Using AVL Trees for Fault Tolerant Group Key Management

Ohad Rodeh, Kenneth P. Birman, Danny Dolev*

November 2, 2000

Abstract

In this paper we describe an efficient algorithm for the management of group-keys for Group Communication Systems. Our algorithm is based on the notion of *key-graphs*, previously used for managing keys in large IP-multicast groups.

The standard protocol requires a centralized key-server that has knowledge of the full key-graph. Our protocol does not delegate this role to any one process. Rather, members enlist in a collaborative effort to create the group key-graph. The key-graph contains n keys, of which each member learns $\log_2 n$.

We show how to balance the key-graph, a result that is applicable to the centralized protocol. We also show how to optimize our distributed protocol and provide a performance study of its capabilities.

1 Introduction

This paper describes an efficient algorithm for key-management in Group Communication Systems (GCSs).

Use of GCSs is now common in industry when clustering technology is required. The SP2, and other IBM clustering systems use the Phoenix system [3], IBM AS/400 clusters use the Clue system [8], and Microsoft Wolfpack clusters use Group Communication technology at the core of their system [32]. GCSs are used for other purposes as well. The first industrial strength GCS, Isis [11], has been used for air-traffic control, large-scale simulations, in the New-York and Swiss stock exchanges and more. Some

*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.

of these projects are described in [2]. Adding security to Group Communication Systems will enable industry to use this technology were it was not possible before, in unprotected wide area networks.

In our work we use the *fortress* security model. The "good guys" are protected by a castle from the hordes of barbarians outside. In the real world this maps to a situation where the "good guys" are the organization's machines, which need protection from hackers lurking outside.

In the context of a Group Communication System, the basic entity is a process (member). The set of honest processes requires protection from the adversary. Since the honest members are distributed across the network, they cannot be protected by a firewall. An alternative vehicle for protection is a shared secret key. Assuming all members agree on the same key, all group messages can be MAC-ed[1] and encrypted. Assuming the adversary has no way of retrieving the group-key, the members are protected.

The group-key needs special handling as it must be distributed *only* to authenticated and authorized group members. This raises two issues: merge and rekey.

- How do we allow new members into an existing group? More generally, how do we merge two existing group components, each using a different group-key?
- How do we switch the secret key, without relying on the old key?

Here, we confine ourselves to describing an efficient rekey algorithm. For a discussion on merge-related issues see [22].

Other work on GCS key architectures and security has been performed in Spread [35], Antigone [23], Rampart [28], and Totem [14]. We postpone discussion of related work until section 7.

2 Model

Consider a universe that consists of a finite group $\mathcal{U}$ of n processes. Processes communicate with each other by passing messages through a network of channels. 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.

[1] MAC is a Message Authentication Code algorithm. In practice, the group-key is composed of two disjoint pieces. A separate key for encryption, and a separate key for MAC-ing.

Processes may get *partitioned* from each other. A partition occurs when $\mathcal{U}$ is split into a set $\{P_1, \ldots, P_k\}$ of disjoint subgroups. Each process in P_i can communicate only with processes in P_i. The subsets P_i are sometimes called *network-components.* We shall consider *dynamic partitions*, where in network-components dynamically merge and split. The partition model is more general than the more common "crash failure" model since crash failures may be modeled as partitions, but not the converse.

A GCS creates process groups in which reliable ordered multicast and point-to-point messaging is supported. Processes may dynamically join and leave a group. Groups may dynamically partition into many *components* due to network failures/partitions; when network partitions are healed group components remerge through the GCS protocols. Information about groups is provided to group members in the form of *view* notifications. For a particular process p a view contains the list of processes currently alive and connected to p. When a membership change occurs due to a partition or a group merge, the GCS goes through a (short) phase of reconfiguration. It then delivers a new view to the applications reflecting the (new) set of connected members.

In what follows p, q, and s denote Ensemble processes and V, V_1, V_2 denote views. The generic group is denoted by G. G's members are numbered from p_1 to p_n.

In this paper, all group-messages are delivered in the order they were sent: "fifo" or "sender-ordered" property. This is the basic guarantee provided by the system.

Ensemble follows the Virtual Synchrony (VS) model. This model describes the relative ordering of message deliveries and view notifications. It is useful in simplifying complex failure and message loss scenarios that may occur in distributed environments. For example, a system adhering to VS ensures "atomic failure". If process q in view V fails then all the members in $V \setminus \{q\}$ observe this event at the "same time".

In particular, virtual synchrony is a strong enough model to support replicated data and consistency in a distributed environment [13, 26]. Our protocols (described later) employ these guarantees to consistently replicate complex data structures. Furthermore, our algorithms strongly depend on the guarantee that views are agreed. This is important in choosing agreed group leaders. The VS guarantee is used to agree an all-or-nothing property whereby either all group members receive the new group key, or none of them do.

To achieve fault-tolerance, GCSs require members to actively participate in failure-detection, membership, flow-control, and reliability protocols.

Therefore, such systems have inherently limited scalability. We have managed to scale Ensemble, the GCS with which the work reported here was undertaken, to a hundred members per group, but no more. For a detailed study of this problem, the interested reader is refered to [12, 2, 27]. In this paper, we do not discuss configurations of more than a hundred members. However, given a GCS capable of supporting larger groups our algorithms are intended to scale at least as well.

We assume processes in a 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 sign messages. Our system currently uses the PGP [24] authentication system. Kerberos [1] is also supported, though our interface is out of date.

As noted earlier, in a secure group, protection is afforded to group members through a shared key, with which all group messages are MAC-ed and encrypted. Accordingly, a secure group should support:

- Authorization and access control. Only trusted and authorized members are allowed into the group.

- Perfect Forward Secrecy (PFS). Briefly, this means that past members cannot obtain keys used in the future. The dual requirement that current members cannot obtain keys used in the past is termed Perfect Backward Secrecy (PBS).

To support PFS, the group key must be switched every time members join and leave. This may incur high cost, therefore, we would like to relax this requirement without breaching safety. To this end we do two things:

- The system rekeys itself every 24 hours

- Rekeying becomes a user initiated action. Hence, the user can trade off security against performance.

Our main goal in this paper is to improve rekeying performance, allowing a better tradeoff for the user. In the next subsection we briefly describe how authorization and access control are enforced. For more details the interested reader is refered to [22].

2.1 Authorization and Access Control

A secure GCS allows only authenticated and authorized members into a group. To support this, Ensemble makes use of the *Exchange* protocol.

This subsection provides a brief overview of this protocol and its security guarantees.

All members in a group component share a single group key. If a group is split into several components, then different components share different keys. To facilitate the merging of such components, a protocol that achieves secure key-agreement is required. Once key-agreement is achieved, members in different components can "hear" each other, and subsequentally merge.

Note that in the case where a group has just a single member, this merge protocol solves the problem of authorizing the join of a new process in an already running system.

To explain the key-agreement protocol, we assume two group components: P and Q with leaders p, q, and keys K_P, K_Q respectively. We assume all members agree on some 1024-bit integer modulos n and generator g for group Z_n, and that member p has chosen a random number $a \in Z_n$, and q chose a random number $b \in Z_n$. We omit the modulo n operations from the arithmetic used below to simplify notation. Each leader multicasts an $ImAlive$ message at fixed intervals. At some time point q and p receive each other's $ImAlive$ message. In what follows the notation $m_x \rightarrow m_{y,z} : M$ is used to show that member x sends to members y and z message M. The member with the lexicographically smaller name ,w.l.o.g q, engages in the following protocol:

Exchange protocol:

- $p \rightarrow q : ImAlive$
- q checks that p is allowed to join Q. If so, it signs and sends:
  $q \rightarrow p : \{q, p, g^b\}$
- p checks the signature on the received message. It also checks that q is allowed to join P. If so it signs and sends:
  $p \rightarrow q : \{p, q, g^a, \{K_P\}g^{ab}\}$
- q checks that p is allowed to join Q, and verifies the signature. If both conditions hold it sends:
  $q \rightarrow Q : \{K_P\}K_Q$

In essence, this is an authenticated Diffie-Hellman exchange, where p's key is passed to q's component, and members of Q switch their key to K_P. The full implementation uses nonces to protect the protocol from reply

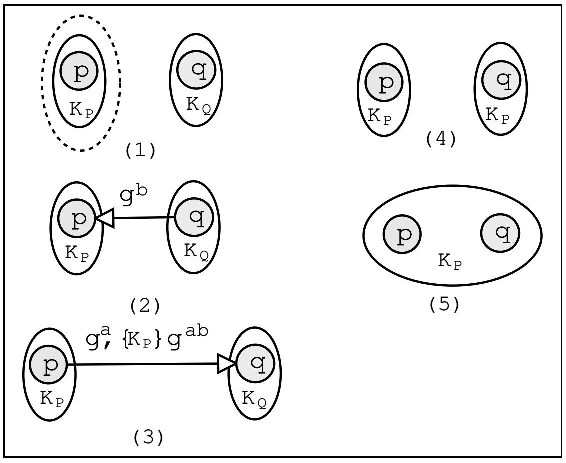

Figure 1: Overview of the merge sequence. (1) Two components with keys K_P and K_Q. Each leader sends ImAlive messages. (2) q sends g^b to p. (3) p sends g^a and K_P encrypted with g^{ab} to q. (4) q decrypts K_P and switches his component key to K_P. (5) Components P and Q now merge, after both have the same key.

attacks. It also allows multiple concurrent runs of Exchange. Security of this protocol draws from the security of the Diffie-Hellman exchange, and from the security of the Bellare-Rogaway protocol [17].

In this protocol, the leader is entrusted with checking the group Access Control List (ACL). This guarantees only a certain level of authorization checking. If leaders p and q are in each other's ACLs then components P and Q will merge. This assumes that the ACL is symmetric and transitive, i.e., an equivalence relation. While it is possible to allow each member in P to check all members in Q, and vice-versa, we have decided this was too expensive in terms of communication and computation.

In Ensemble, it is up to the application developer to make sure that the ACL is in fact an equivalence relation. While this may sound impractical, there are simple ways of implementing this behavior. For example, the system could employ a centralized authorization server, or simply a static ACL. Such an ACL must separate the list of trusted hosts into several disjoint subgroups. Trust is complete in each subgroup, but no member need trust members outside its subgroup.

Our system allows applications to dynamically change their ACL, however, this may temporarily break the equivalence relation. For example, it is possible that ,temporarily, member p trusts q, q trust s but p does not trust s. This may allow the creation of a group $\{p, q, s\}$ where not all members trust each other. Therefore, care is required while changing ACLs, and it is up to the application to make sure such inconsistencies do not occur.

2.2 Secure Groups

Henceforth, we assume that the system creates only secure groups where:

- All members are trusted and authorized
- All members have the same group-key.
- No outside member can eavesdrop, or tamper with group messaging.

In the remainder of this paper, we make extensive use of secure point-to-point communication. Although public keys can be used for this purpose, their poor performance motivates us to employ symmetric point-to-point keys where such communication may occur repeatedly. Accordingly, we introduce a *secure channel* abstraction. A secure channel allows two members to exchange private information, it is set up using a Diffie-Hellman [33] exchange. The initial exchange sets up an agreed symmetric key with which subsequent communication is encrypted. The set of secure channels at member p is managed using a simple caching scheme: (1) only connections to present group members are kept. (2) the cache is flushed every 24 hours[2] to prevent cryptanalysis.

Integer module exponentiations, needed for a Diffie-Hellman exchange, are expensive. We used a 500Mhz PentiumIII with 256Mbytes of memory, running the Linux2.2 OS for speed measurements. An exponentiation with a 1024bit key using the OpenSSL [18] cryptographic library was clocked at 40 milliseconds. Setting up a secure channel requires two messages containing 1024bit long integers. Hence, we view the establishment of secure channels as expensive, in terms of both bandwidth and CPU. Our caching scheme efficiently manages channels, but in what follows, an important goal will be to minimize the need of their use.

2.3 Liveness and Safety

Ideally, one would hope that distributed protocols can be proven *live* and *safe*. Key management protocols must also provide *agreement* and *authen-*

[2]This is a settable parameter.

ticity properties. Here we define these properties, 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.

To show how a rekeying protocol can comply with the above four requirements, we introduce a simplistic rekeying protocol, $\mathcal{P}$.

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.

3) Each group member, once it receives the group-key from the leader, sends an acknowledgment (in the clear) to the leader.

4) The leader, once it receives acknowledgments from all group members, multicasts a *ProtoDone* message.

5) A member that receives a *ProtoDone* message knows that the protocol has terminated and the new key can now be used.

$\mathcal{P}$ is safe since it uses secure channels to group members, all of which are trusted. It satisfies agreement because a single agreed member acts as leader. It satisfies authenticity because only a trusted authorized member can become the leader, and only the leader can choose a new key.

However, $\mathcal{P}$ is not live. Notice that the protocol requires all processes to receive the new-key and send acknowledgments. If some member fails and never recovers during the execution, protocol $\mathcal{P}$ blocks.

To make the protocol fault-tolerant we restart the protocol in case of a view change. Note that restarting the protocol in case of a view-change is not enough to guarantee liveness. In fact, no protocol solving this class of problems can guarantee liveness in an asynchronous networking environment (see FLP [7]). 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.

This short exposition has shown that the four requirements listed above, while fundamental for a rekeying protocol, are easily satisfied (to the degree possible) for a protocol using the services of a Group Communication System. The critical ingredients are the ack-collection stage, and restarting. Such stages can be added to *any* rekeying protocol. Hence, in the more complex protocol described later on in sections 4, and 5 we omit these stages.

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

Here we describe a protocol created by Wong et al. [4], and Wallner et al. [6]. Improvements were later suggested in [5, 25]. The general idea is termed a *Logical Key Hierarchy*, and it uses the notion of *key-graphs*. Key-graphs have been put to use in other security protocols, for example, Cliques [36].

A key-graph 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 chosen and distributed by a key-server. While the general notion of a key-graph is somewhat more general, we focus on binary-trees. In Figure 2 we see a typical key-graph for a group of eight members. This will serve as our running example.

Each member p_i shares a key with the server, K_i, using a secret channel. This key is the secure-channel key used for private communication between p_i and the server. It is called the *basic-key*. Each member also knows all the keys on the path from itself to the root. The root key is the group key. In the example, member p_1 knows keys $K_1, K_{12}, K_{14}, K_{18}$. It shares K_1 with the server, K_{12} with p_2, K_{14} with $\{p_2, p_3, p_4\}$, and K_{18} with members $\{p_2, \dots p_8\}$. In what follows we denote by $\{M\}_{K_1,K_2}$ a tuple consisting of message M encrypted once with key K_1, and a second time with K_2.

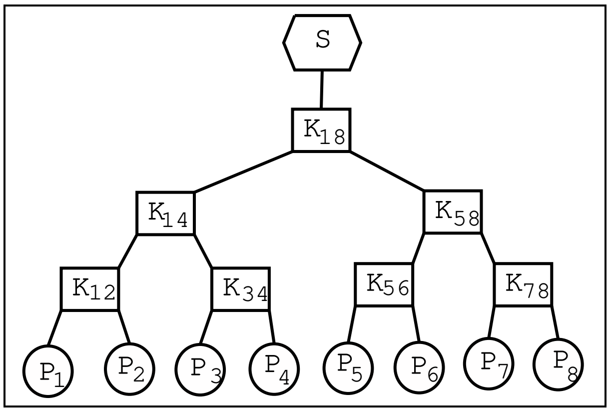

Figure 2: A key-graph for a group of eight members.

The key server uses the basic keys to build the higher level keys. For example, the key-graph in the figure can be built using a single multicast message comprised of three parts:

- Part I: $\{K_{12}\}_{K_1,K_2}$, $\{K_{34}\}_{K_3,K_4}$, $\{K_{56}\}_{K_5,K_6}$, $\{K_{78}\}_{K_7,K_8}$
- Part II: $\{K_{14}\}_{K_{12},K_{34}}$, $\{K_{58}\}_{K_{56},K_{78}}$
- Part III: $\{K_{18}\}_{K_{14},K_{58}}$

All group members receive this multicast, and they can retrieve the exact set of keys on the route to the root. For example, member p_6 can decrypt from the first part (only) K_{56}. Using K_{56}, it can retrieve K_{58} from the second part. Using K_{58} it can retrieve K_{18} from the third part. This completes the set K_{56}, K_{58}, K_{18}. In general, it is possible to construct any key-graph using a single multicast message.

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

Join: Assume member p_9 joins the group. S picks a new (random) group-key K_{19} and multicasts: $\{K_{19}\}_{K_{18},K_9}$. Member p_9 can retrieve K_{19} using K_9, the rest of the members can use K_{18} to do so.

Leave: Assume member p_1 leaves, then the server needs to replace keys K_{12}, K_{14}, and K_{18}. It chooses new keys K_{24}, and K_{28} and multicasts a two part message:

- Part I: $\{K_{24}\}_{K_2,K_{34}}$

- Part II: $\{K_{28}\}_{K_{24},K_{58}}$.

The first part establishes the new key K_{24} as the subtree-key for members p_2, p_3, p_4. The second part establishes K_{28} as the group key.

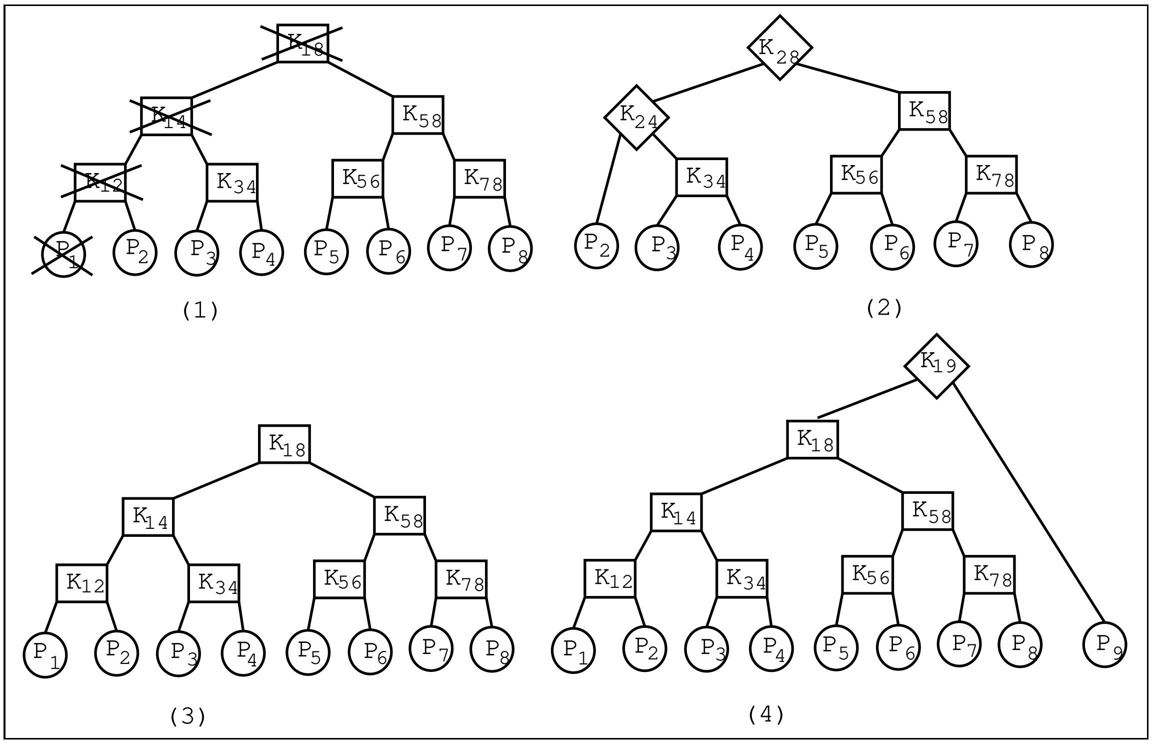

Figure 3: Examples for join/leave cases. Examples one and two depict a scenario where member p_1 leaves. Examples three and four depict a scenario where member p_9 joins. New keys are marked by diamonds.

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. The protocol costs exactly one multicast message (acknowledgments are not discussed here). The message size, as a multiply of the size of a key K, is as follows:

Construction	Leave	Join
$2Kn$	$2K\log_2 n$	$2K$

Trees become imbalanced after many additions and deletions, and it becomes necessary to rebalance them. A simple rebalancing technique is described in [19]. However, their scheme is rather rudimentary. For example, in some full binary key-tree T, if most of the right hand members leave, the tree becomes extremely unbalanced. The simple scheme does not handle this well. Our scheme can handle such extreme cases.

3.1 Tree balancing

After multiple leaves, a key-graph is composed of a disjoint set of sub-key-graphs. More generally, setting up the distributed version of the protocol, we shall be interested in merging different key-graphs. For example, in Figure 4(1), one can see two key graphs, T_1 that includes members $\{p_1, \ldots, p_5\}$, and T_2 that includes members $\{p_6, p_7\}$. The naive approach in merging T_1 and T_2 together is to add members $\{p_6, p_7\}$ one by one to T_1. However, as we can see in Figure 4(2) this yields a rather unbalanced graph. Instead, we create a new key K_{57} and put T_2 under it.

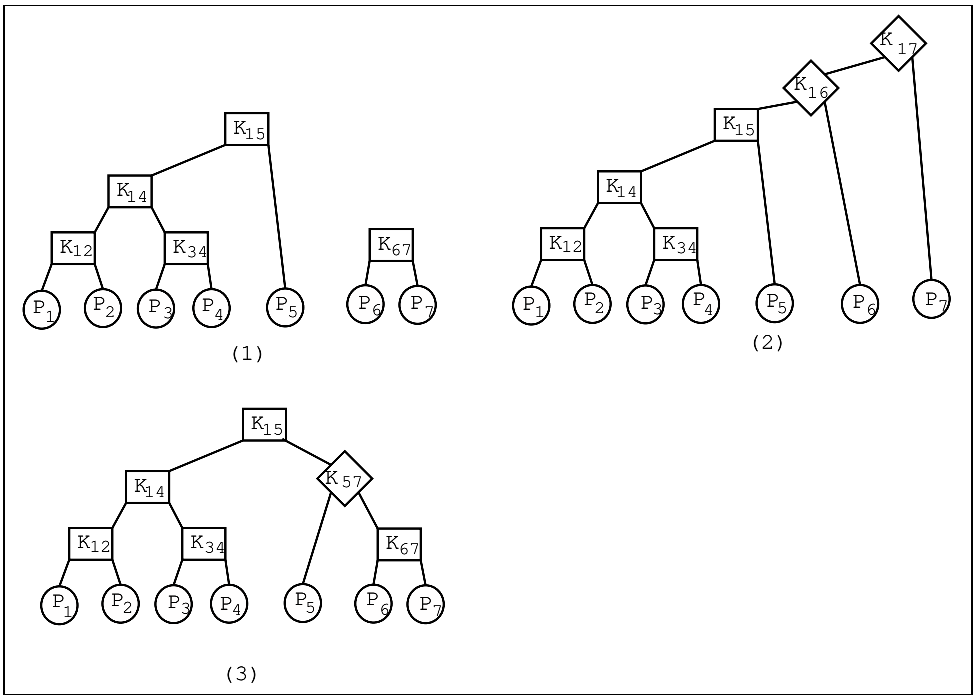

Figure 4: Balancing example.

Merging two trees in this fashion can be performed using one message sent by the key-server. The message includes two parts:

- Part I: The new key, K_{57} encrypted for both of its children: $\{K_{57}\}_{K_{67},K_5}$
- Part II: All keys above T_2 encrypted with K_{57}: $\{K_{15}\}_{K_{67}}$

The example can be generalized to a *balanced merge* (Bal_Merge) procedure. In what follows we denote by $h(T)$ the height of tree T. To describe the structure of a tree we use a recursive data structure that contains nodes of two kinds:

- $Br(L, K, R)$: An inner-node with left subtree L, right subtree R, and key K.
- $Leaf(p_i)$: A leaf node with member p_i.

The simplest tree contains a single member, $Leaf(p_x)$. A single member tree does not have a key, since a single member can communication with itself alone. Such communication does not need protection. The height of a tree $Leaf(p_x)$ is 1.

Assume that L and R are two subtrees that we wish to merge. W.l.o.g we can assume that $h(L) \geq h(R)$. Bal_Merge will find a subtree L' of L such that $1 \geq h(R) - h(L') \geq 0$, and create a new tree T where L' is replaced by $Br(L', K_{new}, R)$. The merge procedure is performed at the key-server, that has complete knowledge of all keys and the structure of existing key-trees. The local computation is a simple tree recursion. The message size required to notify the group is of size $O(\log_2 n)$. This is because R's members must know all keys above K_{new}.

In a general key-graph, there is no bound on depth as a function of the number of leaves. If a member that is situated very deep in the tree is removed, then the tree is split into many disconnected fragments. Therefore, it is desirable to have balanced trees. We propose using AVL trees for this purpose. Assume that T_1 and T_2 are AVL trees. We shall show that the merged tree is also AVL. First, we state more carefully the Bal_Merge algorithm, in the form of pseudo-code.

Algorithm Balanced Merge(T_1, T_2):

```
if h(T1) == h(T2) then {
  choose a new key K_new
  return Br(T1, K_new, T2)
}
if h(T1) > h(T2) then {
  assume T1 = Br(L, K_T1, R)
  if h(L) < h(R) then return (Br(merge(L, T2), K_T1, R))
  else return Br(L, K_T1, merge(T2, R))
}
if h(T1) < h(T2) then {
  the same as above, reversing the names T1, T2.
}
```

Claim 1 *: Merging two AVL trees T_1 and T_2 results in an AVL tree whose height is at most $1 + \max(h(T_1), h(T_2))$.*

Proof: The proof is performed by induction on the difference in height: $h(T_1) - h(T_2)$.

- Base Case: If $h(T_1) = h(T_2)$ then the merged tree is $T = Br(T_1, K_{new}, T_2)$. The depth of the T is $h(T_1) + 1$.

- Induction: We assume that if $h(T_1) - h(T_2) < k$ then the theorem holds. W.l.o.g $h(T_1) > h(T_2)$. If $h(T_1) - h(T_2) = k$ then T_1 is of height at least two, and $T_1 = Br(L, key, R)$. There are two cases:

  - $h(L) < h(R)$: The result will be, $T = Br(merge(T_2, L), key, R)$ By induction we have that $merge(T_2, L)$ is AVL, and that its height is at most $h(R) + 1$.

    Hence, the difference in height between the left and right children of T is no more than 1, and T's height conforms to $h(T) \leq h(R) + 2 = 1 + max(h(T_1), h(T_2))$

    $h(L) \geq h(R)$: The result will be $T = Br(L, key, merge(T_2, R))$. By induction we have that $merge(T_2, R)$ is of height that is at most $h(L) + 1$.

    Hence, the difference in height between the left and right children of T is no more than 1, and T's height conforms to $h(T) \leq h(L) + 2 = 1 + max(h(T_1), h(T_2))$

□

Two AVL trees can be merged into a single AVL tree at low cost. This can be generalized to merging a set of trees. Sets of trees are interesting since they occur naturally in leave operations. For example, if a member leaves an AVL tree T, then T is split into a set of $\log_2 n$ subtrees. These subtrees, in turn, are AVL and they can be merged into a new AVL tree.

The cost of a merge operation is measured by the size of the required multicast message. We are interested in the price of merging m AVL trees, with n members. First, we shall use a naive but imprecise argument to show that the price is bounded by $2K(m-1) + K(max\{h(T_i)\} - min\{h(T_i)\})$. Then we shall provide a proof showing this to be in fact $3K(m-1) + K(max\{h(T_i)\} - min\{h(T_i)\})$.

We sort the trees by height into a set $\{T_1, T_2, \ldots, T_m\}$. They are merged two at a time: first the smallest two into T_{12}, then T_{12} and T_3 into T_{13} etc. The price for merging two trees is $2K + K\|h(T_1) - h(T_2)\|$. Assuming naively that $h(\mathsf{Bal\_Merge}(T_x, T_y)) = \max(h(T_x), h(T_y))$ then the total price is the height difference between the deepest tree, and the shallowest one. Hence, the total price is: $2K(m-1) + K(max\{h(T_i)\} - min\{h(T_i)\})$.

The naive argument fails to take into account the fact that merged trees conform to $h(\mathsf{Bal\_Merge}(T_x, T_y)) \leq 1 + \max(h(T_x), h(T_y))$. Therefore, we insert the trees into a heap. Iteratively, the smallest two trees are poped from the heap, merged together, and put back into the heap.

Claim 2 : *The cost of a balanced merge of m trees $\mathcal{T}_m = \{T_1, \ldots, T_m\}$ is bounded by $3K(m-1) + K(max\{h(T_i)\} - min\{h(T_i)\})$.*

Proof: The proof is by induction. For two trees T_1 and T_2 , the price is simply $2K + K\|h(T_1) - h(T_2)\|$.

Assume the claim is true for up to $m-1$ trees. We shall show for m. We proceed according to the heap discipline and merge the two smallest trees, T_1 and T_2 into $T_{1\uplus 2}$. We get a heap with trees $\mathcal{T}_{m-1} = \{T_{1\uplus 2}, T_3, \ldots, T_m\}$.

The cost for merging the trees in $\mathcal{T}_{m-1}$ is, by induction, $3K(m-2) + K(max\{h(T_i)\}_{\mathcal{T}_{m-1}} - min\{h(T_i)\}_{\mathcal{T}_{m-1}})$. The cost for merging T_1 and T_2 is $K\|h(T_1) - h(T_2)\| + 2K$.

We know that:

1) $max\{h(T_i)\}_{\mathcal{T}_{m-1}}$ is bounded by $max\{h(T_i)\}_{\mathcal{T}_m} + 1$.

2) $min\{h(T_i)\}_{\mathcal{T}_{m-1}} \geq min\{h(T_i)\}_{\mathcal{T}_m} + \|h(T_1) - h(T_2)\|$

Hence, the sum:

$K(max\{h(T_i)\}_{\mathcal{T}_{m-1}} - min\{h(T_i)\}_{\mathcal{T}_{m-1}}) + 3K(m-2) + 2K + K\|h(T_1) - h(T_2)\|$

is bounded by:

$$K((max\{h(T_i)\}_{\mathcal{T}_m} - min\{h(T_i)\}_{\mathcal{T}_m}) + 1) + 3K(m-2) + 2K =$$
$$= \quad K(max\{h(T_i)\}_{\mathcal{T}_m} - min\{h(T_i)\}_{\mathcal{T}_m}) + 3K(m-1)$$

□

In Figure 5 we can see a balanced tree, after a join.

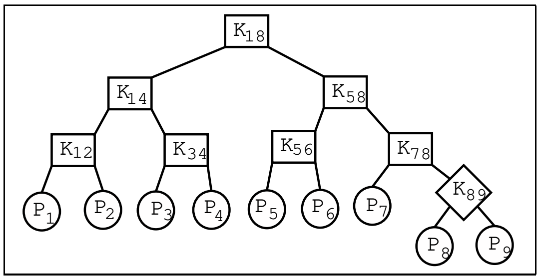

Figure 5: Balancing the join case.

The table below lists the costs of balanced join and leave events. This shows that AVL trees do not cost significantly more than standard key-graphs. We shall use them henceforth.

Construction	Leave	Join
$2Kn$	$2K \log_2 n$	$K \log_2 n$

Note that the technique described so far does not maintain Perfect Backward Secrecy (PBS). For example, in the join case, member p_9 learns the set of keys used before by members $\{p_1, \ldots p_8\}$. To guaranty PBS, we must replace all keys on the path from the joiner to the root with fresh keys.

Currently, we do not address the issue of PBS in full. However, we allow the application to determine a time period after which the whole tree is discarded and built from scratch. By default this timer is set to 24 hours. Hence, PBS is guaranteed up to 24 hours.

4 The basic ($\mathcal{B}$) distributed protocol

The centralized solution $\mathcal{C}$ does not satisfy our objectives because it relies on a centralized server which has knowledge of the complete key-graph. We require a completely distributed solution, without a single point of failure. While our algorithm is based on $\mathcal{C}$, each member keeps only $\log_2 n$, keys and members play symmetric roles. The algorithm presented here is called Basic ($\mathcal{B}$) since it is rather simple and inefficient. We optimize it in the next sections.

We shall say that a group of members G has key-graph T if the set of leaf-nodes that T contains is equal to G. Furthermore, each member of G must posses exactly the set of keys on the route from itself to the root of T. The leader of T, denoted C_T, is the leaf that is the left-most in the tree. Since members agree on the tree structure, each member knows if it is a leader or not. We denote by $M(T)$ the members of T, and by K_T the top key in T (if one exists).

We can see that members in G must have routes in the tree that "match", i.e., when pulled together they make a tree that is the same as a tree obtained by the centralized algorithm. To perform this 'magic" in a distributed environment we rely on Virtual Synchrony (see more below 4.1.

To motivate the general algorithm, we shall describe a naive protocol that establishes key-trees for groups of size 2^n. We use the notion of subtrees *agreeing* on a mutual key. Informally, this means that the members of two subtrees L and R, securely agree on a mutual encryption key. The protocol used to agree on a mutual key is as follows:

Agree(L, R):

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

The *agree* primitive costs one point-to-point and two multicast messages. Its effect is to create the key-graph $Br(L, K_{LR}, R)$ for athe members of $M(L) \cup M(R)$.

Below is an example for the creation of a key-tree for a group of 8 members (see Figure 6):

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 occur concurrently. In this case, the algorithm takes 3 rounds, and each member stores 3 keys. This protocol can be generalized to any number of members.

Base case: If the group contains 0 or 1 members, then we are done.

Recursive step ($n = 2^{k+1}$): Split the group into two subgroups, M_L and M_R, containing each 2^k members. Apply the algorithm recursively to M_L and M_R. Now, subgroup M_L possesses key-tree L, and subgroup M_R posseses key-tree R. Apply the agreement primitive to L and R such that they agree on a group key.

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

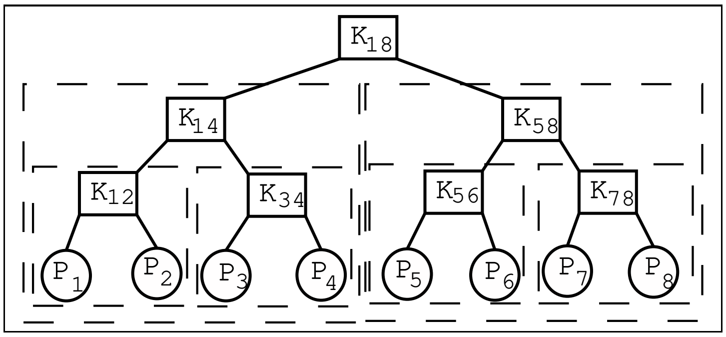

Figure 6: The naive protocol for a group of eight members. Groups that agree on a key are surrounded by a dashed rectangle.

Clearly, the naive protocol builds key-graphs for groups of members of size 2^n. Note that no member holds more than $\log_2 n$ keys, and the total number of keys is n.

In the lifetime of the group members join, and leave, the group partitions and merges with other group components. For example, when member p_1 leaves, the key-graph is split into three pieces. When p_9 joins, the new key-tree is a merging of the key-graphs containing $\{p_1, \ldots, p_8\}$, and $\{p_9\}$. To handle these events efficiently, the computation of the new key-tree, as a function of existing subtrees, is performed by the group leader. After a view-change, the leader requests subtree *layouts* from subleaders. A tree *layout* is a symbolic description of a key-tree. The leader combines the subtree layouts and multicasts a new complete layout, coupled with a schedule describing how to merge subtrees together.

Note that the leader is not allowed to learn the actual keys used by other members. This is the reason for using symbolic representations.

We now extend *agree* to conform to the merge step described in the previous section. If L and R are two subtrees that should be merged together then, assuming $h(L) \geq h(R)$, the leader finds a subtree L' of L such that $1 \geq h(R) - h(L') \geq 0$. The protocol is as follows. $C_{L'}$ denotes the leader within L', C_R the leader within R, C_T the tree leader. The algorithm as a whole is initiated by C_T.

Extended Agree(L, R):

1. C_T initiates the protocol by multicasting the new layout and schedule.
2. $C_{L'}$ chooses a new key $K_{L'R}$, and sends it to C_R using a secure channel.
3. $C_{L'}$ encrypts $K_{L'R}$ with $K_{L'}$ and multicasts it.
   $C_{L'}$ encrypts all keys above $K_{L'R}$ with $K_{L'R}$ and multicasts them.
   C_R encrypts $K_{L'R}$ with K_R and multicasts it.
4. All members of $L \cup R$ securely receive the new key.

Below are two examples, the first is a leave scenario, the second is a join scenario. Both reference Figure 3.

In figures (1) and (2), member p_1 fails. All members receive a view-change notification, and locally erase from their key-trees all keys of which p_1 had knowledge. The key-graph has now been split into three pieces T_1, T_2, T_3 containing: $\{p_2\}, \{p_3, p_4\}, \{p_5, p_6, p_7, p_8\}$ respectively. The leader collects the structures of T_1, T_2, and T_3 from the subleaders p_2, p_3, and p_5. It decides that the merged tree will have the layout described in Figure 7(2). It multicasts this structure to the group, with instructions for T_1 to merge with T_2 into T_{12}, and later for T_{12} to merge with T_3.

In figures (3) and (4) member p_9 joins the group. All members receive a view-change that includes member p_9. The key-graph is now comprised of two disjoint components: T_1 containing: $\{p_1, \ldots, p_8\}$, and T_2 containing

$\{p_9\}$. The leader collects the structures of T_1 and T_2 from the subleaders p_1 and p_9. It decides that the merge tree will have the layout described in Figure 7(4). It multicasts this structure to the group with instructions for T_1 to merge with T_2.

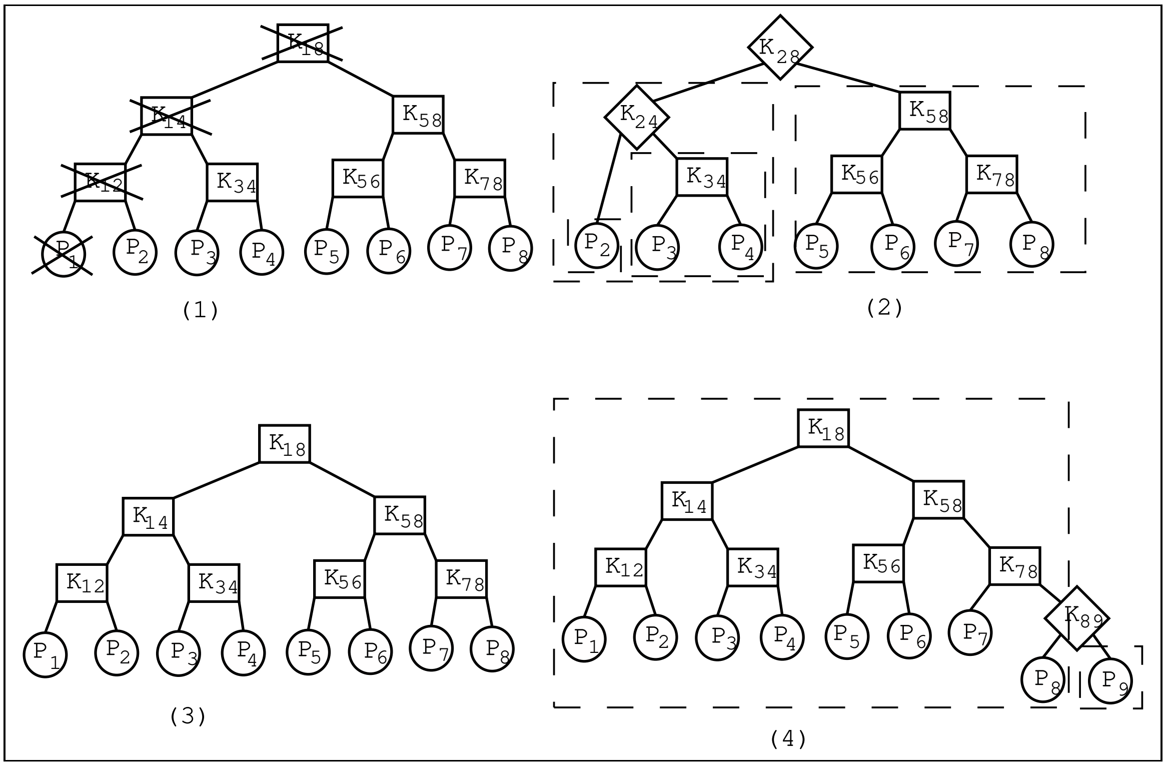

Figure 7: Single member join/leave cases. Examples (1) and (2) show the actions performed when member p_1 leaves the group. Examples (3) and (4) show the actions performed when member p_9 joins.

Protocol $\mathcal{B}$ takes 2 stages in case of join, $\log_2 n$ stages in case of leave, and $\log_2 n$ in case of tree construction. In general, the number of rounds required is the maximal depth of the set of new levels in the tree that must be constructed. The optimized protocol improves this result to two communication rounds, regardless of the number of levels that must be constructed.

4.1 Virtual Synchrony

Protocol $\mathcal{B}$ makes use of the GCS guarantees on message delivery, and view agreement. All point-to-point and multicast messages are reliable, and sender-ordered. Furthermore, they arrive in the view they are sent, hence, an instance of the protocol is run in a single view. If the view changes, due to members leaving or joining, then the protocol is restarted.

The guarantee that members agree on the view is crucial. It allows members to know what role they play in the protocol, be it leader, sub-leader, or no-role.

Although the protocol is multi-phased, we can guarantee an *all-or-nothing* property. Either all members receive the new key-tree, and the new group-key, or the protocol fails (and all members agree on this). To see this, examine the last step of the protocol, where $\mathcal{B}$ has been completed, and acknowledgments are collected by the leader. Once acknowledgment collection is complete, a *ProtoDone* message is multicast by the leader. Members that receive *ProtoDone* know that the protocol is over, and that other members also posses the same key-tree.

Should failures occur prior to the last multicast, then the protocol is restarted with the previous sub-trees. If a failure occurs afterwards, then remaining members agree on the key-tree and group-key. Failed members will be removed in the next view, and the protocol will be restarted.

This description is also valid for the optimized solution described in the next section.

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

First, we describe an example showing that $\mathcal{B}$ can be substantially improved. Then we describe the optimized algorithm $\mathcal{O}$.

Examine the case where member p_1 leaves the group. There are three components to merge, T_1, T_2, T_3 and two tree levels to reconstruct. This requires two *agree* rounds. However, we can transform the protocol and improve it substantially by having p_2 choose all required keys at the same time. See Figure 8(1).

The protocol used is as follows:

- Round I: p_2 chooses K_{24}, K_{28}. It sends K_{24} to the leader of T_2, K_{28} to the leader of T_3. Finally, it multicasts $\{K_{28}\}_{K_{24}}$ so that members of T_2 can retrieve K_{28}.

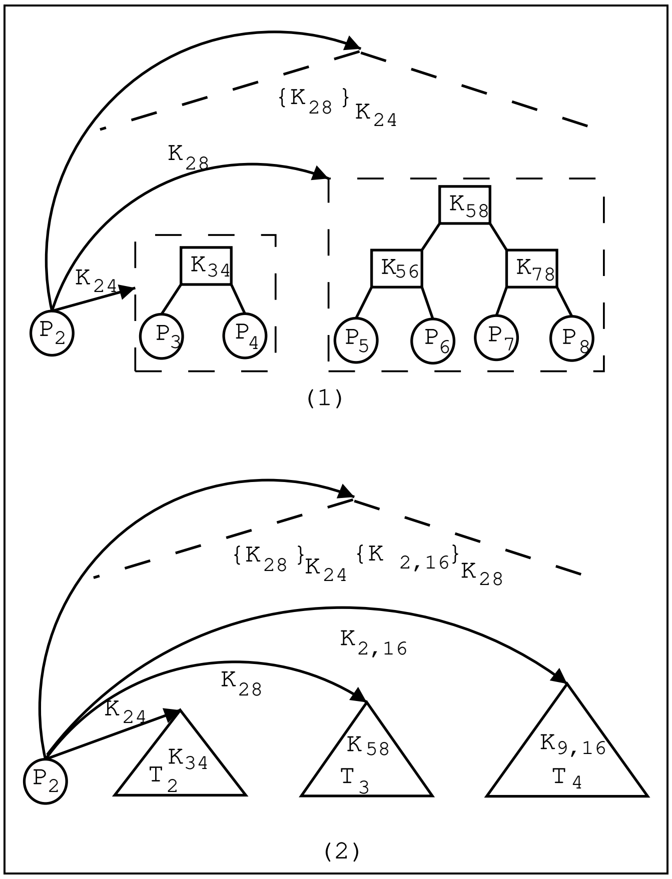

Figure 8: Optimizing the leave case.

- Round II: The leader of T_2 multicasts $\{K_{24}\}_{K_{34}}$.
  The leader of T_3 multicasts $\{K_{28}\}_{K_{58}}$.

A larger example is depicted in Figure 8(2). Here, in a 16 member group, p_1 leaves. The regular algorithm requires three agree rounds, costing six communication rounds. This can be optimized to cost only two communication rounds, similarly to the above example.

- Round I: p_2 chooses $K_{24}, K_{28}, K_{2,16}$. It sends K_{24} to the leader of T_2, K_{28} to the leader of T_3, and $K_{2,16}$ to the leader of T_4. Finally, it multicasts $\{K_{28}\}_{K_{24}}$, $\{K_{2,16}\}_{K_{28}}$, so that members of T_2 and T_3 can retrieve K_{28} and $K_{2,16}$.

- Round II: The leader of T_2 multicasts $\{K_{24}\}_{K_{34}}$.
  The leader of T_3 multicasts $\{K_{28}\}_{K_{58}}$.
  The leader of T_4 multicasts $\{K_{2,16}\}_{K_{9,16}}$.

Prior to presenting the general solution, we provide an example for the creation of a complete key-tree of a group of eight members (see Figure 9).

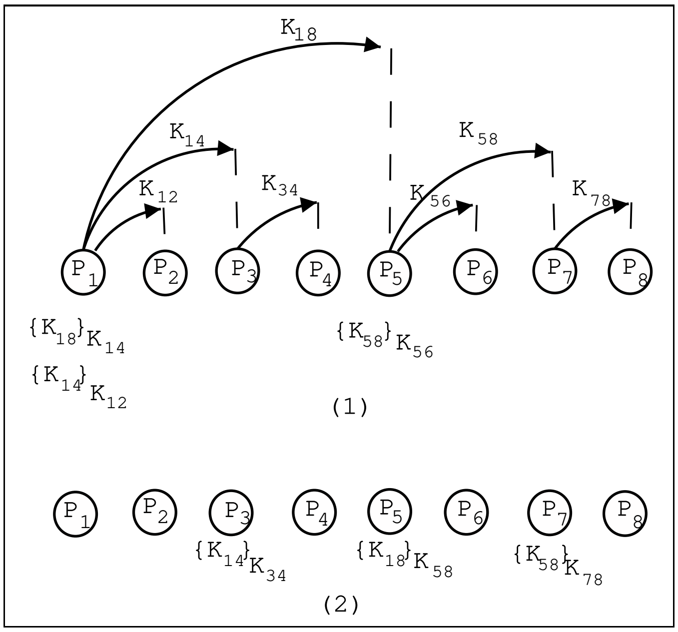

Figure 9: Tree construction for an eight member group.

In the first stage (Figure 9(1)), members engage in a protocol similar to the leave case. It is performed in parallel by all leaders in the group. New keys are denoted by full arrows. Multicasted keys are written underneath member nodes. In the second stage, members pass received keys along, using multicast.

Round 1:

p_1 chooses K_{12}, K_{14}, K_{18}.

$m_1 \rightarrow m_2 : K_{12}$

$m_1 \rightarrow m_3 : K_{14}$

$m_1 \rightarrow m_5 : K_{18}$

$p_1 \rightarrow G : \{K_{14}\}_{K_{12}},\ \{K_{18}\}_{K_{14}}$

p_3 chooses K_{34}.

$m_3 \rightarrow m_4 : K_{34}$

p_5 chooses K_{56}, K_{58}.

$m_5 \rightarrow m_6 : K_{56}$

$m_5 \rightarrow m_7 : K_{58}$

$p_1 \rightarrow G : \{K_{58}\}_{K_{56}}$

p_7 chooses K_{78}.

$m_7 \rightarrow m_8 : K_{78}$

Round 2:

$p_3 \rightarrow G : \{K_{14}\}_{K_{34}}$

$p_5 \rightarrow G : \{K_{18}\}_{K_{58}}$

$p_7 \rightarrow G : \{K_{58}\}_{K_{78}}$

At the end of the protocol, all members have all the keys. For example, member p_1 has all the keys at the end of the first round. It learns no other keys as the protocol progresses. Member p_8 receives in round one K_{78}. In round two it receives: $\{K_{18}\}_{K_{58}}$ and $\{K_{58}\}_{K_{78}}$. Thus, it can retrieve K_{58}, and K_{18}.

We now generalize the protocol. We shall describe how the leader, after retrieving subtree layouts $\{T_1, \ldots, T_m\}$, creates an optimal schedule describing how to merge the subtrees together. Since members simply follow the leader's schedule, we shall focus on the scheduling algorithm (Sched).

Sched works recursively. First, we use the centralized algorithm, and merge $T_1, \ldots, T_m$ together. There are $m - 1$ instances where merge operations occur, each such operation is translated using a simple template.

Assuming subtrees L and R are to be merged, and $h(L) \geq h(R)$, then there is a subtree L' of L with which to merge R. Mark the list of keys on the route from $K_{L'}$ to the root of L by L_{path}.

The template is:

Stage 1:

$C_{L'}$ choose a new key $K_{L'R}$ and:

$C_{L'} \rightarrow C_R : K_{L'R}$

$C_{L'} \rightarrow G : \{K_{L'R}\}_{K_{L'}}$, $\{L'_{path}\}_{K_{L'R}}$

Stage 2:

$C_R \rightarrow G : \{K_{L'R}\}_{K_R}$

Building a complete schedule requires performing Sched iteratively for $\{T_1, \ldots, T_k\}$ and storing the set of instructions in a data structure with two fields: one for the first stage, and another for the second stage. Each application of Sched adds two multicasts and one point-to-point message to the total. The number of rounds remains two.

An example for a more general case of Sched can be seen in Figure 10. The figure shows a set of three subtrees T_1, T_2, T_3 that need to be merged.

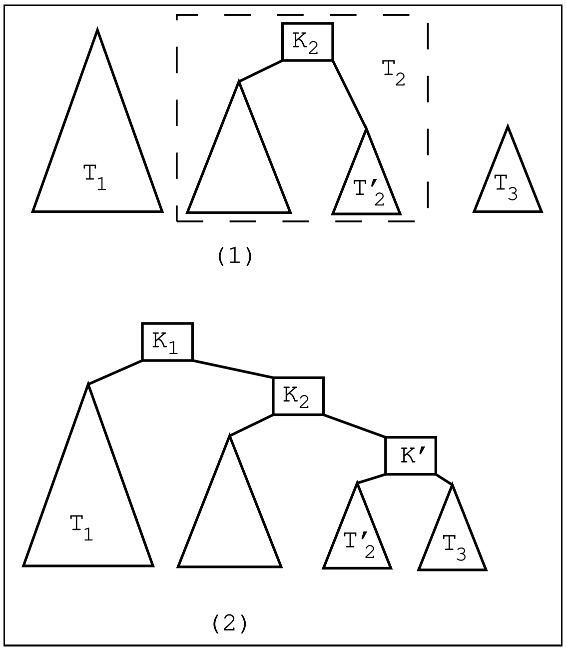

Figure 10: The leader of T_2' must send K_2 to T_3.

The resulting schedule will be:

Stage1:

C_{T_1} chooses K_1.

$C_{T_1} \to C_{T_2} : K_1$

$C_{T_1} \to G : \{K_1\}_{K_{T_1}}$

$C_{T_2'}$ chooses K'.

$C_{T_2'} \to C_{T_3} : K'$

$C_{T_2'} \to G : \{K'\}_{K_{T_2'}}$

$C_{T_2'} \to G : \{K_2\}_{K'}$

Stage2:

$C_{T_2} \to G : \{K_1\}_{K_{T_2}}$
$C_{T_3} \to G : \{K'\}_{K_{T_3}}$

For example, all members of T_3 will receive: $\{K'\}_{K_{T_3}}$ from C_{T_3}, $\{K_2\}_{K'}$ from $C_{T_2'}$, and $\{K_1\}_{K_2}$ from C_{T_2}. Using K_{T_3} they can decrypt K', K_2, and finally K_1. Hence, they can retrieve all the keys on the route to the root. Notice that $K_{T_2'}$ must send the path inside T_2 to all members of T_3. In this case, the path included only K_2.

Claim 3 *Algorithm* Sched *produces a schedule that guarantees all members receive exactly all the keys on the path to the root.*

Proof: The proof is performed by induction on the number of merge steps. If m trees need to be merged, then there are $m-1$ steps to perform. The induction invariant is that the leader of a subtree knows all keys on the path to the root of its subtree in stage 1.

Base case:

If m trees need to merged, then the base case is merging the first two.

For two trees L, R, we assume w.l.o.g that $h(L) \geq h(R)$. At the end of the protocol, C_R learns $K_{L'R}$, and the set of multicasts: $\{K_{L'R}\}_{K_{L'}}$, $\{L_{path}\}_{K_{L'R}}$, $\{K_{L'R}\}_{K_R}$ is sent. Members of R learn (exactly) $K_{L'R}$ and L_{path}. Members of L learn $K_{L'R}$, and the rest of the members learn nothing.

Note that the leader of L' chooses the new key $K_{L'R}$, hence the invariant holds.

Induction step:

Assume $k-1$ steps can be performed in two stages. Now we need to show for the k'th step. We perform the first $k-1$ steps, and get a tree L coupled with a two step schedule. The k'th step consists of scheduling a merge between the smallest tree in the heap R and L. Assume w.l.o.g that $h(L) \geq h(R)$, and that $h(R) = h(L')$, where L' is a subtree of L. The final tree will be L with L' replaced with $Br(L', K_{L'R}, R)$.

$C_{L'}$ chooses $K_{L'R}$ and passes it to C_R. It knows $K_{L'}$, hence it multicasts $\{K_{L'R}\}_{K_{L'}}$. All members of L' will know $K_{L'}$ at the end of stage two, hence they will also know $K_{L'R}$.

Leader C_R knows K_R in the first stage, hence, it multicasts $\{K_{L'R}\}_{K_R}$ in the second stage. By induction, all members of R will know K_R at the end of the second stage, hence they will also be able to retrieve $K_{L'R}$.

Furthermore, In the first stage, $C_{L'}$ multicasts $\{L_{path}\}_{K_{L'R}}$. This guarantees that members of R will know all the keys on the path to the root at the end of the second stage.

Note that the leader of L' chooses $K_{L'R}$, hence the invariant holds.

□

5.1 Three round solution ($\mathcal{O}_3$)

The optimized solution $\mathcal{O}$ is optimized for rounds, i.e., it reduces the number of communication rounds to a minimum. However, it is not optimal with respect to the number of 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 may have up to $O(n \times \log_2 n)$ multicast messages sent in a protocol run.

Here we improve $\mathcal{O}$ and create protocol $\mathcal{O}_3$. Protocol $\mathcal{O}_3$ is equivalent to $\mathcal{O}$ except for one detail, in each view, a member p_x is chosen. All subtree-leaders, in stage 2, send their multicasts messages point-to-point to p_x. Member p_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 p_x, and one multicast message by p_x. Hence, we add another round to the protocol but reduce multicast traffic.

5.2 Costs

Here, we compare the three-round solution with the regular centralized solution. Such a comparison is inherently unfair, since the distributed algorithm must overcome many obstacles that do not exist for the centralized version. To even out the playing ground somewhat, we do not take into account (1) collection of acknowledgments (2) sending tree-layouts. We compare three scenarios — building the key-tree, the join algorithm and the leave algorithm. 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. The following table summarizes the costs for each algorithm:

	# pt-2-pt	# multicast	# bytes	# rounds
$\mathcal{C}$	0	1	$2Kn$	1
$\mathcal{O}_3$	$1.5n - 1$	1	$3Kn$	3

Since the group contains n members, there are $n - 1$ keys to create. A single secure point-to-point message is used to create each key. There are $n/2$ members that act as subleaders. These all send multicast messages in the first and second stages. These messages are converted into point-to-point messages sent to the leader. All counted, there are $1.5n - 1$ point-to-point messages. The number of bytes is broken down as follows: (1) the $n - 1$ secure point-to-point messages cost Kn. (2) The messages to the leader cost a total of Kn (3) The final multicast bundles together all point-to-point messages, and it therefore costs an additional Kn. The cost is in fact Kn for the second and third items because the final multicast contains essentially all the group key-tree, except for the lowest level.

The leave algorithm costs:

	# pt-2-pt	# multicast	# bytes	# rounds
$\mathcal{C}$	0	1	$2K \log_2 n$	1
$\mathcal{O}_3$	$log_2 n$	1	$3K \log_2 n$	3

The join algorithm costs:

	# pt-2-pt	# multicast	# bytes	# rounds
$\mathcal{C}$	0	1	$\log_2 n$	1
$\mathcal{O}_3$	2	1	$\log_2 n$	3

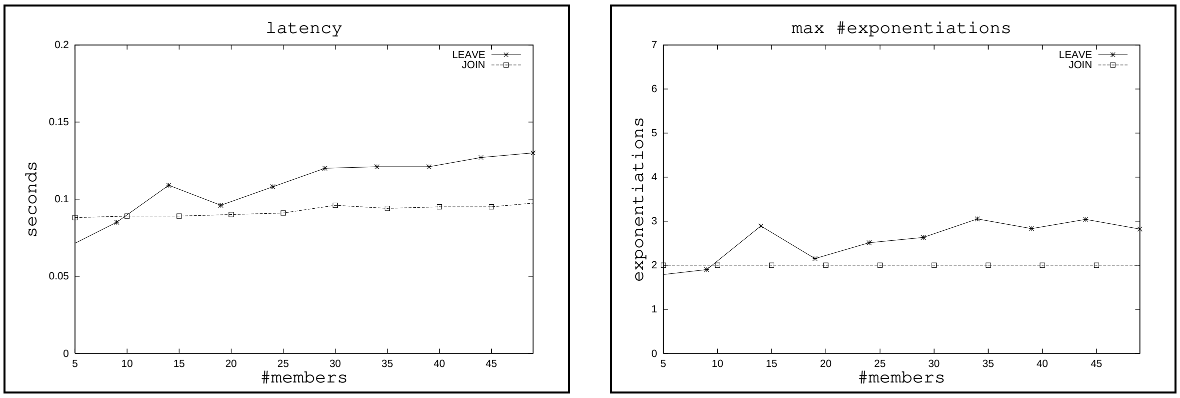

Figure 11: (1)Latency of the dWGL algorithm for join/leave operations.(2) The maximal number of exponentiations on the critical path.

6 Performance

We have fully implemented our algorithm in the Ensemble [30] group communication system. This section describes its performance. Our test-bed was a set of 20 PentiumIII 500Mhz Linux2.2 machines, connected by a switched 10Mbit/sec Ethernet. Machines were lightly loaded during testing. A group of n members was created, a set of member join/leave operations was performed, and measurements were taken. To simulate real conditions, we flushed the cache once every 30 rekey operations, and discarded "cold-start" results. To simulate more than 20 members, we ran a couple of processes on the same machine.

Figure 11(1) describes the latency of join/leave operations. Performance in the join case is optimal. There are exactly two integer exponentiations on the critical path. This is optimal. For example, if a new member is added on the left hand side of the tree, then that member must create connections to $\log_2 n$ components. The improvement comes from using a *tree-skewing* technique. When a single member joins, we attempt to add it as far to the right as possible. For example, if it becomes the right-most member, then only one connection will be required to connect it to the tree. Since we must also keep our tree AVL, we can keep skewing the tree until it reaches a state depicted in figure 12.

When a member leaves, naively, $\log_2 n$ components should be merged together. This should cost $2 \log_2 n$ exponentiations. Examining the latency

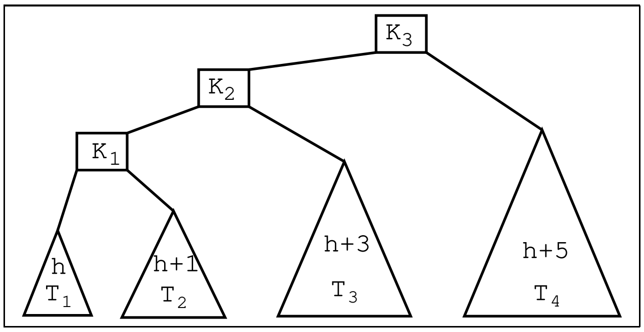

Figure 12: A skewed tree. The tree is comprised of four subtrees: T_1, T_2, T_3, and T_4 with heights $h, h+1, h+3$, and $h+5$.

graph, and Figure 11(2), we can see that very few exponentiations take place. This is due to the caching optimization, and due to tree-skewing. To see how tree-skewing is beneficial, examine Figure 12. Assume a member of T_1 leaves, and keys K_1, K_2, K_3 must be reconstructed. The leader of T_1 needs three secure connections in order to disseminate new keys. Had the tree been fully balanced, the depth of T_1 would have been substantially larger than three, incurring the creation of additional channels.

Figure 13(1) describes the size of the multicast message describing the layout of the group-tree. Figure 13(2) depicts the size of the round three multicast message containing the set of encrypted keys. Figure 13(2) depicts the (average) total number of exponentiations. For the join case, this is a constant two, whereas for the leave case, this slowly rises.

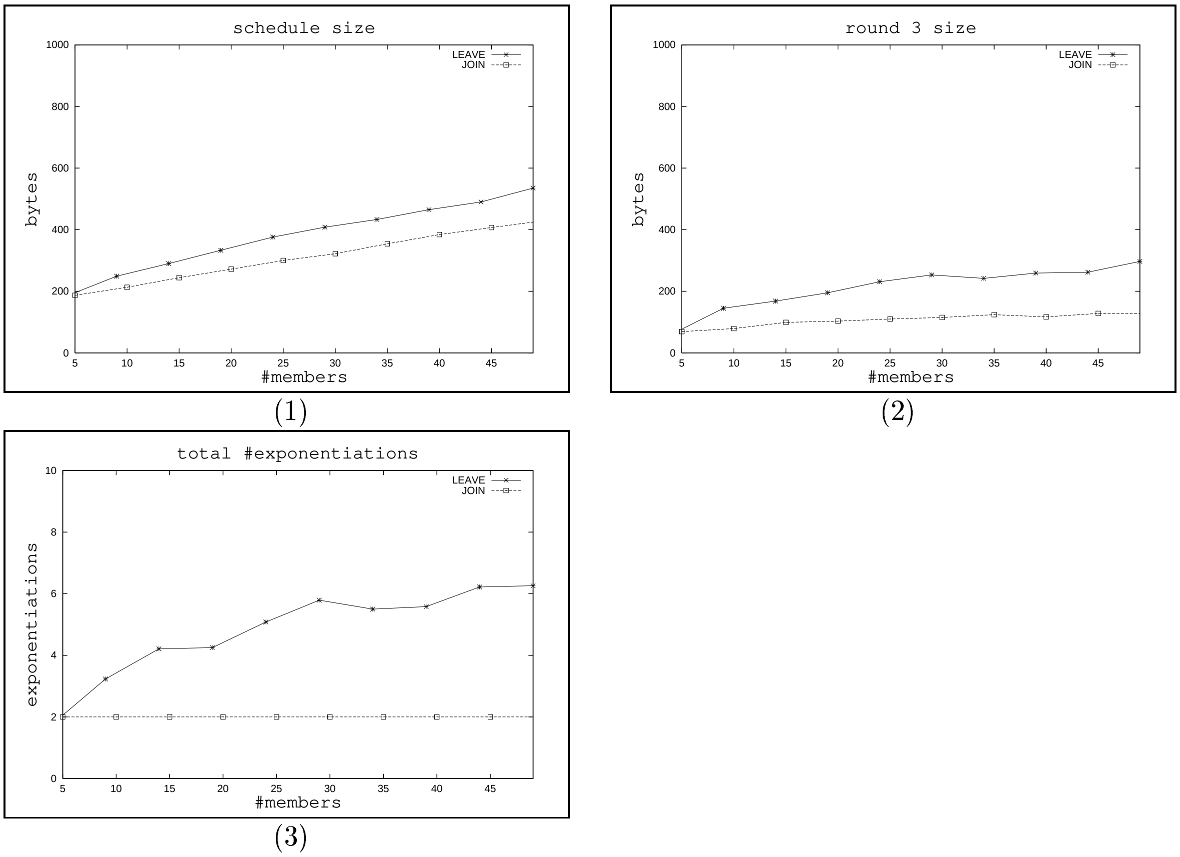

Figure 13: Additional performance measurements. (1) Size of the schedule sent by the leader. (2) Size of the bundled keys sent in the third round. (3) Total number of integer exponentiations.

To summarize, the protocol does not incur significant communication overhead, and the number of exponentiations is kept to a bare minimum. In some cases, the number of exponentiations is in fact optimal.

7 Related Work

Ensemble is a direct descendent of three systems: Isis [11], Horus [31], and Transis [34]. The Ensemble security architecture [22] has evolved from seminal work by Reiter [21, 29] done in the context of the Isis and Horus systems.

Many other GCSs have been built around the world. The secure GCSs that we know of are: Antigone [23], and Spread [35]. Spread splits the GCS functionality into a server and client sides. Protection, in the form of a shared encryption and MAC key, is offered to the client while the server is left unprotected. Access control is not supported. The shared group-key is created using Cliques cryptographic toolkit [20]. Cliques uses contributed shares from each member to create the group-key. Cliques's keys are stronger than our own, however, they require substantially more computation.

Antigone has been used to secure video conferences over the web, using the VIC and VAT tools. However, to date, it has not been provided with a fault tolerance architecture. The Totem [14], and Rampart [28] systems can survive Byzantine faults, at the cost of a further degradation of performance.

The Enclave system [9] allows a set of applications to create a shared security context in which secure communicate is possible. All multicast communication is encrypted and signed using a symmetric key. The security context is managed by a member acting as leader. Security is afforded to any application implementing the Enclave API. The Enclave system addresses the security concerns of a larger set of applications than our own, however, fault-tolerance is not addressed. Should the group-leader fail, the shared security context is lost and the group cannot recover.

The Cactus system [15] is a framework allowing the implementation of network services and applications. A Cactus application is typically split into many small layers (or *micro-protocols*), each implementing specific functionality. Cactus has a security architecture [16] that allows switching encryption and MAC algorithms as required. Actual micro-protocols can be switched at runtime as well. This allows the application to adapt to attacks, or changing network conditions at runtime.

Of all systems, ours is closest to Reiter's security architecture for Horus [29]. Horus is a group communication system, sharing much of the characteristics of Ensemble. The system followed the fortress security model, where a single partition was allowed, and members could join and leave the group, protected by access control, and authentication barriers. Group members share a symmetric group key used to encrypt and MAC all inner group messages. Furthermore, the system allocated public keys for groups, that clients could use to perform secure group-RPC. Horus was built at a time when authentication services were not standard, therefore, it included a secure time service, and a replicated byzantine fault-tolerant authentication service. Symmetric encryption was optimized through the generation of one-time-pads in the background.

By comparison, our system uses off-the-shelf authentication services, it does not handle group-RPC, and symmetric encryption is not a bottleneck.

In Ensemble we handle the "next tier" of issues: supporting efficient group merge (not just join and leave), allowing multiple partitions (not just primary partition), and efficient group rekeying with PFS (and a weak form of BFS).

8 Conclusions

We have described an efficient protocol for the management of group-keys for Group Communication Systems. Our protocol is based on the notion of key-graphs, or Logical Key Hierarchy, as originally suggested by Wong Gauda and Lam. We adapt and extend this work, making the protocol completely decentralized and fault-tolerant (to the extent possible), and employing a highly efficient tree balancing scheme.

In contrast, keygraphs were previously used in centralized settings, where a large group of clients is managed by a central server. The server builds a keygraph encompassing all the clients, allowing it to manage the set of clients. In a GCS, where no single point of failure is allowed, we could not afford to delegate the server-task to any single member.

Therefore, our protocol differs substantially from the centralized approach. Rather, members enlist in a collaborative effort to create the group key-graph. The surprising result is that the completely distributed solution has comparative performance to the centralized one.

An issue for future work is supporting Perfect Backward Secrecy efficiently. Currently, we support it in a weak sense.

Another problematic area is scalability to thousands of nodes. Our protocol relies on a Group Communication system, that can operate up to a hundred nodes. The protocol strongly relies on the agreed membership, and virtual synchrony properties of the GCS. It remains to be seen whether or not the protocol can be rewritten to use a weaker, yet more scalable type of infrastructure. For example, recent work on probabilistic failure tracking [10] has yielded a scalable membership mechanisms but with properties weaker than virtual synchrony. Applying our algorithm to such a scalable system is an interesting open question.

References

[1] B. C. Neuman and T. Ts'o. Kerberos: An authentication service for computer networks. In *IEEE Communications*, volume 32(9), pages 33–38, September 1994.

[2] K. P. Birman. A review of experiences with reliable multicast. *Software, Practice and Experience*, 29(9):741–774, Sept 1999.

[3] E. Chiakpo. *RS/6000 SP High Availability Infrastructure*. IBM, November 1996.

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

[5] D. Balenson, D. McGrew, and A. Sherman. Key management for large dynamic groups: One-way function trees and amortized initialization. Technical report, Febuary 1999. draft-balenson-groupkeymgmt-oft-00.txt.

[6] 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.

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

[8] G. Goft and E. Y. Lotem. The as/400 cluster engine: A case study. In *International Workshop on Group Communication (IWGC'99)*, September 1999.

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

[10] I. Gupta, R. V. Renesse, and K. P. Birman. A pobabilistically correct leader election protocol for large groups. Tr, Department of Computer Science, University of Cornell, 2000.

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

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

[13] I. Keidar. A highly available paradigm for consistent object replication. Master's thesis, Hebrew University Jerusalem, 1995.

[14] 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.

[15] M. A. Hiltunen and R. D. Schlichting. Adaptive distributed and fault-tolerant systems. *International Journal of Computer Systems Science and Engineering*, 11(5):125–133, September 1996.

[16] M. A. Hiltunen, S. Jaiprakash, R. D.Schlichting, and C. A. Ugarte. Fine-grain configurability for secure communication. Technical Report TR00-05, Department of Computer Science, University of Arizona, June 2000.

[17] M. Bellare and P. Rogaway. Entity authentication and key distribution. In *Advances in Cryptology*, August 1993.

[18] M. J. Cox, R. S. Engelschall, S. Henson, B. Laurie, E. A. Young, and T. J. Hudson. Open ssl. http://www.openssl.org.

[19] 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.

[20] M. Steiner, G. Tsudik, and M. Waidner. Cliques: A new approach to group key agreement. In *IEEE International Conference on Distributed Computing Systems (ICDCS'98)*, May 1998.

[21] 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.

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

[23] 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.

[24] P. Zimmermann. Pretty good privacy. http://www.pgpi.com.

[25] 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.

[26] R. Friedman and A. Vaysburd. Fast replicated state machines over partitionable networks. In *IEEE 16th International Symposium on Reliable Distributed Systems*, October 1997.

[27] R. V. Rensesse, Y. Minsky, and M. Hayden. A gossip-style failure detection service. In *Middleware*, 1998.

[28] 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.

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

[30] 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.

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

[32] W. Vogels, D. Dumitriu, K. Birman, R. Gamache, R. Short, J. Vert, M. Massa, J. Barrera, and J. Gray. The design and architecture of the microsoft cluster service – a practical approach to high-availability and scalability. In *Symposium on Fault-Tolerant Computing*, number 28, June 1998.

[33] W.Diffie and M.Hellman. New directions in cryptography. *IEEE Transactions on information Theory*, IT-22:644–654, November 1976.

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

[35] Y. Amir, G. Ateniese, D. Hasse, Y. Kim, C. Nita-Rotaru, T. Schlossnagle, J. Schultz, Jonathan Stanton, and Gene Tsudik. Secure group communication in asynchronous networks with failures: Integration and experiments. In *International Conference on Distributed Computing Systems*, April 2000.

[36] Y. Kim, A. Perrig, and G. Tsudik. Simple and fault-tolerant key agreement for dynamic collaborative groups. In *7th ACM Conference on Computer and Communication Security*, November 2000.