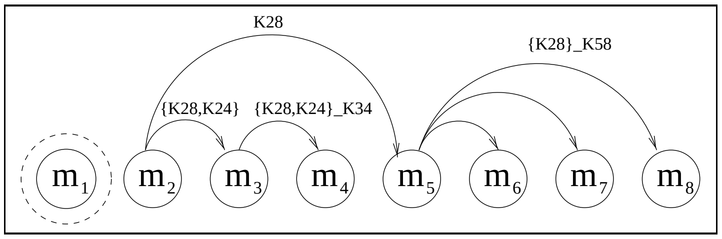
# multicast: The number of multicast messages sent.

# bytes: The total number of bytes sent

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



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



Optimizing the single member failure case.

First, we compare the case of building a key-tree for a group of size 2^n where there are no preexisting subtrees. The following table summarizes the costs for each algorithm:

	# pt-2-pt	# multicast	# bytes	# rounds
$\mathcal{C}$	0	1	$(O(nlog_2n)$	1
$\mathcal{O}_3$	$O(n)$	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
$\mathcal{C}$	0	1	$O(log_2n)$	1
$\mathcal{O}_3$	log_2n	log_2n	$O((log_2n)^2)$	2

The join algorithm costs:

	# pt-2-pt	# multicast	# bytes	# rounds
$\mathcal{C}$	0	1	$O(1)$	1
$\mathcal{O}_3$	1	2	$O(1)$	2

5 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. Malloth and A. Schiper. View Synchronous Communication in Large Scale Networks. In *Proc 2nd Open Workshop of the ESPRIT projet BROADCAST (#6360)*, July 1995.

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

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

[5] M. Fischer, N. Lynch, and M. Paterson. Impossibility of distributed consensus with one faulty process. *J. ACM*, 32:374–382, April 1985.

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

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

[8] H. Krawczyk, M. Bellare, and R. Canetti. Hmac: Keyed-hashing for message authentication. RFC 2104, IETF, Febuary 1997.

[9] K. Birman and R. V. Renesse. *Reliable Distributed Computing with the Isis Toolkit*. IEEE Computer Society Press, 1994.

[10] K. P. Birman, M. Hayden, O. Ozkasap, Z. Xiao, M. Budiu, and Y. Minsky. Bimodal multicast. *ACM Transactions on Computer Systems*, May 1999.

[11] K.P. Birman and T.A. Joseph. Exploiting virtual synchrony in distributed systems. TR 87-811, Department of Conmputer Science, University of Cornell, 1987.

[12] K.P. Kihlstrom, L.E. Moser, and P.M. Melliar-Smith. The securering protocols for securing group communication. In *Hawaii International Conference on System Sciences*, volume 3(31), pages 317–326, january 1998.

[13] L. E. Moser, P. M. Melliar-Smith, D. A. Agarwal, R. K. Budhia, and C. A. Lingley-Papadopoulos. Totem: A fault-tolerant multicast group communication system. In *Communications of the ACM*, April 1996. homepage: http://beta.ece.ucsb.edu/totem.html.

[14] M. J. Moyer, J.R. Rao, and P. Rohatgi. Maintaining balanced key trees for secure multicast. Technical report, IETF, June 1999. draft-irtf-smug-key-tree-balance-00.txt.

[15] S. Mishra. Constructing applications using the timewheel group communication service. In *Proceedings of the 1998 International Conference on Parallel and Distributed Processing Techniques and Applications*, pages 1591–1598, July 1998.

[16] 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.

[17] O. Babaoglu, R. Davoli, and A. Montresor. Partitionalbe Group Membership: Specification and Algorithms. TR UBLCS97-1, Department of Conmputer Science, University of Bologna, January 1997.

[18] O. Rodeh, K. P. Birman, and D. Dolev. Optimized group rekey for group communication systems. Technical Report 2, Department of Computer Science, Hebrew Univesity, 1999.

[19] O. Rodeh, K.P. Birman, M. Hayden, Z. Xiao, and D. Dolev. Ensemble security. TR 1703, Department of Conmputer Science, University of Cornell, 1998.

[20] P. D. McDaniel, A. Prakash, and P. Honeyman. ”Antigone: A Flexible Framework for Secure Group Communication”. In *Proceedings of the 8th USENIX Security Symposium*, August 1999.

[21] 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.

[22] R. Shirey. Security glossary. Technical report, IETF, 1999. draft-shirey-security-glossary-00.txt.

[23] M. Reiter. Secure agreement protocols: Reliable and atomic group multicast in rampart. In *ACM Conference on Computer and Communication Security*, pages 68–80, November 1994.

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

[25] R.V. Renesse, K.P. Birman, and S. Maffeis. Horus, a flexible group communication system. *Communications of the ACM*, April 1996.

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

[27] Y. Amir, D. Dolev, S. Kramer, and D. Malki. Transis: A Communication Sub-System for High Availability. In *FTCS conference*, July 1992.

[28] Y. Amir, G. Ateniese, D. Hasse, Y. Kim, C. Nita-Rotaru, T. Schlossnagle, J. Schultz, Jonathan Stanton, and Gene T-sudik. Secure group communication in asynchronous networks with failures: Integration and experiments. In *1999 International Conference on Distributed Computing Systems*, August 1999. In submission.

[29] Y. Amir and J. Stanton. The spread wide area group communication system. TR CNDS-98-4, Department of Computer Science, 1998.