Tree-based Group Key Agreement ⋆

Yongdae Kim[1], Adrian Perrig[2], and Gene Tsudik[1]

[1] Department of Information and Computer Science,
University of California at Irvine, CA, USA
{kyongdae, gts}@ics.uci.edu
[2] Computer Science Department
University of California, Berkeley
perrig@cs.berkeley.edu

Abstract. Secure and reliable group communication is an increasingly active research area prompted by the growing popularity of many types of group-oriented and collaborative applications. The central challenge is secure and efficient group key management. While centralized methods are often appropriate for key distribution in large multicast-style groups, many collaborative group settings require distributed key agreement techniques. This work investigates a novel group key agreement approach which blends so-called key trees with Diffie-Hellman key exchange. The resultant protocol suite is simple, secure and fault-tolerant. Moreover, its efficiency appreciably surpasses that of prior art.

1 Introduction

Fault-tolerant, scalable, and reliable communication services have become critical in modern computing. An important and popular trend is to convert traditional centralized services (e.g., file sharing, authentication, web, and mail) into distributed services spread across multiple systems and networks. Many of these newly distributed and other inherently collaborative applications (e.g., conferencing, white-boards, shared instruments, and command-and-control systems) need secure communication. However, experience shows that security mechanisms for collaborative and dynamic peer groups tend to be both expensive and unexpectedly complex. In that regard, dynamic peer groups are very different from non-collaborative, centrally managed, one-to-many (or few-to-many) broadcast groups such as those encountered in Internet multicast.

Dynamic Peer Groups (DPGs) are common in many layers of the network protocol stack and many application areas of modern computing. Examples of DPGs include replicated servers (such as database, web, time), audio and video conferencing and, more generally, applications supporting collaborative work. In contrast to large multicast groups, DPGs tend to be relatively small in size, on the order of hundred members. (Larger groups are harder to control on a peer basis and are often organized in a hierarchy.) DPGs typically assume a many-to-many (or, equivalently, any-to-any) communication pattern rather than one-to-many pattern common of larger hierarchical groups.

Despite their relatively small number, group members in a DPG may be spread throughout the Internet and must be able to deal with arbitrary partitions due to network failures, congestion, and hostile attacks. In essence, a group can be split into a number of disconnected partitions each of which must persist and function as an independent peer group.

Security requirements in collaborative DPGs present several interesting research challenges. Among these requirements, we focus on secure and efficient **group key management**. Group key management sets up a shared secret key among the group members, and serves as a cornerstone for many DPG secure services.

1.1 Group Key Management

There are several fundamentally different approaches to group key management in peer groups.

One approach relies on a single entity (called a key server) to generate keys and distribute them to the group. We refer to it as **centralized group key distribution**. Essentially, a key server maintains long-term shared keys with each group member in order to enable secure two-party communication for the actual key distribution. One form of this solution uses a fixed trusted third party (TTP) as the key server. This approach has two problems: 1) TTP must

⋆ An early version of this paper has appeared, in part, in [18].

be constantly available, and 2) TTP must exist in every possible subset of a group in order to support continued operation in the event of network partitions. The first problem can be addressed with fault-tolerance and replication techniques. The second, however, is impossible to solve in a scalable and efficient manner. We note, however, that the centralized approach works well in one-to-many multicast scenarios since a TTP (or a set thereof) placed at, or very near, the source of communication can support continued operation within an arbitrary partition as long as it includes the source. Typically, one-to-many settings only aim to offer continued operation within a single partition that includes the source. Whereas, many-to-many environments must offer continued operation in an arbitrary number of partitions.

Another approach – called **decentralized group key distribution** – involves dynamically selecting a group member to generate and distribute keys to other group members. This approach is more robust and, thus, more applicable to many-to-many groups since any partition can continue operation by electing a key server. The drawback is that, as in the TTP case, a key server must establish long-term pairwise secure channels with all current group members in order to distribute group keys. Consequently, each time a new key server comes into play, significant costs must be incurred to set up these channels. Another disadvantage, again as in the TTP case, is the reliance on a single entity to generate good (i.e., cryptographically strong, random) keys.

In contrast to the above approaches, contributory group key management requires each group member to contribute an equal share to the common group key (which is then computed as a function of all members' contributions). This avoids the problems with the centralized trust and the single point of failure. Moreover, some contributory methods do not require the establishment of pairwise secret channels among group members. One significant problem with, current contributory **group key agreement** [1] protocols is that they are not designed to tolerate failures and group membership changes during execution. In particular, nested (cascaded) failures, partitions and other group events are not accommodated. This is not surprising since most multi-round cryptographic protocols do not offer built-in robustness with the notable exception of protocols for fair exchange [6].

1.2 Overview

In this paper, we focus on contributory group key agreement. In doing so, we unify two important trends in group key management: 1) *key trees* to efficiently compute and update group keys and 2) Diffie-Hellman key exchange to achieve provably secure and fully distributed protocols. Our main result is a secure, surprisingly simple and efficient key management solution, called the TGDH (Tree-based Group Diffie-Hellman) protocol suite.

The TGDH protocol suite has two interesting features. First, the protocol is inherently robust by virtue of being able to cope with cascaded (nested) key management operations which can stem from tightly spaced group membership changes. Second, the protocol provides self-clustering.

Our key tree is logical, meaning that physical location of the membership does not depend on the tree structure. After a network partition, the members scattered in the key tree will form a smaller subtree, i.e. cluster. TGDH protocol is relatively more expensive to process this partition. However, following the heal of the partition, future partition on the same link will result in faster process. We believe these to be issues of independent interest.

Organization: The rest of this paper is organized as follows. Section 2 presents our assumptions and requirements for the reliable group communication system. Section 3 introduces cryptographic requirements of our group key agreement protocol and Section 4 introduces notation and terminology. The actual protocols are described in Section 5 followed by practical aspects of the protocol in Section 6. Section 7 analyzes both conceptual and experimental protocol complexity. The summary of related work appears in Section 8. Finally, security argument of the proposed protocols are provided in Appendix A.

2 Group Communication and Group Key Agreement

As noted in the introduction, many modern collaborative and distributed applications require a reliable group communication platform. The latter, in turn, needs specialized security mechanisms to perform – among other things – group key management. This dependency (or need) is mutual since practical group key agreement protocols themselves rely on the underlying group communication semantics for protocol message transport and strong membership semantics. Implementing multi-party and multi-round cryptographic protocols without such support is foolhardy as, in the end, one winds up reinventing reliable group communication tools.

[1] We use the term "agreement," as opposed to "distribution", to emphasize the contributory nature of the key management.

In this section we begin with a brief discussion of reliable group communication. Next, we summarize the relationship between group membership events and group key management protocols and conclude with the summary of desired cryptographic properties.

2.1 Group Communication Semantics and Support

There are two commonly used strong group communication semantics: Extended Virtual Synchrony (EVS) [22, 2] and View Synchrony (VS) [16]. Both guarantee that: 1) group members see the same set of messages between two sequential group membership events, and, 2) the sender's requested message order (e.g., FIFO, Causal, or Total) is preserved. VS provides a stricter service whereas EVS implementations are generally more efficient.

The main difference between EVS and VS is that EVS guarantees that messages are delivered to all receivers in the same membership as existed when the message was originally sent on the network. VS, in contrast, offers a stricter guarantee that messages are delivered to all recipients in the same membership as viewed by the sender application when it originally sent the message.

Providing the latter property requires an extra round of acknowledgment messages from all members before installing a new membership. This need for acknowledgments dictates that the groups be closed, only allowing members of the group to send messages to it. However, the knowledge that a message is received in the membership the sender believed it was sent in makes implementing secure group communication easier because every message is encrypted with the same key as the receiver believes is current when the message is delivered to them.

An implementation of any distributed fault-tolerant group key agreement protocol requires VS. This is because implementing group key agreement on top of EVS would require the key agreement protocol to incorporate and implement semantics identical to those of VS in order to correctly keep state of which messages were sent in which key *epoch*. (Intuitively, this is because membership events are unpredictable and each triggers an instance of a key agreement protocol. Thus, multiple key agreement protocols can overlap in time and cause instability unless significant amount of state is kept within the key agreement protocol implementation.) For this reason, there is no particular benefit to building key agreement on top of EVS semantics.

The issues surrounding implementation of key agreement in dynamic peer groups are addressed in detail in [3]. Suffice it to say that, in the context of this paper we require for the underlying group communication to provide View Synchrony (VS). However, we stress that VS is needed for the sake of fault-tolerance and robustness; the security of our protocols is in no way affected by the lack of VS.

2.2 Group Membership Events

A comprehensive group key agreement solution must handle adjustments to group secrets subsequent to all membership change operations in the underlying group communication system.

We distinguish among single and multiple member operations. Single member changes include member addition or deletion. The former occurs when a prospective member wants to join a group and the latter occurs when a member wants to leave (or is forced to leave) a group. While there might be different reasons for member deletion – such as voluntary leave, involuntary disconnect or forced expulsion – we believe that group key agreement must only provide the tools to adjust the group secrets and leave the rest up to the higher-layer (application-dependent) security mechanisms.

Multiple member changes also include addition and deletion. We refer to the multiple addition operation as *group merge*, in which case two or more groups merge to form a single group. We refer to the multiple leave operation as *group partition*, whereby a group is split into smaller groups. A group partition can take place for several reasons two of which are fairly common:

1. Network failure – this occurs when a network event causes disconnectivity within the group. Consequently, a group is split into fragments some of which are singletons while others (those that maintain mutual connectivity) are sub-groups.
2. Explicit (application-driven) partition – this occurs when the application decides to split the group into multiple components or simply exclude multiple members at once.

Similarly, a group merge can be either voluntary or involuntary:

1. Network fault heal – this occurs when a network event causes previously disconnected network partitions to reconnect. Consequently, groups on all sides (and there might be more than two sides) of an erstwhile partition are merged into a single group.
2. Explicit (application-driven) merge – this occurs when the application decides to merge multiple pre-existing groups into a single group. (The case of simultaneous multiple-member addition is not covered.)

At the first glance, events such as network partitions and fault heals might appear infrequent and dealing with them might seem to be a purely academic exercise. In practice, however, such events are common owing to network misconfigurations and router failures. In addition, in the environment of *ad hoc* wireless communication, network partitions are both common and expected. In [22], Moser et al. offer some compelling arguments in support of these claims. Hence, dealing with group partitions and merges is a crucial component of group key agreement.

In addition to the aforementioned membership operations, periodic refreshes of group secrets are advisable so as to limit the amount of ciphertext generated with the same key and to recover from potential compromises of members' contributions or prior session keys. This is discussed in the next section.

3 Cryptographic Properties

One of the most important security requirements of a group key agreement is called **key freshness**. A key is fresh if it can be guaranteed to be new, as opposed to possibly an old key being reused through actions of either an adversary or authorized party.

Furthermore, the session key should be known only to the involved parties. Based on the related parties, we can define four important security properties encountered in group key agreement (and actually these are the major security requirements for this thesis).

Definition 1. Assume that a group key is changed m times and the sequence of successive group keys is $\mathcal{K} = \{K_0, \ldots, K_m\}$.

1. **Group Key Secrecy** guarantees that it is computationally infeasible for a passive adversary to discover any group key $K_i \in \mathcal{K}$ for all i.
2. **Forward Secrecy** guarantees that a passive adversary who knows a contiguous subset of old group keys (say $\{K_i, K_{i+1}, \ldots, K_j\}$) cannot discover any subsequent group key K_l for all i, j, l where $0 \leq i < j < l$.
3. **Backward Secrecy** guarantees that a passive adversary who knows a contiguous subset group keys (say $\{K_i, K_{i+1}, \ldots, K_j\}$) cannot discover preceding group key K_l for all l, j, k where $l < i < j$.
4. **Key Independence** guarantees that a passive adversary who knows a proper subset of group keys $\hat{K} \subset \mathcal{K}$ cannot discover any other group key $\bar{K} \in (\mathcal{K} - \hat{K})$.

The relationship among the properties is intuitive. Either of Backward or Forward Secrecy subsumes Group Key Secrecy and Key Independence subsumes the rest. Also, the combination of Backward and Forward Secrecy forms Key Independence. While the requirement of key independence is quite intuitive, it may be undesirable in a certain environment in practice. For example, a group conference may allow some belated member to know previous group keys in order to catch up old conference. In any case, this decision should be determined by local group policy.

Our definition of group key secrecy allows leakage of partial information. For example, if we splits the group key into secret key to a symmetric cipher and secret key to a message authentication code naively, an attacker may get one of the secret key according to this definition. Therefore, it would be more desirable to to guarantee that any bit of the group key is unpredictable. Due to this reason, we prove a decisional version of group key secrecy in Section A. In other words, decisional version of group key secrecy guarantees that it is computationally infeasible for a passive adversary to **distinguish** any group key K_i from random number.

Our definitions of Backward and Forward Secrecy are stronger than those typically found in the literature. The two are often defined (respectively) as [29, 24]:

– Previously used group keys must not be discovered by new group members.
– New keys must remain out of reach of former group members.

The difference is that the adversary here is assumed to be a current or a former group member. Our definition additionally includes the cases of inadvertently leaked or otherwise compromised group keys. We refer to the above as Weak Forward Secrecy and Weak Backward Secrecy, respectively.

In this paper we do not consider (implicit or explicit) key authentication as part of the group key management protocols. All communication channels are public but authentic. This means, as discussed later in the paper, that all messages are digitally signed by the sender using a sufficiently strong public key signature method, such as DSA or RSA. Furthermore, each message includes: the protocol identifier (TGDH), the event type identifier (i.e., JOIN, LEAVE, etc.), the protocol sequence number and the sender's timestamp. All receivers are required to verify signatures on all received messages and check the aforementioned fields. Since no long-term secrets or other keys are used for encryption, we are not concerned with Perfect Forward Secrecy (PFS) as it is achieved trivially.

4 Notation and Definitions

We use the following notation:

N	number of protocol parties (group members)
$\mathcal{C}$	set of current group members
$\mathcal{L}$	set of leaving members
$\mathcal{J}$	set of newly joining members
M_i	i-th group member; $i \in \{1, \ldots, N\}$
h	height of a tree
$\langle l, v \rangle$	v-th node at level l in a tree
T_i	M_i's view of the key tree
$\widehat{T}_i$	M_i's modified tree after membership operation
$T_{\langle i,j \rangle}$	A subtree rooted at node $\langle i, j \rangle$
BK_i^*	set of M_i's blinded keys
p, q	prime integers
α	exponentiation base

Key trees have been suggested in the past for centralized group key distribution systems; the work of Wallner et al. [31] is the earliest such proposal. One of the key features of our work is the adaptation of key trees for use in fully distributed, contributory key agreement. Figure 1 shows an example of a key tree. The root is located at level 0 and the lowest leaves are at level h. Since we use binary trees,[2] every node is either a leaf or a parent of two nodes. The nodes are denoted $\langle l, v \rangle$, where $0 \leq v \leq 2^l - 1$ since each level l hosts at most 2^l nodes.[3] Each node $\langle l, v \rangle$ is associated with the key $K_{\langle l,v \rangle}$ and the blinded key (bkey) $BK_{\langle l,v \rangle} = f(K_{\langle l,v \rangle})$ where the function $f()$ is modular exponentiation in prime order groups, i.e., $f(k) = \alpha^k \bmod p$ (analogous to the Diffie-Hellman protocol). Assuming a leaf node $\langle l, v \rangle$ hosts the member M_i, the node $\langle l, v \rangle$ has M_i's session random key $K_{\langle l,v \rangle}$. Furthermore, the member M_i at node $\langle l, v \rangle$ knows every key along the path from $\langle l, v \rangle$ to $\langle 0, 0 \rangle$, referred to as the *key-path* and denoted KEY_i^*. In Figure 1, if a member M_2 owns the tree T_2, then M_2 knows every key $\{K_{\langle 3,1 \rangle}, K_{\langle 2,0 \rangle}, K_{\langle 1,0 \rangle}, K_{\langle 0,0 \rangle}\}$ in $KEY_2^* = \{\langle 3, 1 \rangle, \langle 2, 0 \rangle, \langle 1, 0 \rangle, \langle 0, 0 \rangle\}$ and every bkey $BK_2^* = \{BK_{\langle 0,0 \rangle}, BK_{\langle 1,0 \rangle}, \ldots, BK_{\langle 3,7 \rangle}\}$ on T_2. Every key $K_{\langle l,v \rangle}$ is computed recursively as follows:

$$\begin{aligned} K_{\langle l,v \rangle} &= (BK_{\langle l+1,2v+1 \rangle})^{K_{\langle l+1,2v \rangle}} \bmod p \\ &= (BK_{\langle l+1,2v \rangle})^{K_{\langle l+1,2v+1 \rangle}} \bmod p \\ &= \alpha^{K_{\langle l+1,2v \rangle} K_{\langle l+1,2v+1 \rangle}} \bmod p \\ &= f(K_{\langle l+1,2v \rangle} K_{\langle l+1,2v+1 \rangle}) \end{aligned}$$

In other words, computing a key at $\langle l, v \rangle$ requires the knowledge of the key of one of the two child nodes and the bkey of the other child node. $K_{\langle 0,0 \rangle}$ at the root node is the group secret shared by all members. We note that this value

[2] Note that the tree needs to be binary, since our protocol uses the two-party Diffie-Hellman key exchange to derive a node key from the contribution of the two children.

[3] Even though the key tree is not balanced, we assume a perfectly balanced tree for node numbering. Thus, a node's $\langle l, v \rangle$ left and right children have indexes $\langle l + 1, 2v \rangle$ and $\langle l + 1, 2v + 1 \rangle$, respectively.

is never used as a cryptographic key for the purposes of encryption, authentication or integrity. Instead, such special-purpose sub-keys are derived from the group secret, e.g., by setting $K_{\text{group}} = h(K_{\langle 0,0\rangle})$ where h is a cryptographically strong hash function.

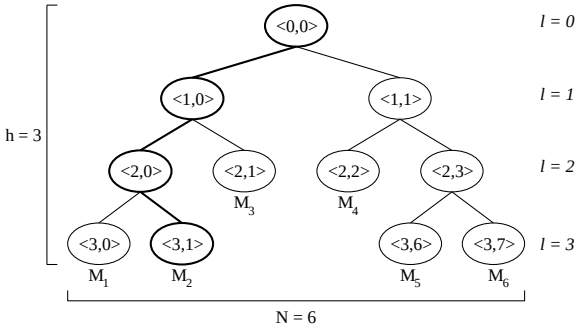

Fig. 1. Notation for tree

For example, in Figure 1, M_2 can compute $K_{\langle 2,0\rangle}$, $K_{\langle 1,0\rangle}$ and $K_{\langle 0,0\rangle}$ using $BK_{\langle 3,0\rangle}$, $BK_{\langle 2,1\rangle}$, $BK_{\langle 1,1\rangle}$, and $K_{\langle 3,1\rangle}$. The final group key $K_{\langle 0,0\rangle}$ is:

$$K_{\langle 0,0\rangle} = \alpha^{\left(\alpha^{r_3(\alpha^{r_1 r_2})}\right)\left(\alpha^{r_4(\alpha^{r_5 r_6})}\right)}.$$

To simplify our subsequent protocol description, we introduce the term *co-path*, denoted as CO_i^*, which is the set of siblings of each node in the key-path of member M_i. For example, the co-path CO_2^* of member M_2 in Figure 1 is the set of nodes $\{\langle 3,0\rangle, \langle 2,1\rangle, \langle 1,1\rangle\}$. Consequently, every member M_i at leaf node $\langle l, v\rangle$ can derive the group secret $K_{\langle 0,0\rangle}$ from all bkeys on the co-path CO_i^* and its session random $K_{\langle l,v\rangle}$.

5 TGDH Protocols

In this section we introduce the four basic protocols that form the TGDH protocol suite: join, leave, merge, and partition. All protocols share a common framework with the following notable features:

- Each group member contributes an equal share to the group key. The key is computed as a function of all current group members' shares.
- Each share is secret (private to each group member) and is never revealed.
- As the group grows, new members' shares are factored into the group key and, upon each new member's joining, one of the old members changes its share.
- As the group shrinks, departing members' shares are removed from the new group key, and at least one remaining member changes its key share[4].
- All protocol messages are signed by the sender. (We use RSA for this purpose, since the number of receivers are greater than the number of senders in our protocol.)

After every membership change, all remaining members independently update the key tree structure. Since we assume that the underlying communication system provides *view synchrony* (see Section 2), all members who correctly execute the protocol, recompute identical key trees after any membership event. The following is the minimal requirement for computing the group key:

Proposition 2. *A group key can be computed from any member's secret share (i.e., any leaf value) and all bkeys on the co-path to the root.*[5]

[4] This prevents the group from reusing old keys. For example, if a member joins and immediately leaves, the group key would be the same before the join and after the leave. Although, in practice, this is not always a problem and might even be a desirable feature, we choose to err on the side of caution and change the key. In more concrete terms, changing the key upon all membership changes preserves key independence [29, 7].

[5] We introduced the notion of co-path in Section 4.

It is easy to see that knowledge of its own secret share and all sibling bkeys on the path to the root enables a member to compute all intermediate keys on its key-path, including the root group key. This is similar to other tree-based schemes [33, 31] where each member is required to know all keys on the path from itself (leaf) to the root. Although not strictly necessary for computing group key, our protocol also requires each member to know **all** bkeys in the entire key tree. As will be seen below, this makes the handling of future membership changes more efficient and robust.

As part of the protocol, a group member can take on a special **sponsor** role which involves computing intermediate keys and broadcasting to the group. Each broadcasted message contains the key tree with every bkey known to the source. (We stress that intermediate keys are never broadcasted!) Any member in the group can unilaterally take on this responsibility, depending on the type of membership event. In some cases, such as a partition event, multiple sponsors might be involved.

In case of an additive change (join or merge), all group members identify a unique sponsor. This sponsor is responsible for updating its secret key share, computing affected $[key, bkey]$ pairs and broadcasting all bkeys of the new tree to the rest of the group. The common criteria for sponsor selection is determined by the tree maintenance strategy described in Section 5.5. We emphasize, from the outset, that sponsor is not a privileged entity: its only task is the updating and broadcasting of tree information to the group.

In response to a subtractive membership change (leave or partition), all members update the tree in the same manner. Since the case of partition subsumes the case of a single leave, we discuss it in more detail. Group partition results in a smaller tree since some leaf nodes disappear. As a result, some subtrees acquire new siblings; therefore, new intermediate keys and bkeys must be computed through a Diffie-Hellman exchange between the new siblings sub-trees. The computation proceeds in a bottom-up fashion with each member computing keys and bkeys until either: 1) it blocks due to a dependency on a new sibling bkey that it does not yet know, or 2) it computes the new root (group) key. If a member blocks without computing any new keys, it does nothing. Otherwise, it broadcasts its view of the key tree which includes the newly computed bkeys. This process is repeated at most h times where h is the height of the tree, i.e., until all remaining members compute the new group key.

5.1 TGDH Membership Events

As discussed in Section 2, a group key agreement scheme needs to provide key adjustment protocols stemming from membership changes. TGDH includes protocols in support of the following operations:

- Join: a new member is added to the group
- Leave: a member is removed from the group
- Merge: a group is merged with the current group
- Partition: a subset of members are split from the group
- Key refresh: the group key is updated

Before turning our attention to the actual protocols we stress that, while a comprehensive protocol suite must address all types of key adjustment operations, the general policy (or case-by-case decisions) regarding if and when to change a group key is the responsibility of the application and/or the group communication system.

The following sections (5.2 to 5.6), present the four protocols. In each section, we assume that every member can unambiguously determine both the sponsors and the insertion location in the key tree (in case of an additive event). Later in Section 5.5, we will explain how this works. Note that the key refresh operation can be considered a special case of leave without any members actually leaving the group.

5.2 Join Protocol

We assume the group has n members: $\{M_1, \ldots, M_n\}$. The new member M_{n+1} initiates the protocol by broadcasting a join request message that contains its own bkey $BK_{\langle 0,0 \rangle}$. This message is distinct from any JOIN messages generated by the underlying group communication system, although, in practice, the two might be combined for efficiency's sake.

Each current member receives this message and first determines the insertion point in the tree. The insertion point is the shallowest rightmost node, where the join does not increase the height of the key tree. Otherwise, if the key tree is fully balanced, the new member joins to the root node. The sponsor is the rightmost leaf in the subtree rooted at

Step 1: The new member broadcasts request for join

M_{n+1} $\xrightarrow{BK_{\langle 0,0\rangle}=\alpha^{r_{n+1}}}$ $\mathcal{C} = \{M_1, \ldots, M_n\}$

Step 2: Every member

- updates key tree by adding new member node and new intermediate node,
- removes all keys and bkeys from the leaf node related to the sponsor to the root node,

The sponsor M_s additionally

- generates new share and computes all $[key, bkey]$ pairs on the key path,
- and broadcasts updated tree $\widehat{T}_s$ including only bkeys.

$\mathcal{C} \cup \{M_{n+1}\} = \{M_1, \ldots, M_{n+1}\}$ $\xleftarrow{\widehat{T}_s(BK_s^*)}$ M_s

Step 3: Every member computes the group key using $\widehat{T}_s$

Fig. 2. Join Protocol

the insertion node. Next, each member creates a new intermediate node and a new member node, and promotes the new intermediate node to be the parent of both the insertion node and the new member node. After updating the tree, all members, except the sponsor, block. The sponsor proceeds to update its share and compute the new group key; it can do this since it knows all necessary bkeys. Next, the sponsor broadcasts the new tree which contains all bkeys. All other members update their trees accordingly and compute the new group key (see Proposition 2).

It might appear wasteful to broadcast the entire tree to all members, since they already know most of the bkeys. However, since the sponsor needs to send a broadcast message to the group anyhow, it might as well include more information which is useful to the new member, thus saving one unicast message to the new member (which would have to contain the entire tree).

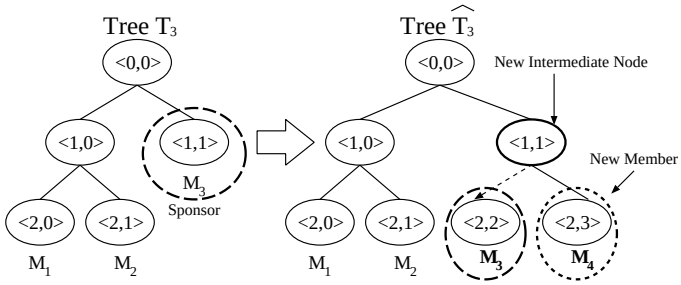

Fig. 3. Tree update: join

Figure 3 shows an example of member M_4 joining a group where the sponsor (M_3) performs the following actions:

1. renames node $\langle 1,1\rangle$ to $\langle 2,2\rangle$
2. generates a new intermediate node $\langle 1,1\rangle$ and a new member node $\langle 2,3\rangle$
3. promotes $\langle 1,1\rangle$ as the parent node of $\langle 2,2\rangle$ and $\langle 2,3\rangle$

Since all members know $BK_{\langle 2,3\rangle}$ and $BK_{\langle 1,0\rangle}$, M_3 can compute the new group key $K_{\langle 0,0\rangle}$. Every other member also performs step 1 and 2, but cannot compute the group key in the first round. Upon receiving the broadcasted bkeys, every member can compute the new group key.

5.3 Leave Protocol

Once again, we start with n members and assume that member M_d leaves the group. The sponsor in this case is the rightmost leaf node of the subtree rooted at the leaving member's sibling node. First off, as shown in Figure 4, each member updates its key tree by deleting the leaf node corresponding to M_d. The former sibling of M_d is promoted to replace M_d's parent node. The sponsor generates a new key share, computes all $[key, bkey]$ pairs on the key path up to the root, and broadcasts the new set of bkeys. This allows all members to compute the new group key.

Step 1: Every member
- updates key tree by by removing the leaving member node and relevant parent node,
- removes all keys and bkeys from the leaf node related to the sponsor to the root node,

Sponsor M_i additionally
- generates new share and computes all $[key, bkey]$ pairs on the key path,
- and broadcasts updated tree $\widehat{T}_s$ including only bkeys.

$$M_s \xrightarrow{\widehat{T}_s(BK_s^*)} \{M_1..M_n\} - \{M_d\}$$

Step 2: Every member computes the group key using $\widehat{T}_s$

Fig. 4. Leave Protocol

Looking at the setting in Figure 5, if member M_3 leaves the group, every remaining member deletes $\langle 1,1\rangle$ and $\langle 2,2\rangle$. After updating the tree, the sponsor (M_5) picks a new share $K_{\langle 2,3\rangle}$, recomputes $K_{\langle 1,1\rangle}, K_{\langle 0,0\rangle}, BK_{\langle 2,3\rangle}$ and $BK_{\langle 1,1\rangle}$, and broadcasts the updated tree $\widehat{T}_5$ with BK_5^*. Upon receiving the broadcast message, all members compute the group key. Note that M_3 cannot compute the group key, though it knows all the bkeys, because its share is no longer part of the group key.

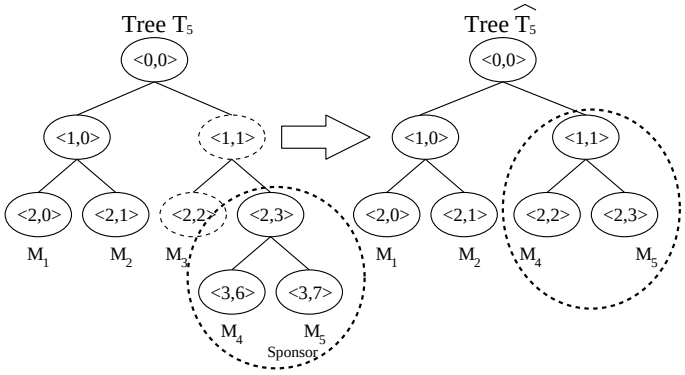

Fig. 5. Tree updating in leave operation

5.4 Partition Protocol

Assume that a network fault causes a partition of the n-member group. From the viewpoint of each remaining member, this event appears as a concurrent leave of multiple members. The partition protocol is involves multiple rounds; it runs until all members compute the new group key.

In the first round, each remaining member updates its tree by deleting all partitioned members as well as their respective parent nodes and "compacting" the tree. The procedure is as follows:

All leaving nodes are sorted by depth order. Starting at the deepest level, each pair of leaving siblings is collapsed into its parent which is then marked as leaving. This node is re-inserted into the leaving nodes list. The above is repeated until all leaving nodes are processed, i.e., there are no more leaving nodes that can be collapsed.
The resulting tree has a number of leaving (leaf) nodes but every such node has a remaining sibling node. Now, for each leaving node we identify a sponsor using the same criteria as described in Section 5.3.

Each sponsor now computes keys and bkeys on the key-path as far up the tree as possible. Then, each sponsor broadcasts the set of new bkeys. Upon receiving a broadcast, each member checks whether the message contains new bkeys. This procedure iterates until all members obtain the group key. (Recall that a member can compute the group key if it has all the bkeys on its co-path.)

Step 1: Every member
- updates key tree by by removing all the leaving member nodes and their parent node,
- removes all keys and bkeys from the leaf node related to the sponsor to the root node,

– Each sponsor M_{s_t}
- If M_{s_t} is the shallowest rightmost sponsor, generates new share,
- computes all $[key, bkey]$ pairs on the key path until it can proceed,
- and broadcasts updated tree $\widehat{T}_{s_t}$ including only bkeys.

$$M_{s_t} \xrightarrow{\widehat{T}_{s_t}(BK^*_{s_t})} \mathcal{C} - \mathcal{L}$$

Step 2 to h (Until a sponsor M_{s_j} could compute the group key)
– For each sponsor M_{s_t}
- computes all $[key, bkey]$ pairs on the key path until it can proceed,
- and broadcasts updated tree $\widehat{T}_{s_t}$ including only bkeys.

$$M_{s_t} \xrightarrow{\widehat{T}_{s_t}(BK^*_{s_t})} \mathcal{C} - \mathcal{L}$$

Step $h+1$: Every member computes the group key using $\widehat{T}_{s_t}$

Fig. 6. Partition Protocol

To provide key independence, one of the remaining members needs to change its key share. For this reason, in the first round of the partition protocol, we require the shallowest rightmost sponsor to generate a new key share.

This protocol takes multiple rounds to complete. We analyze the number of rounds after p members are partitioned from a group of n members. In the first round, each remaining member updates its tree by deleting all partitioned members as well as their respective parent nodes. Now, each key tree has at most p paths with empty bkeys. The expected number of paths with empty keys is $p/2$. Filling up these bkeys requires at most $min(\log_2 p, h)$ rounds, since 1) every sponsor in each subsequent rounds computes bkeys as far up the tree as possible, and 2) the number of rounds never exceeds the tree height.

Figure 7 shows an example where all remaining members delete all nodes of leaving members and compute keys and bkeys in the first round. In the figure on the right, any of M_2 or M_3 (M_5 or M_6) cannot compute the new group key, since they lack the bkey $BK_{\langle 1,1\rangle}$ ($BK_{\langle 1,0\rangle}$), respectively. However, M_3 broadcasts $BK_{\langle 1,0\rangle}$ in the first round, and M_6 can thus compute the group key. Finally, every member knows all bkeys and can compute the group key. As discussed above, before computing $K_{\langle 1,1\rangle}$, M_6 changes its share $K_{\langle 2,3\rangle}$.

Note that, if some member M_i is able to compute the new group key in round h', then all other members can compute the group key, at the latest, in round $h'+1$, since M_i's broadcast message contains every bkey in the key tree. Hence, each member can detect the completion of the partition protocol independently.

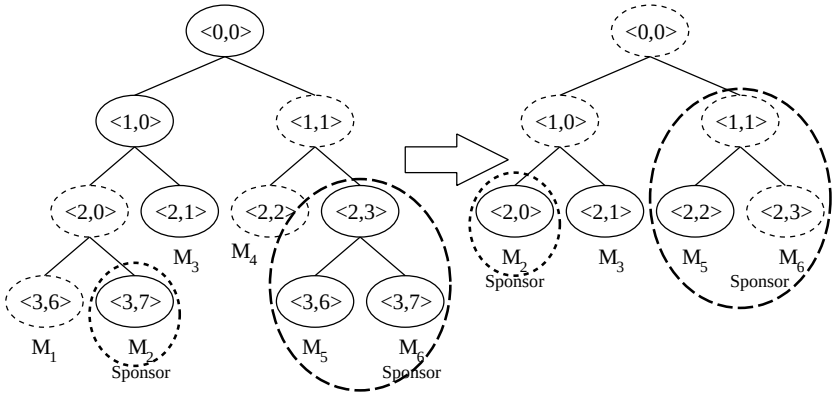

Fig. 7. Tree updating in partition operation

As discussed in Section 2, network faults can partition a group into several subgroups. After the network faults heal, subgroups may need to be merged back into a single group. We now describe the merge protocol for k merging groups.

In the first round of the merge protocol, each sponsor (the rightmost member of each group) broadcasts its tree with all bkeys to all other groups after updating the secret share of the sponsor and relevant $[key, bkey]$ pairs up to the root node. Upon receiving these messages, all members can uniquely and independently determine how to merge those k trees by tree management policy described in 5.5.

Next, each sponsor computes all $[key, bkey]$ pairs on the key-path until it either reaches the root or encounters a dependency.[6] It then broadcasts its view of the tree to the group. All members update their tree views with the new information. If the broadcasting sponsor computed the root key, then, upon receiving the broadcast, all other members can compute the root key as well. In a more general case, a broadcast *unblocks* exactly one locked sponsor who can now compute further $[key, bkey]$ pairs. This process is incremental, similar to the partition protocol. Finally, some sponsor will compute the new root key and will broadcast the key tree. Now, all members can compute the group key.

The communication overhead of the merge protocol may appear expensive at first. However, this is not the case. Let us assume k merging groups. In the first round, a sponsor in each group broadcasts its key tree after updating its session random. Upon receiving these broadcast messages, every member rebuilds a key tree which has some missing bkeys. At most k paths will have missing bkeys. Filling up these bkeys requires at most $\log_2 k$ rounds, since each sponsor in each subsequent round computes bkeys as far as it can. Therefore, a merge of k groups takes at most $\log_2 k + 1$ rounds.

Figure 9 shows an example of merging two groups, where the sponsors M_2 and M_7 broadcast their trees (T_2 and T_7) containing all the bkeys, along with BK_2^* and BK_7^*. Upon receiving these broadcast messages, every member checks whether it is the sponsor in the second round. Every member in both groups merges two trees, and then M_2, the sponsor in this example updates the key tree and computes and broadcasts bkeys.

5.5 Tree Management

Modular exponentiation is the computationally most expensive operation in TGDH. The number of exponentiations for a membership event depends on the tree structure. For example, if a single member or a whole tree merges to the root node of the current tree, at most 5 modular exponentiations are required to complete a merge operation. If a key tree is balanced, and a member joins to a leaf node, then the number of exponentiations is $4\lceil \log_2 n \rceil$ where n is the current number of users. Hence, it is easy to see that joining to the root always requires the minimum number of exponentiations for additive membership operations. If n members join to the root, however, the resulting tree becomes unbalanced (similar to a linked list). If a member in the deepest node leaves the group, $n - 1$ exponentiations are required to update the group key. However, if a key tree is fully balanced, the number of exponentiations is $4\lceil \log_2 n \rceil$. These examples indicate that a well-balanced key tree reduces the expected cost of a leave.

[6] If a sponsor cannot compute a new intermediate key, it does not broadcast but simply blocks.

5.6 Merge Protocol

Step 1: All sponsors M_{s_i} in each T_{s_i}

- generate new share and computes all $[key, bkey]$ pairs on the key path of T_{s_i},
- and broadcast updated tree $\widehat{T}_{s_i}$ including only bkeys.

$$M_{s_i} \xrightarrow{\widehat{T}_{s_i}(BK^*_{s_i})} \cup_{i=1}^{k} \mathcal{C}_i$$

Step 2: Every member

- updates key tree by adding new trees and new intermediate nodes,
- removes all keys and bkeys from the leaf node related to the sponsor to the root node,

Each sponsor M_{s_t} additionally

- computes all $[key, bkey]$ pairs on the key path until it can proceed,
- and broadcasts updated tree $\widehat{T}_s$ including only bkeys.

$$M_{s_t} \xrightarrow{\widehat{T}_{s_t}(BK^*_{s_t})} \cup_{i=1}^{k} \mathcal{C}_i$$

Step 3 to h (Until a sponsor M_{s_j} could compute the group key): Each sponsor M_{s_t}

- computes all $[key, bkey]$ pairs on the key path until it can proceed,
- and broadcasts updated tree $\widehat{T}_{s_t}$ including only bkeys.

$$M_{s_t} \xrightarrow{\widehat{T}_{s_t}(BK^*_{s_t})} \cup_{i=1}^{k} \mathcal{C}_i$$

Step $h+1$: Every member computes the group key using $\widehat{T}_{s_t}$

Fig. 8. Merge Protocol

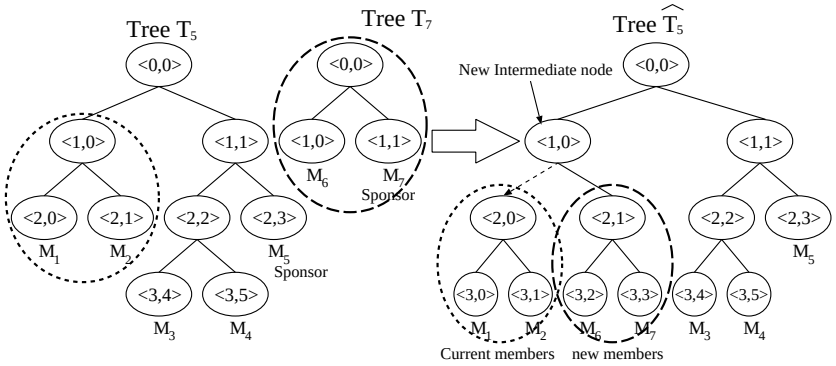

Fig. 9. Tree updating in merge operation

Remark 3. Our goal for the tree management policy is to:

- keep the key tree as balanced as possible, and
- minimize the number of modular exponentiations, and
- minimize the number of protocol rounds

5.6.1 Policy for Additive and Subtractive Events Our heuristic for keeping the key tree balanced is to choose the insertion node of a join or merge operation as the rightmost shallowest node, which does not increase the height. If we

have to increase the height of the key tree, we simply join to the root. (See also Sections 5.2 and 5.6.) We do not employ any tree balancing scheme for the subtractive events or attempt to re-balance when key tree becomes unbalanced.

In the rest of this section, we discuss our tree management policy for Merge (Join can be seen as a special case thereof). In particular, we focus on how each member independently, simultaneously, and unambiguously merges k trees and selects an insertion point for each merge. Clearly, these properties: independency, concurrency, and consistency, are crucial to obtain a correct and efficient protocol.

Recall that we have k merging trees as described in 5.6. To merge them, each member invokes the `merge_trees` function $k-1$ times:

1. First, the trees are ordered from the highest T_1 to the lowest T_k. If multiple trees are of the same height, we list them in lexicographic order of the first member in each tree.
2. Let $\widehat{T} = T_1$.
3. For $i = 2$ to k, $\widehat{T} \longleftarrow$ `merge_trees`$(\widehat{T}, T_i)$.

Since every member can order the merging trees independently and unambiguously, all members can agree on a key tree if the `merge_trees` algorithm can guarantee uniqueness of the result. We now show how to merge two trees.

If two trees are of the same height, we join one tree to the root node (insertion point) of the other. To provide unambiguous ordering we lexicographically compare the identifiers of the respective sponsors. Otherwise, we join the smaller shallower tree to the deeper tree. To locate the insertion point we first try find the rightmost shallowest node (not necessarily a leaf) where the join would not increase the overall tree height. If no such node exists (i.e., the tree height would increase anyway) the insertion point is the root node.

As an illustration, consider two trees T_h and T_l, where the height of T_h is greater than that of T_l. The `merge_trees` algorithm is as follows:

```
merge_trees (T_h, T_l) {
  T = T_h
  i = 1, j = 2^i-1;

  While (1) {
    If (height (T_l) >= Max {height (T_<i, j>) | 0 <= j < 2^i} {
      // If the height of the smaller tree is greater than that of all subtrees
      result = T_h // Nowhere to join, join to root
      Break
    } EndIf

    If (T_l is joinable to node <i, j> of tree T_h){
      result = T_<i, j> // Join to node T_<i, j>
    } EndIf
    Else{
      j--
      If(j < 0){
        i++, j = 2^i-1
      } EndIf
    } EndElse
  } EndWhile

  // Merge two trees
  T_<i+1, 2j> = T_<i, j> // Old T_<i, j> becomes the left child of new T_<i,j>
  T_<i+1, 2j+1> = T_l    // T_l becomes the right child of new T_<i, j>

  Return T
}
```

The first `if` statement in the while loop breaks when there exists no join-able node in T_h; the trees will then be joined at the root node. Join-able means that we can merge two trees without increasing the height of T_h by placing

a subtree rooted at the join-able node as the left child of itself, and putting T_l as the right child. We can see that `merge_trees` algorithm tries to fulfill the requirements alluded to in Remark 3.

5.6.2 Sponsor Selection Summary Sponsor selection in TGDH is necessary in each protocol round. (Recall that TGDH is a multi-round protocol.) As mentioned earlier, uniqueness, consistency and independence of this process is crucial for protocol correctness. Sponsor selection is performed as follows.

We already alluded to the behavior of the sponsor in two situations:

1. Additive event: member associated with the rightmost leaf node of each key tree becomes the sponsor.
2. Subtractive event: member associated with the rightmost leaf node rooted at the sibling node of each leaving member. In case of partition, there may be multiple sponsors.

The above only covers the initial protocol round. In subsequent rounds, a sponsor is always the rightmost leaf rooted at the node which lacks a current bkey.

To summarize, the role of a sponsor is three-fold: 1) refresh its key share[7], 2) compute all $[key, bkey]$ pairs as far on the key path as possible, and 3) broadcast the updated key tree to all *current* group members.

6 Practical Considerations

In this section, we desribe the TGDH implementation issues and then discuss self-stabilization and self-clustering properties.

6.1 Implementation Architecture

TREE_API is a group key agreement API that implements the cryptographic primitives of TGDH. It contains only the following three function calls:

- `tree_new_user` generates a group context for a new group member (including its secret share).
- `tree_merge_req` called by every group member when a merge occurs. This function first identifies the sponsor unambiguosly as described in Section 5.5. It then removes all $[key, bkey]$ pairs on its key path. If the caller is the sponsor, it generates its new secret share and computes $[key, bkey]$ pairs on its key path, and returns an output token. This token is then broadcast to the whole group.
- `tree_cascade` core function of TREE_API. It is invoked by every member when a subtractive event happens or when all members tries to compute the group key collaboratively. In the former case, this function removes all leaving members and their parents as described in Section 5.3. If the caller is a sponsor, this function also tries to compute $[key, bkey]$ pairs on the sponsor's key-path. In the latter case, this function is called repeatedly until the group key is computed.

The underlying communication system is assumed to deal with group communication and network events such as merges, partitions, failures and other abnormalities. [8] We use OpenSSL 0.9.6 [23] as the underlying cryptographic library.

In the following Sections (6.2 and 6.3), we show that `tree_cascade` provides robustness against cascaded network events. Since TREE_API does not provide its own communication facility, the robustness of the API was tested by simulating random events on a single machine running all group members.

6.2 Protocol Unification

Although described separately in the preceding sections, the four TGDH operations (join, leave, merge and partition) actually represent different strands of a single protocol. We justify this claim with an informal argument below.

Obviously, join and leave are special cases of merge and partition, respectively. We observe that merge and partition can be collapsed into a single protocol, since, in either case, the key tree changes and remaining group members lack

[7] In a join, the new member simply generates its first share.

[8] Currently, TGDH is integrated with Spread [4] group communication system.

some number of bkeys. This prevents them from computing the new root key. In a partition, the remaining members (in any surviving group fragment) reconstruct the tree where some bkeys are missing. In case of a merge of two groups, let us suppose that a taller (deeper) tree $\mathcal{A}$ is merged with a shorter (shallower) tree $\mathcal{B}$. Similar to a partition, all members formerly in $\mathcal{A}$ construct the new tree where some bkeys – those in $\mathcal{B}$ – are missing. (This view is symmetric since the members in $\mathcal{B}$ see the same tree but with missing bkeys in the subtree $\mathcal{A}$.)

We now established that both partition and merge initially result in a new key tree with a number of missing bkeys. In the first round of merge protocol, sponsor in each group broadcasts the key tree after updating its session random. Upon receiving this broadcast message, every member rebuilds a key tree which has some missing bkeys. Filling up this bkeys takes at most $log_2 k$ rounds as discussed in Section 5.6. A partition is very similar except the first broadcast message of merge. As we discuss in Section 5.4, every member reconstructs the key tree after a partition in at most $min(\log_2 p, h)$ rounds, where h is the tree height and p is the number of leaving memebers.

```
receive msg (msg type = membership event)
construct new tree
while there are missing bkeys
  if ((I can compute any missing keys and I am the sponsor) ||
      (received message contains all bkeys))
    compute missing bkeys and keys until it can compute
    broadcast new bkeys
  endif
  receive msg (msg type = broadcast)
  update current tree
endwhile
```

Fig. 10. Unified protocol pseudocode

The apparent similarity between partition and merge allows us to collapse the protocols stemming from all membership events into a single unified protocol. Figure 10 shows the pseudocode. The incentive for doing this is threefold. First, unification allows us to simplify the implementation and minimize its size. Second, the overall security and correctness are easier to demonstrate with a single protocol. Third, we can now claim that (with a slight modification) the TGDH protocol is self-stabilizing and fault-tolerant as discussed below.

6.3 Cascaded Events

Since network disruptions are random and unpredictable, it is natural to consider the possibility of so-called *cascaded membership events*. (In fact, cascaded events and their impact on group and multi-round protocols are often considered in group communication literature, but, alas, not often enough in the security literature.) A cascaded event occurs, in a simplest form, when one membership change occurs while another is being handled. Here *event* means any of: join, leave, partition, merge or any combination thereof. For example, a partition can occur while a prior partition is being dealt with, resulting in a cascade of size two. In principle, cascaded events of arbitrary size can occur if the underlying network is highly volatile.

We claim that the TGDH partition protocol is self-stabilizing, i.e., robust against cascaded network events. This property is notable and rare as most multi-round cryptographic protocols are not geared towards handling of such events. In general, self-stabilization is a very desirable feature since lack thereof requires extensive and complicated protocol "coating" to either: 1) shield the protocol from cascaded events, or 2) harden it sufficiently to make the protocol robust with respect to cascaded events (essentially, by making it re-entrant).

The high-level pseudocode for the self-stabilizing protocol is shown in Figure 11. The changes from Figure 10 are minimal.

Instead of providing a formal proof of self-stabilization we demonstrate it with an example. Figure 12 shows an example of a cascaded partition event. The first part of the figure depicts a partition of M_1, M_4, and M_7 from the prior group of ten members $\{M_1, \ldots, M_{10}\}$. This partition normally requires two rounds to complete the key agreement. As described in Section 5.4, every member constructs the same tree after completing the initial round. The middle part shows the resulting tree. In it, all non-leaf nodes except $K_{\langle 2,3 \rangle}$ must be recomputed as follows:

```
receive msg (msg type = membership event)
construct new tree
while there are missing bkeys
  if ((I can compute any missing keys and I am the sponsor) ||
      (received message contains all bkeys))
    compute missing keys and bkeys
    broadcast new bkeys
  endif
  receive msg
  if (msg type = broadcast)
    update current tree
  else (msg type = membership event)
    construct new tree
endwhile
```

Fig. 11. Self-stabilizing protocol pseudocode

1. First, M_2 and M_3 both compute $K_{\langle 2,0\rangle}$, M_5 and M_6 compute $K_{\langle 2,1\rangle}$ while M_8, M_9 and M_{10} compute $K_{\langle 1,1\rangle}$. All bkeys are broadcasted by each sponsor M_2, M_5 and M_8.
2. Then, as all broadcasts are received, M_2, M_3, M_5 and M_6 compute $K_{\langle 1,0\rangle}$ and $K_{\langle 0,0\rangle}$. The bkeys are broadcasted by the sponsor M_6.
3. Finally, all broadcasts are received and M_8, M_9 and M_{10} compute $K_{\langle 0,0\rangle}$.

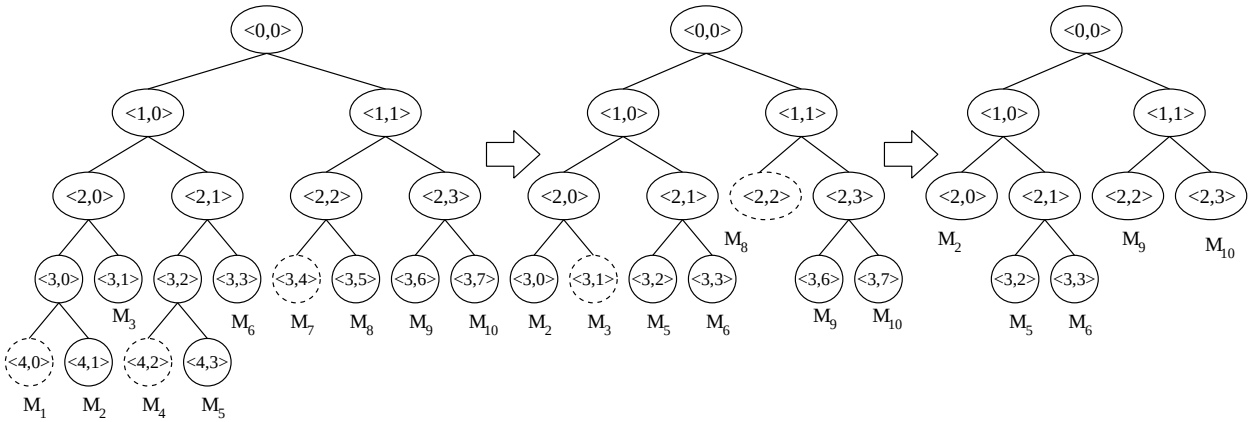

Fig. 12. An Example of Cascaded Partition

Suppose that, in the midst of handling the first partition, another partition (of M_3 and M_8) takes place. Note that, regardless of which round (1,2,3) of the first partition is in progress, the departure of M_3 and M_8 does not affect the keys (and bkeys) in the subtrees formed by M_9 and M_{10} as well as M_5 and M_6. All remaining members update the tree as shown in the rightmost part of Figure 12. The bkey of $K_{\langle 1,0\rangle}$ is the only one missing in all members' view of the tree. It is computed by M_2, M_5 and M_6 and broadcasted by M_6. When the broadcast is received, all members compute the root key.

The only remaining issue is whether a broadcast from the first partition can be received after the notification of the second (cascaded) partition. Here we rely on the underlying group communication system to guarantee that **all membership events are delivered in sequence after all outstanding messages are delivered**. In other words, if a message is sent in one membership view and membership changes while the message is not yet delivered, the mem-

bership change must be postponed until the message is delivered to the (surviving) subset of the original membership. This is essentially a restatement of View Synchrony (as discussed in Section 2).

6.4 Self-clustering

The Internet as a whole provides sporadic and unstable connectivity, e.g., web users frequently experience disconnects and server failures. The instability can occur because of congestion, equipment failures or lossy links. It can also take place as a result of denial-of-service attacks, worms and viruses. It is often the case that an unstable network component (router or link) tends to have multiple failures. In other words, an isolated, "once-in-a-blue-moon" type of failure is uncommon. Repeated failures typically complicate protocol implementation. However, oddly enough, TGDH not only survives but also benefits from repeated failures.

Similar to other tree-based key management schemes (e.g., [31, 33, 21]) the key tree in TGDH is logical: group members are leaves in a tree and internal nodes are logical. The initial placement of members (as tree leaves) is not dependent on their relative physical location. Therefore, members physically close to each other might not be neighbors in a key tree. When a partition occures, all members in the same physical group fragment form a new key tree and a new group. Then, when the partition heals, the previously separated groups are merged together in a single key tree, however, they are still clustered along the lines of the partition. If another partition happens on the same link, the members are not scattered across the key tree any longer. Therefore, this partition will take fewer rounds than the initial partition. This property is especially important in high delay wide area networks since clustering lowers the number of communication rounds as well as the number of modular exponentiations, in many cases.

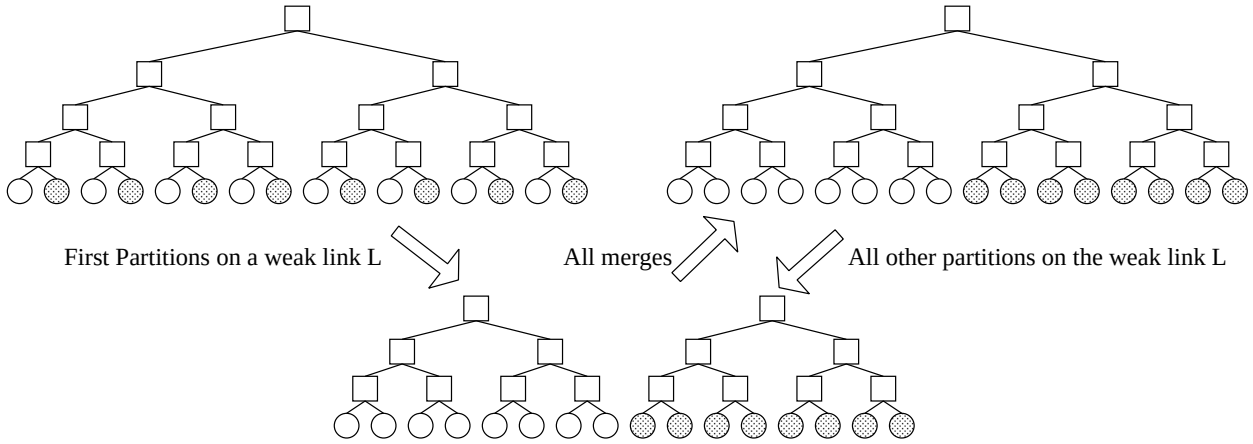

Fig. 13. An Extreme Example of Self-Clustering

Figure 13 shows an extreme example of self-clustering. Suppose that a group has sixteen members numbered M_1 through M_{16} where white odd-numbered nodes are located in one physical cluster (e.g., a LAN) and shaded even-numbered nodes in another. The two partitions are connected via an unstable link L. If L fails and a partition occurs, it takes three rounds to complete the partition protocol. It can be clearly seen that each group forms a cluster after the partition. When L comes up and the partition heals (i.e., a merge occurs), two rounds are needed to complete the merge protocol. Subsequently, all partitions on link L will require only one round and all merges – two rounds.

7 Performance Analysis

7.1 Complexity Analysis and Comparison

This section analyzes the communication and computation costs for join, leave, merge and partition protocols of TGDH. Among the various costs that we can consider, we focus on the number of rounds, the total number of messages, the serial number of exponentiations, the serial number of signature generations, and the serial number of signature

verifications. Note that we use RSA signature for message authentication, since RSA signature is more efficient in verification and some applications require that more than one processes (members) exist in one machine. The serial cost assumes parallelization within each protocol round and represents the greatest cost incurred by any participant in a given round (or protocol). The total cost cost incurred by any participant in a given round. The total cost is simply the sum of all participants' costs in a given round (or protocol).

We also compare our protocol to other contributory group key agreement schemes including GDH.3 [29], BD (Burmester-Desmedt) [12], and STR [19]. Although BD protocol was originally designed to support only group formation without authetication, we modify the BD protocol to support dynamic membership operation. This modification is minimal. As Section 8 discusses, BD protocol needs to restart whenever membership event happens due to security reasons.

Table 1 summarizes the communication and computation costs of these four protocols. The number of current group members, merging members, merging groups, and leaving members are denoted by: n, m, k[9] and p, respectively.

The height of the key tree constructed by the TGDH protocol is h. The overhead of the TGDH protocol depends on the tree height, the balancedness of the key tree, the location of the joining tree, and the leaving nodes. In our analysis, we assume the worst case configuration and list the worst-case cost for TGDH.

Table 1. Communication and Computation Costs

		Communication		Computation		
		Rounds	Messages	Exponentiations	Signatures	Verifications
GDH	Join	4	$n+3$	$n+3$	4	$n+3$
	Leave	1	1	$n-1$	1	1
	Merge	$m+3$	$n+2m+1$	$n+2m+1$	$m+3$	$n+2m+1$
	Partition	1	1	$n-p$	1	1
TGDH	Join	2	3	$\frac{3h}{2}$	2	3
	Leave	1	1	$\frac{3h}{2}$	1	1
	merge	$log_2 k+1$	$2k$	$\frac{3h}{2}$	$\log_2 k+1$	$\log_2 k$
	Partition	$min(\log_2 p, h)$	$2h$	$3h$	$min(\log_2 p, h)$	$min(\log_2 p, h)$
STR	Join	2	3	4	2	3
	Leave	1	1	$\frac{3n}{2}+2$	1	1
	Merge	2	$k+1$	$3m+1$	2	3
	Partition	1	1	$\frac{3n}{2}+2$	1	1
BD	Join	2	$2n+2$	3	2	$n+3$
	Leave	2	$2n-2$	3	2	$n+1$
	Merge	2	$2n+2m$	3	2	$n+m+2$
	Partition	2	$2n-2p$	3	2	$n-p+2$

The number of modular exponentiations for STR after a leave event depends on the location of the deepest leaving node. We thus compute the average cost, i.e. the case when the $\frac{n}{2}$ th node left the group. All other protocols, except TGDH and STR, show exact cost numbers.

In the current implementations of TGDH and STR, all group members recompute bkeys that have already been computed by the sponsors. This provides a weak form of key confirmation, since a user who receives a token from another member can check whether his bkey computation is correct. This computation, however, can be removed for better efficiency, and we consider this optimization when counting the number of exponentiations.

The BD protocol has a hidden cost which is not explained in Table 1: BD has $n-1$ modular exponentiations with a small exponent. Unfortunately, $n-1$ exponentiations can be expensive when n is large. For example when $n=50$, it requires 373 1024-bit modular multiplications, if modular exponentiation is implemented with the square-and-multiply algorithm. (OpenSSL uses Montgomery reduction and the sliding window algorithm to implement the modular exponentiation, which is faster than simple square-and-multiply algorithm. However, the former requires almost the same

[9] Clearly, $m \geq k$.

time as the latter for small exponents.) Because of this hidden cost, it is hard to compare the computational overhead of BD to the other protocols. Below, we compare the four protocols for each membership event.

Join: All the protocols except GDH.3 require two communication rounds. The most expensive communication protocol is BD which uses n messages (which are all broadcast) for each round. The other protocols use a constant number of messages, two or three. GDH is the most expensive in terms of computation, requiring a linear number of exponentiations relative to the group size. TGDH is comparatively efficient, scaling logarithmically in the number of exponentiations. STR, in turn, uses a constant number of modular exponentiations. BD requires the least modular exponentiations, but has the hidden cost explained above.

Leave: Table 1 shows that, for a leave event, the BD protocol is the most expensive from the communication point of view. The cost order between the GDH, STR and TGDH is determined strictly by the computation cost, since they all have the same communication cost (one round consisting of one message). Therefore, TGDH is best for handling leave events. STR, and GDH scale linearly with the group size. Since BD, again, has a hidden cost, it is hard to compare with other algorithms.

Merge: We first look at the communication cost. Note that GDH scales linearly with the number of the members added to the group in communication rounds, while BD and STR are more efficient using a constant number of rounds. Although a merge for TGDH takes multiple rounds, it depends on the number of groups merging that is small in most group communication system. Since BD and TGDH use $2n$ and $2k$ messages (at most) respectively, STR is the most communication-efficient for handling merge events. Examining the computation requirements, we identify BD as having the lowest cost, only three exponentiations. TGDH scales logarithmically with the group size, being more efficient than STR and GDH which scale linearly with both the group size and the number of new members added to the group.

Partition: Table 1 shows that the GDH and STR protocols are bandwidth efficient: only one round consisting of one message. BD is less efficient with two rounds of n messages each. Partition is the most expensive operation in TGDH, requiring a number of rounds bounded by at most the tree height. Computation-wise it is difficult to compare BD with the other protocols because of its hidden cost. TGDH requires a logarithmic number of exponentiations. GDH and STR scale linearly with the group size.

7.2 Experimental Results

As explained above, all the four protocols have some hidden costs. To compare the actual performance, we implement the four protocols and compare their computational cost in this section. [10] We simulate the total computation delay from the time when the membership event happens to the time when group key agreement finishes. Average delay has been measured, since each member does not finish the group key agreement simultaneously. For example, M_1 requires only 1 modular exponentiation in Figure 2 after M_3 broadcasts the key tree with bkeys. In the mean while, M_4 requires 3 modular exponentiations to complete the key agreement.

7.2.1 Test Method To perform fair comparisons, we consider the followings:

- We use $p = 1024$ and $q = 160$ for all measurements. These values are known to be secure in the current technology [20].
- We use 1024-bit RSA signature with public exponent 3 for message authentication. All protocols have multiple signature verifications that need to be processed in serial. Small public exponent is, therefore, required for fair comparison. No security risk has been known for RSA signature with small public exponent [11].
- For TGDH test, we first generate a random tree by repeating random partition/merge event enough. Since the cost of TGDH depends on the tree structure, it is fair to generate a random tree instead of well-balanced tree.

We use the following scenario to measure this delay. For join and leave, the number of current group members is n ($n = 8, 16, 24, \ldots, 128$). For partition and merge, n varies by 16, 32, 64, and 128.

Join We measure the computational delay for a member to join a group of n members. (Left graph of Figure 14) In case of TGDH, we use random tree described above. x axis denotes the number of current group members, while y axis denotes the computational delay in seconds.

[10] Total cost integrated with the actual group communication system will be reported in a separate paper.

Leave We measure the computational delay for a random member to leave a group of n members. (Right graph of Figure 14) Note that the delay for GDH and BD does not depend on the location of the member. However, the number of modular exponentiations for STR upon a leave event depends on the location of the deepest leaving node. For TGDH, we pick random member from the random tree, and measure the average delay for the leave. x axis denotes the number of resulting group members and y axis is for the computational delay in seconds.

Partition Computational delay due to a partition has been measured. If the number of current group members is n and this group shrinks to group of k, we measure the average delay for the resulting group members. For BD and GDH, the location of the leaving members does not matter. However, it is important in STR and TGDH. We, therefore, choose random members. In Figure 15, x axis denotes the number of resulting group members.

Merge Merge is the most tricky algorithm to measure fairly. First of all, in BD and GDH, only the number of resulting members decides the total delay independent of the number of groups merging. Second, the performance of the merge for STR depends on the size of the largest group (which decides the number of modular exponentiation), and the number of groups merging (which determines the number of signature verifications). Lastly, the number of groups merging (which affects the number of signature generation and verification), and the key tree structure (which is related to the number of modular exponentiations and also the signature generation/verification). Note that the number of current group members is not important for TGDH.

Since each protocol has different characteristics as above, we decide to measure the merge delay in the following way:

- We measure the delay when the number of resulting group members is $16, 32, 64$ and 128.
- We assume the maximum number of merging groups is 5. In practice, merge of two groups is most frequent. However, we allow up to five groups since some group communication systems may allow (require) more than two groups merging. Of course, it is clear that as the number of groups increases, the computational overhead of TGDH and STR increases.
- For TGDH and STR, values in x-axis mean the number of current group members. The resulting group size is 16, 32, 64, 128, respectively. The values in y-axis are the average computational delays for a member in the current group after a merge of two to five groups.

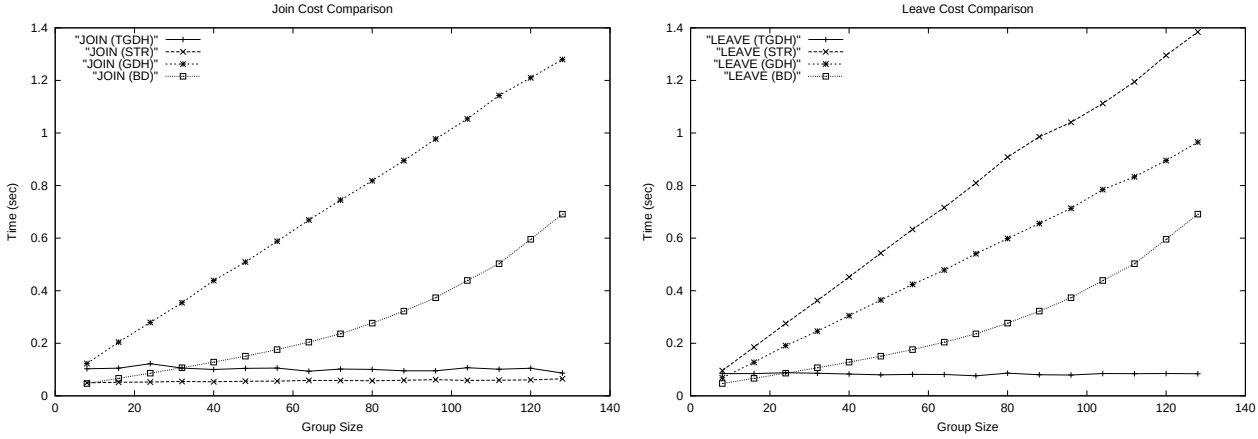

Fig. 14. Join and Leave Cost Comparison: (x, y) =(number of remaining group members after JOIN/LEAVE, computational overhead in seconds)

7.2.2 Join Result Left graph of Figure 14 depicts measurement for join results. As expected, STR requires smallest delay to join a group. A bit surprising result comes from the TGDH for a random tree. The difference between TGDH and STR is minor. In case of random tree, the joining node is located close to the root node. GDH is the worst

performer due to intensive modular exponentiations. BD also shows interesting results. Though it has constant number of modular exponentiations, the hidden cost explained above plays an important role.

7.2.3 Leave Result As expected, STR is the worst performer. Note that the worst case (when a lowest member leaves the group) cost for STR will be almost twice as much as the current average value. Performance of TGDH looks best in overall, while BD performs very well when the number of group members is less than 25. Leave cost of BD is almost the same as Join cost, since the protocol needs to restart whenever new membership event happens.

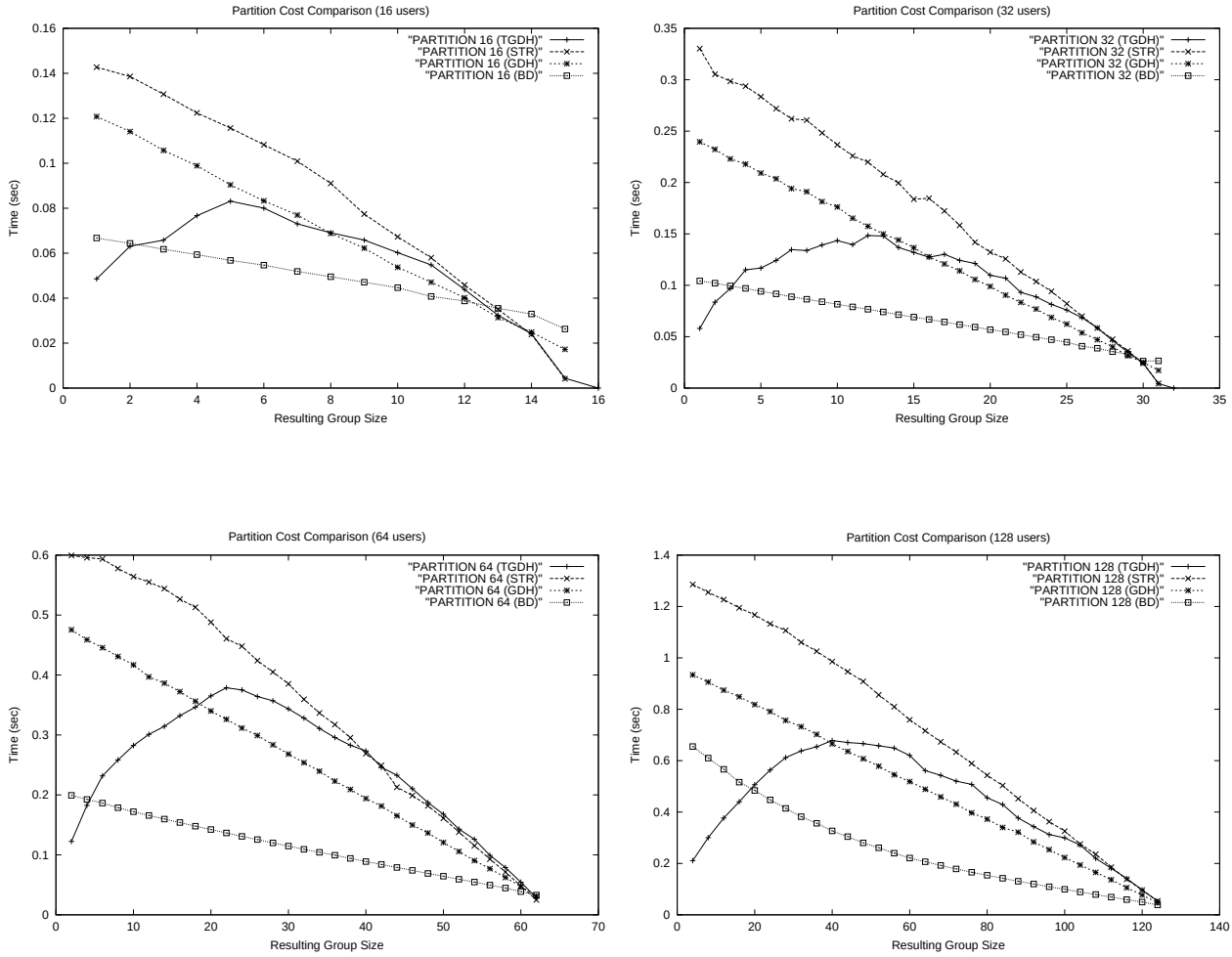

Fig. 15. Partition Cost Comparison: (x, y) =(number of remaining group members after the partition, computational overhead for an existing member if the original group shrinks to a group of x members), the original numbers of group members are 16, 32, 64, 128 respectively.

7.2.4 Partition Result Figure 15 shows partition cost when number of current group members is 16, 32, 64, and 128 respectively. As expected from the theoretical results, STR shows the worst performance due to heavy modular exponentiations. The cost of TGDH shows an interesting graph: it increases until 40% of the group members leave

the group, and decreases afterwards. This is because 1) As the number of leaving members increases, the number of modular exponentiations decreases, 2) When many members leave the group, the resulting group has many empty bkeys spread over the tree, and hence requires more messages (i.e. digital signature required). The cost of BD and GDH decreases almost linearly, because the partition cost of both only depends on the number of resulting group members.

As Section 6.4 explains, the cost of partition for TGDH can be greatly improved when the group experiences repeated network partition on the same links.

7.2.5 Merge Result Merge cost is shown in Figure 16 when the number of resulting group size is 16, 32, 64, and 128 respectively. The number of current group member is associated with x-axis. For fixed number of resulting group size, TGDH and BD show almost constant cost meaning that the merge cost does not depend on the number of current group members. In contrast, the performance of GDH strongly depends on the number of current group members, for the last member in the current group becomes the sponsor.

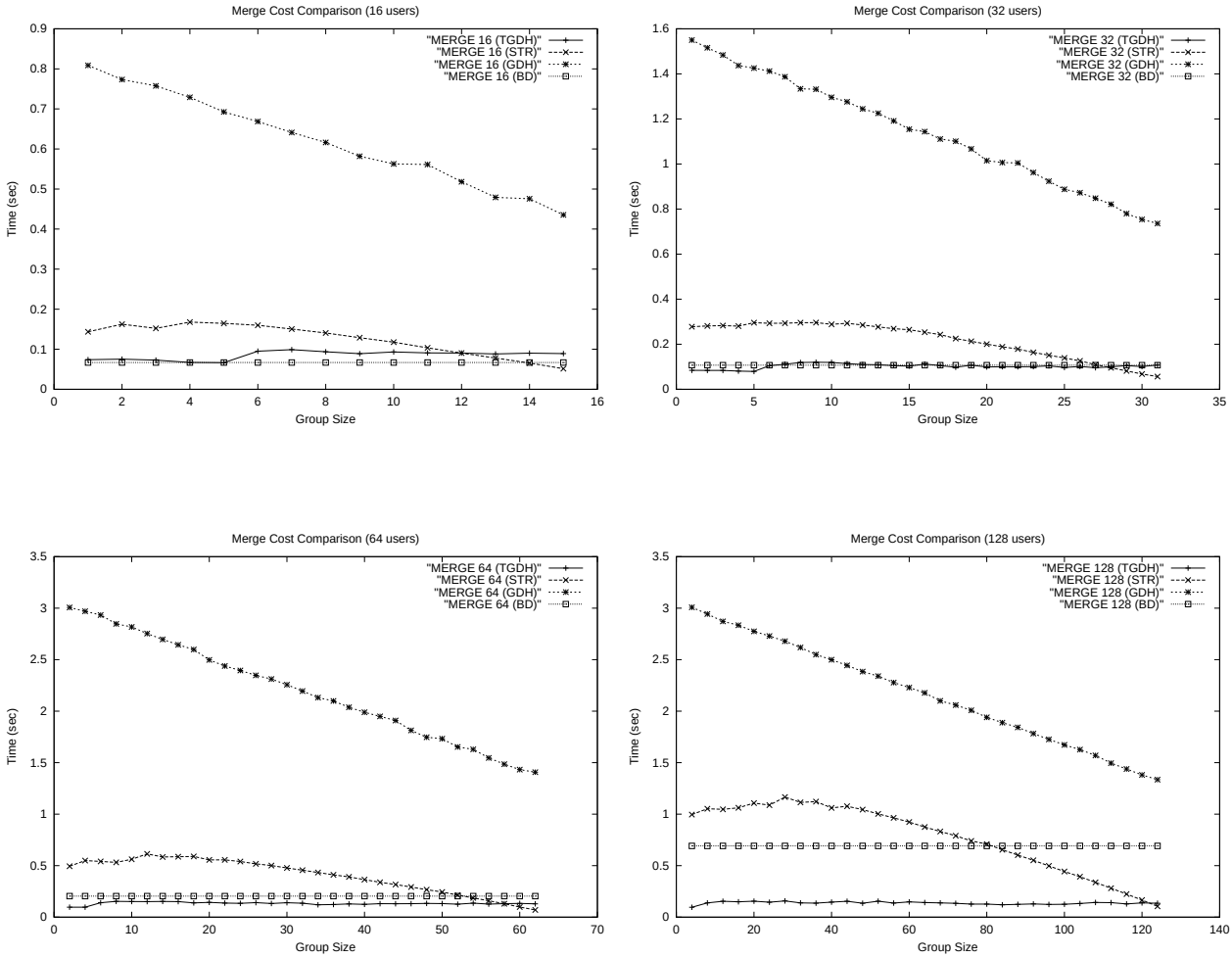

Fig. 16. Merge Cost Comparison: (x, y) =(number of current group members, computational overhead for a member located in the group of x members), after the membership event the number of group members becomes 16, 32, 64, and 128 respectively.

7.3 Discussion

Based on the experimental results on computational cost , TGDH shows the best performance despite the high cost of partition. Note that self-clustering effect of TGDH may lessen actual delay in practice.

If communication costs are considered, it seems that TGDH still outperforms other protocols (except for partition events). Having expensive computational cost for partition, TGDH also has expensive communication cost. This implies that the overall partition cost of TGDH over a high delay wide area network may be significant, although this effect can be lessened by the self-clustering effect.

Over high delay wide area network (round trip time $>$ 300 msec), it is easy to see that computation cost for a small group is not so important. For example, if the group size is 40, the maximum difference in computational delay for a join is about 300 msec. In other words, communication costs (e.g. multicast vs. unicast, the number of multicasts, the number of rounds) are much more important in the high delay network. Based on this consideration, the performance of STR gets better (relatively) as communication delay increases.

Overall, we can conclude that TGDH can perform best on low (or medium) delay networks.

8 Related Work

Group key management protocols come in three different flavors: contributory key agreement protocols, centralized, decentralized group key distribution scheme, and server-based key distribution protocols. Since the focus of this work is to provide common key to the dynamic peer group, we only consider the first two below.

8.1 Group Key Agreement Protocols

Research on group key agreement protocols started in 1982. We first summarize the early (and theoretical) group key agreement protocols, which often do not consider dynamic membership operations; they only support group genesis.

The earlist attempt for contributory group key agreement built on 2-party Diffie-Hellman (DH) is due to Ingemarsson et al. (called ING) for teleconferencing [17]. In the fist round of ING, every member M_i generates its session random N_i and computes α^{N_i}. In the subsequent rounds k to $n-1$, M_i computes $K_{i,k} = (K_{i-1 \bmod n, k-1})^{N_i}$ where K_{i-1} is the message received from M_{i-1} in the previous round $k-1$ when n is the number of group members. The resulting group key is of the form:

$$K_n = \alpha^{N_1 N_2 N_3 \ldots N_n}.$$

The ING protocol is inefficient: 1) every member has to start synchronously, 2) $n-1$ rounds are required to compute a group key, 3) it is hard to support dynamic membership operations due to its symmetricity and 4) n sequential modular exponentiations are required.

Another group key agreement developed for teleconferencing was proposed by Steer et al. [28]. This protocol is of particular interest since its group key structure is similar to that in TGDH.

$$K_n = \alpha^{N_n(\alpha^{N_{n-1}\cdots(\alpha^{N_3(\alpha^{N_2 N_1})})\cdots})}.$$

STR is well-suited for adding new group members as it takes only two rounds and four modular exponentiations. Member exclusion, however, is relatively difficult (for example, consider excluding N_1 from the group key). Due to the small number of rounds, which results in a low communication overhead, we extend the STR protocol [19].

Burmester and Desmedt construct an efficient protocol (called BD) which takes only two rounds and three modular exponentiations per member to generate a group key [12]. This efficiency allows all members to re-compute the group key for any membership change by performing this protocol. However, according to [29], most (at least half) of the members need to change their session random on every membership event. The group key in this protocol is different from STR and TGDH:

$$K_n = \alpha^{N_1 N_2 + N_2 N_3 + \ldots + N_n N_1}.$$

A shortcoming of BD is the high communication overhead. It requires $2n$ broadcast messages and each member needs to generate 2 signatures and verify $2n$ signatures. Note that BD has hidden cost mentioned in Section 7.2.

Becker and Wille analyze the minimal communication complexity of contributory group key agreement in general [8] and propose two protocols: *octopus* and *hypercube*. Their group key has the same structure as the key in TGDH. For example, for eight users their group key is:

$$K_n = \alpha^{(\alpha^{\alpha^{r_1 r_2} \alpha^{r_3 r_4}})(\alpha^{\alpha^{r_5 r_6} \alpha^{r_7 r_8}})}.$$

The Becker/Wille protocols handle join and merge operations efficiently, but the member leave operation is inefficient. Also, the *hypercube* protocol requires the group to be of size 2^n (for some integer n); otherwise, the efficiency slips.

Asokan et al. look at the problem of small-group key agreement, where the members do not have previously set up security associations [5]. Their motivating example is a meeting where the participants want to bootstrap a secure communication group. They adapt password authenticated DH key exchange to the group setting. Their setting, however, is different from ours, since they assume that all members share a secret password, whereas we assume a PKI where each member can verify any other members authenticity and authorization to join the group.

Tzeng and Tzeng propose an authenticated key agreement scheme that is based on secure multi-party computation [30]. This scheme also uses $2 \cdot N$ broadcast messages. Although the cryptographic mechanisms are quite elegant, a shortcoming is that the resulting group key does not provide perfect forward secrecy (PFS). If a long-term secret key is broken and/or published, all previous and future group keys (where that key was used) are also revealed.

Steiner et al. first address dynamic membership issues [7, 29] in group key agreement and propose a family of Group Diffie Hellman (GDH) protocols based on straight-forward extensions of the two-party Diffie-Hellman. GDH provides contributory authenticated key agreement, key independence, key integrity, resistance to known key attacks, and perfect forward secrecy. Their protocol suite is fairly efficient in leave and partition operation, but the merge protocol requires as many rounds as the number of new members to complete key agreement.

Perrig extends the work of one-way function trees (OFT, originally introduced by McGrew and Sherman [21]) to design a tree-based key agreement scheme for peer groups [24]. However, this work does not consider group merges and partitions.

8.2 Decentralized Group Key Distribution Protocols

Decentralized group key distribution protocols can be preferred to contributory group key agreement protocols, since they rely on inexpensive symmetric key encryption technique. However, all group key distribution schemes assume secure channel that is, in practice, implemented by public key cryptosystem (e.g. Diffie-Hellman). Furthermore, they require the leader to establish multiple secure two-party channels between itself and other group members in order to securely distribute the new key. Maintaining such channels in dynamic groups can be expensive since setting up each channel involves a separate two-party key agreement. When a group is dynamic, amortized number of secure channel becomes $O(n^2)$. Another disadvantage is the reliance on a single entity to generate good (i.e., cryptographically strong, random) keys.

First decentralized group key distribution scheme is due to Waldvogel et al. [13]. They propose efficient protocols for small-group key agreement and large-group key distribution. Unfortunately, their scheme for autonomous small group key agreement is not collusion resistant.

Dondeti et al. modified OFT (One-way Function Tree) [21] to provide dynamic server election [15]. This protocol has same key tree structure and uses similar notations (e.g. keys, blinded keys). Other than expensive maintainence of secure channels described above, this protocol has expensive communication cost: Even for single join and leave, this protocol can take $O(h)$ rounds to complete, when h is the height of the key tree. The authors do not consider merge and partition event, and also implementation. One advantage different from others is that their group key does not depend on a single entity.

Rodeh et al. [25] propose a decentralized group key distribution protocol extended from LKH protocol [32]. It tolerates network partitions and other network events. Even though this approach cannot help incurring basic disadvantages discussed above, authors reduce the communication and computational cost. In addition, authors use AVL tree to provide provable and efficient tree height.

9 Conclusion

This paper presented a novel decentralized group key management approach, TGDH. In doing so, we unified two important trends in group key management: 1) *key trees* to efficiently compute and update group keys and 2) group

Diffie-Hellman key exchange to achieve provably secure and fully distributed protocols. This yielded a secure, surprisingly simple and very efficient key management solution, which is supported, respectively, by the security arguments and the experiments. Moreover, our solution is inherently robust by virtue of being able to cope with cascaded (nested) key management operations which can stem from tightly spaced group membership changes. We believe this to be an issue of independent interest.

References

1. *5th ACM Conference on Computer and Communications Security*, San Francisco, California, Nov. 1998. ACM Press.
2. Y. Amir. *Replication using Group Communication over a Partitioned Network*. PhD thesis, Institute of Computer Science, The Hebrew University of Jerusalem, Jerusalem, Israel, 1995.
3. Y. Amir, G. Ateniese, D. Hasse, Y. Kim, C. Nita-Rotaru, T. Schlossnagle, J. Schultz, J. Stanton, and G. Tsudik. Secure group communication in asynchronous networks with failures: Integration and experiments. In *ICDCS 2000*, Apr. 2000.
4. Y. Amir and J. Stanton. The spread wide area group communication system. Technical Report 98-4, Johns Hopkins University Department of Computer Science, 1998.
5. N. Asokan and P. Ginzboorg. Key-agreement in ad-hoc networks. In *Nordsec'99*, 1999.
6. N. Asokan, V. Shoup, and M. Waidner. Optimistic fair exchange of digital signatures. *IEEE Journal on Selected Area in Communications*, 18(4):593–610, 2000.
7. G. Ateniese, M. Steiner, and G. Tsudik. Authenticated Group Key Agreement and Friends. In ACMCCS98 [1], pages 17–26.
8. C. Becker and U. Wille. Communication complexity of group key distribution. In ACMCCS98 [1].
9. M. Bellare and P. Rogaway. Random oracles are practical: A paradigm for designing efficient protocols. In *1st ACM Conference on Computer and Communications Security*, 1993.
10. D. Boneh. The Decision Diffie-Hellman problem. In *Third Algorithmic Number Theory Symposium*, number 1423 in Lecture Notes in Computer Science, pages 48–63. Springer-Verlag, Berlin Germany, 1998.
11. D. Boneh. Twenty years of attacks on the RSA cryptosystem. *Notices of the American Mathematical Society (AMS)*, 46(2):203–213, 1999.
12. M. Burmester and Y. Desmedt. A secure and efficient conference key distribution system. In A. D. Santis, editor, *Advances in Cryptology – EUROCRYPT '94*, number 950 in Lecture Notes in Computer Science, pages 275–286. International Association for Cryptologic Research, Springer-Verlag, Berlin Germany, 1995. Final version of proceedings.
13. G. Caronni, M. Waldvogel, D. Sun, N. Weiler, and B. Plattner. The VersaKey framework: Versatile group key management. *IEEE Journal on Selected Areas in Communications*, 17(9), Sept. 1999.
14. D. Chaum. Zero-knowledge undeniable signatures. In I. Damgard, editor, *Advances in Cryptology – EUROCRYPT '90*, number 473 in Lecture Notes in Computer Science, pages 458–464. Springer-Verlag, Berlin Germany, May 1991.
15. L. Dondeti, S. Mukherjee, and A.Samal. Disec: A distributed framework for scalable secure many-to-many communication. In *Proceedings of The Fifth IEEE Symposium on Computers and Communications (ISCC 2000)*, July 2000.
16. A. Fekete, N. Lynch, and A. Shvartsman. Specifying and using a partionable group communication service. In *ACM PODC '97*, pages 53–62, Santa Barbara, CA, August 1997.
17. I. Ingemarsson, D. T. Tang, and C. K. Wong. A conference key distribution system. *IEEE Transactions on Information Theory*, 28(5):714–720, Sept. 1982.
18. Y. Kim, A. Perrig, and G. Tsudik. Simple and fault-tolerant key agreement for dynamic collaborative groups. In S. Jajodia, editor, *7th ACM Conference on Computer and Communications Security*, pages 235–244, Athens, Greece, Nov. 2000. ACM Press.
19. Y. Kim, A. Perrig, and G. Tsudik. Communication-efficient group key agreement. In *Information Systems Security, Proceedings of the 17th International Information Security Conference IFIP SEC'01*, 2001.
20. A. K. Lenstra and E. R. Verheul. Selecting cryptographic key sizes. `http://www.cryptosavvy.com/`, Nov. 1999. Shorter version of the report appeared in the proceedings of the Public Key Cryptography Conference (PKC2000) and in the Autumn '99 PricewaterhouseCoopers CCE newsletter. To appear in Journal of Cryptology.
21. D. A. McGrew and A. T. Sherman. Key establishment in large dynamic groups using one-way function trees. Manuscript, May 1998.
22. L. Moser, Y. Amir, P. Melliar-Smith, and D. Agarwal. Extended virtual synchrony. In *ICDCS '94*, pages 56–65, June 1994.
23. OpenSSL Project team. Openssl, May 2001. http://www.openssl.org/.
24. A. Perrig. Efficient collaborative key management protocols for secure autonomous group communication. In *International Workshop on Cryptographic Techniques and E-Commerce (CrypTEC '99)*, July 1999.
25. O. Rodeh, K. Birman, and D. Dolev. Optimized rekey for group communication systems. In *NDSS2000*, pages 37–48, 2000.
26. V. Shoup. Lower bounds for discrete logarithms and related problems. In W. Fumy, editor, *Advances in Cryptology – EUROCRYPT '97*, number 1233 in Lecture Notes in Computer Science, pages 256–266. International Association for Cryptologic Research, Springer-Verlag, Berlin Germany, 1997.

27. V. Shoup. Using hash functions as a hedge against chosen ciphertext attacks. In B. Preneel, editor, *Advances in Cryptology – EUROCRYPT '2000*, number 1807 in Lecture Notes in Computer Science, pages 275–288. International Association for Cryptologic Research, Springer-Verlag, Berlin Germany, 2000.
28. D. Steer, L. Strawczynski, W. Diffie, and M. Wiener. A secure audio teleconference system. In S. Goldwasser, editor, *Advances in Cryptology – CRYPTO '88*, number 403 in Lecture Notes in Computer Science, pages 520–528, Santa Barbara, CA, USA, 1990. International Association for Cryptologic Research, Springer-Verlag, Berlin Germany.
29. M. Steiner, G. Tsudik, and M. Waidner. Key agreement in dynamic peer groups. *IEEE Transactions on Parallel and Distributed Systems*, August 2000.
30. W.-G. Tzeng and Z.-J. Tzeng. Round-efficient conference-key agreement protocols with provable security. In *Advances in Cryptology – ASIACRYPT '2000*, Lecture Notes in Computer Science, Kyoto, Japan, December 2000. International Association for Cryptologic Research, Springer-Verlag, Berlin Germany.
31. D. Wallner, E. Harder, and R. Agee. Key management for multicast: Issues and architecture. Internet-Draft draft-wallner-key-arch-00.txt, June 1997.
32. C. Wong, M. Gouda, and S. Lam. Secure group communications using key graphs. In *Proceedings of the ACM SIGCOMM '98 conference on Applications, technologies, architectures, and protocols for computer communication*, pages 68–79, 1998. Appeared in ACM SIGCOMM Computer Communication Review, Vol. 28, No. 4 (Oct. 1998).
33. C. Wong, M. Gouda, and S. Lam. Secure group communications using key graphs. *IEEE/ACM Trans. on Networking*, 8(1):16–30, 2000.

A Security Proof

This section proves the security of TGDH. We introduce decisional binary Tree Group Diffie-Hellman problem (DTGDH) in this section. On a specific group G, we prove that DTGDH problem is reducible to 2-party Decision Diffie-Hellman (DDH) problem. Later in Sections (A.3 and A.4), this result will be used to prove the security of TGDH group key agreement protocols.

A.1 2-party Decision Diffie-Hellman Problem

The proof of DTGDH problem requires a specific group G. In this section, we introduce the group G and define 2-party Decision Diffie-Hellman problem on G.

Let k be a security parameter and n be an integer. All algorithm run in probabilistic polynoimal time with k and n as inputs.

For concreteness, we consider a specific class of groups G:

On input k, algorithm *gen* chooses at random a pair (q, α) where q is a $2k$-bit value [11], and q and $p = 2q + 1$ are both prime. Before introducing G, we first consider a group $\widehat{G}$, which is a group of squares modulo prime p. This group can be explained more precisely as follows: Consider an element α which is a square of a primitive element $\widehat{\alpha}$ of multiplicative group $\mathbb{Z}_p^*$, i.e. $\alpha = \widehat{\alpha}^2$. (Without loss of generality, we may assume $\alpha < q$.) Then group $\widehat{G}$ can be represented as

$$\widehat{G} = \left\{ \alpha^i \bmod p \mid i \in [1, q] \right\}.$$

An attractive variation of this group is to represent the elements by the integers from 0 to $q - 1$. The group operation is slightly different: Let a funtion f be defined as

$$f(x) = \begin{cases} x & \text{if } x \leq q \\ p - x & \text{if } q < x < p. \end{cases}$$

Using this f function, we can introduce the group G as

$$G = \left\{ f(\alpha^i \bmod p) \mid i \in \mathbb{Z}_q \right\}.$$

Group operation on group G is defined as $a \cdot b = f(a \cdot b \pmod p)$, where $a, b \in G$.

Proposition 4. *Let $g(x) = \alpha^x \bmod p$. Then the function $f \circ g$ is a bijection from $\mathbb{Z}_q$ to $\mathbb{Z}_q$.*

Proof. To see this, suppose $f \circ g(x) = f \circ g(y)$. Then this can be written and $f(X) = f(Y)$ where integer $X = \alpha^x \bmod p$ and $Y = \alpha^y \bmod p$. Now we can have four different cases:

– $X \leq q, Y \leq q$**:** In this case, $f(X) = X$ and $f(Y) = Y$ and hence $X = Y$. Now we have an equation $\widehat{\alpha}^{2(x-y)} = 1 \bmod p$. Since $\widehat{\alpha}$ is a generator for $\mathbb{Z}_p^*$, its order (i.e. $2q$) has to divide $2(x - y)$. This implies that q has to divide $x - y$ and finally $x = y$ since $0 < x, y \leq q$.
– $X > q, Y > q$**:** In this case, $f(X) = p - X$ and $f(Y) = p - Y$ and hence $X = Y$. Rests are same as above.
– $X \leq q, Y > q$**:** This case is impossible, since $\left(\frac{X}{p}\right) = 1$ and $\left(\frac{p-Y}{p}\right) = -1$ since $p \equiv 3 \bmod 4$ and $X = p - Y$.
– $X > q, Y \leq q$**:** This is also impossible by the similar reason.

Therefore, $f \circ g$ is an injection, and hence also a surjection, since the size of domain and images are the same.

Proposition 5. *Note that a distribution r is chosen unformly at random in G, $f \circ g(r)$ is still unform and random in G, since $f \circ g$ is bijective.*

Group of this type is also considered by Chaum [14]. It is generally assumed that DDH is believed to be intractable for these groups [10]. More conretely, **2-party decision Diffie-Hellman assumption on group** G is that for all polynomial time attackers $\mathcal{A}$, for all polynomials $Q(k)$ $\exists k_0$ $\forall k > k_0$, for $X_0 := N_1 N_2$ and $X_1 := N_3$ with $N_1, N_2, N_3 \in_{\mathcal{R}} G$ uniformly chosen, and for a random bit b, the following equation holds:

$$\left| Prob[\mathcal{A}(1^k; G; \alpha; \alpha^{N_1}; \alpha^{N_2}; \alpha^{X_b}) = b] - 1/2 \right| < 1/Q(k)$$

[11] In order to achieve the security level 2^{-k}, the group size should be at least 2^{2k} [26].

A.2 Decisional Binary Tree Group Diffie-Hellman Problem

In this section we define the DTGDH problem (and assumption) and prove this problem is equivalent to 2-party decisional Diffie-Hellman problem. Figure 17 is an example of a key tree when $n = 8$.

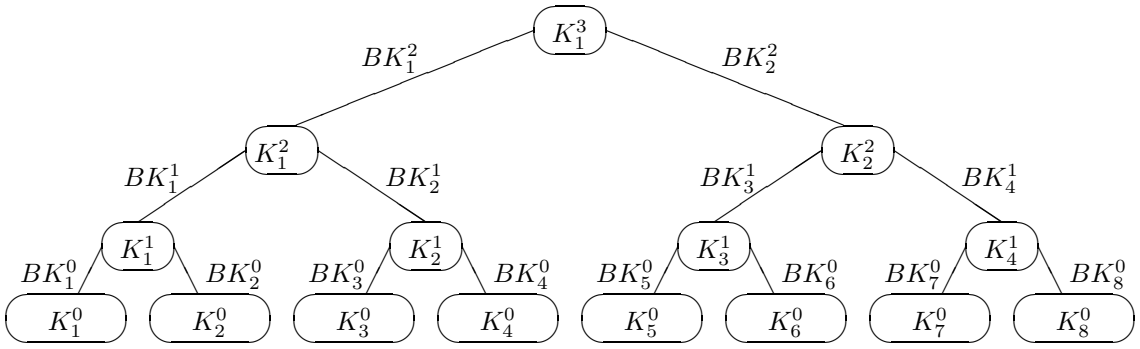

Fig. 17. Notations for fully balanced binary tree

For $(q, \alpha) \leftarrow gen(k), n \in \mathbb{N}$ and $X = (N_1, N_2, \ldots, N_n)$ for $N_i \in G$ and a key tree T with n leaf nodes which correspond to N_i, we can define the following random variables:

- K_j^i: i-th level j-th key (secret information), each leaf node is associated with a member's session random, i.e. $K_j^0 = N_k$ for some $k \in [1, n]$.
- BK_j^i: i-th level j-th blinded key (public information), i.e. $\alpha^{K_j^i}$
- K_j^i is recursively defined as follows if K_j^i is not located on the leaf node:

$$\begin{aligned} K_j^i &= \alpha^{K_{2j-1}^{i-1} K_{2j}^{i-1}} \\ &= (BK_{2j-1}^{i-1})^{K_{2j}^{i-1}} \\ &= (BK_{2j}^{i-1})^{K_{2j-1}^{i-1}} \end{aligned}$$

  In other words, we can cosider $K_j^i = f_1(N_1, N_2, \ldots, N_n)$ for some function f_1, and hence $BK_j^i = \alpha^{f_1(N_1, N_2, \ldots, N_n)} = f_2(N_1, N_2, \ldots, N_n)$ for some funtion f_2.

$(q, \alpha) \leftarrow gen(k), n \in \mathbb{N}$ and $X = (N_1, N_2, \ldots, N_n)$ for $N_i \in G$ and a key tree T with n leaf nodes which correspond to N_i, let:

$$\begin{aligned} view(q, \alpha, h, X, T) &:= \{BK_j^i \text{ where } j \text{ and } i \text{ are defined according to the the key tree } T\} \\ &= \{\alpha^{K_j^i} \bmod p \text{ where } j \text{ and } i \text{ are defined according to the key tree } T\} \\ K(q, \alpha, h, X, T) &:= \alpha^{K_1^{h-1} K_2^{h-1}} \end{aligned} \tag{1}$$

For (q, α) are obvious from the context, we omit them in $view()$ and $K()$. Note that $view(h, X, T)$ is exactly the view of the adversary in TGDH, where the final secret key is $K(h, X, T)$. Let the following two random variables be defined by generating $(q, \alpha) \leftarrow gen(k)$ and choosing X randomly from G and choosing key tree T randomly from all binary trees having n leave nodes:

- $A_h := (view(h, X, T), y)$
- $F_h := (view(h, X, T), K(h, X, T))$

Let the operator "$\approx_{poly}$" denote polynomial indistinguishability.

Proposition 6. *[29] Let K and R be l-bit strings such that R is a random and K is a Diffie-Hellman key. We say that K and R are* **polynomially indistinguishable** *if, for all polynomial time distinguishers, A, the probability of distinguishing K and R is smaller than $(\frac{1}{2} + \frac{1}{Q(k)})$, for all polynomial $Q(l)$.*

Now we can define DTGDH algorithm concretely:

Definition 7. Let $(q, \alpha) \leftarrow gen(k), n \in \mathbb{N}$ and $X = (N_1, N_2, \ldots, N_n)$ for $N_i \in G$ and a key tree T with n leaf nodes which correspond to N_i, and A_h and F_h is defined as above. **DTGDH algorithm** $\mathcal{A}$ for group G is a probabilistic polynomial time algorithm satisfying, for some fixed $k > 0$ and sufficiently large m:

$$|Pr[\mathcal{A}(A_h) = \text{“}True''] - Pr[\mathcal{A}(F_h) = \text{“}True'']| > \frac{1}{m^k}.$$

Accordingly, **DTGDH problem** is to find an Binary Tree DDH algorithm.

Now, we prove that DTGDH problem is hard to the passive advesary: If 2-party DDH on the group G defined above, Binary Tree DDH problem is hard.

Using the polynomial indistinguishabilty, we can restate DTGDH problem in the definition 7: Find the polynomial distinguisher $\mathcal{A}$ which can distinguish A_h and F_h defined above.

Theorem 8. *If 2-party DDH on G is hard, then $A_h \approx_{poly} F_h$.*

Proof. We first note that A_h and F_h can be rewritten in the following way if $X_L = (R_1, R_2, \ldots, R_k)$ and $X_R = (R_{k+1}, R_{k+2}, \ldots, R_n)$ where R_1 through R_k are associated with leaf node in the left tree T_L and R_{k+1} through R_h are in the right tree T_R:

$$\begin{aligned}
A_h &:= (view(h, X, T), y) \quad \text{for random } y \in G \\
&= (view(h-1, X_L, T_L), view(h-1, X_R, T_R), BK_1^{h-1}, BK_2^{h-1}, y) \\
&= (view(h-1, X_L, T_L), view(h-1, X_R, T_R), \alpha^{K_1^{h-1}}, \alpha^{K_2^{h-1}}, y) \\
F_h &:= (view(h, X), K(h, X)) \\
&= (view(h-1, X_L, T_L), view(h-1, X_R, T_R), BK_1^{h-1}, BK_2^{h-1}, \alpha^{K_1^{h-1}K_2^{h-1}}) \\
&= (view(h-1, X_L, T_L), view(h-1, X_R, T_R), \alpha^{K_1^{h-1}}, \alpha^{K_2^{h-1}}, \alpha^{K_1^{h-1}K_2^{h-1}})
\end{aligned}$$

We prove this theorem by induction and contradiction. Note that 2-party DDH of G is equivalent to distinguish A_1 and F_1. We assume that A_{h-1} and F_{h-1} are indistinguishable in polynoimial time for induction hypothesis. We further assume that there exists a polynomial algorithm that can distinguish between A_h and E_h for a random binary tree. We will show that this algorithm can be used to distinguish A_{h-1} and E_{h-1} or can be used to distinguish 2-party DDH problem.

Consider the following equations:

$$\begin{aligned}
A_h &:= (view(h-1, X_L, T_L), view(h-1, X_R, T_R), \alpha^{K_L^{h-1}}, \alpha^{K_R^{h-1}}, y) \\
B_h &:= (view(h-1, X_L, T_L), view(h-1, X_R, T_R), \alpha^{r}, \alpha^{K_R^{h-1}}, y) \\
C_h &:= (view(h-1, X_L, T_L), view(h-1, X_R, T_R), \alpha^{r}, \alpha^{r'}, y) \\
D_h &:= (view(h-1, X_L, T_L), view(h-1, X_R, T_R), \alpha^{r}, \alpha^{r'}, \alpha^{rr'}) \\
E_h &:= (view(h-1, X_L, T_L), view(h-1, X_R, T_R), \alpha^{r}, \alpha^{K_2^{h-1}}, \alpha^{rK_2^{h-1}}) \\
F_h &:= (view(h-1, X_L, T_L), view(h-1, X_R, T_R), \alpha^{K_L^{h-1}}, \alpha^{K_R^{h-1}}, \alpha^{K_L^{h-1}K_R^{h-1}})
\end{aligned}$$

Since we can distinguish A_h and D_h in polynomial time, we can distinguish at least one of (A_h, B_h), (B_h, C_h), (C_h, D_h), (D_h, E_h), or (E_h, F_h).

– A_h and B_h: Suppose one can distinguish A_h and B_h in polynomial time. We will show that this distinguisher $\mathcal{A}_{AB_h}$ can be used to solve Binary Tree DDH problem with height $h-1$. Suppose We want to decide $P'_{h-1} =$

$(view(h-1, X', T'), r')$ is an instance of Binary Tree DDH problem or r' is a random number. To solve this problem, we generate another tree T'' of height $h-1$ with a distribution X''. Note that we know complete secret and public informations of T''. Using P'_{h-1} and (T'', X''), we can generate a distribution

$$P'_h = (view(h-1, X', T'), view(h-1, X'', T''), \alpha^{r'}, \alpha^{K(h-1, X'', T'')}, y)$$

where $y \in_{\mathcal{R}} G$. Now we put P'_h as an input of $\mathcal{A}_{AB_h}$. If P'_h is an instance of A_h (B_h), then P'_{h-1} is an instance of F_{h-1} (A_{h-1}) by Proposition 5, repectively.

– B_h **and** C_h**:** Suppose one can distinguish B_h and C_h in polynomial time. We will show that this distinguisher $\mathcal{A}_{BC_h}$ can be used to solve Binary Tree DDH problem with height $h-1$. Suppose We want to decide $P'_{h-1} = (view(h-1, X', T'), r')$ is an instance of Binary Tree DDH problem or r' is a random number. To solve this problem, we generate another tree T'' of height $h-1$ with a distribution X'' and choose $r'' \in_{calR} G$. Note that we know complete secret and public informations of T''. Using P'_{h-1} and (T'', X''), we can generate a distribution

$$P'_h = (view(h-1, X'', T''), view(h-1, X', T'), \alpha^{r''}, \alpha^{r'}, y)$$

where $y \in_{\mathcal{R}} G$. By Proposition 5, r'' is still random and uniform in G. Now we put P'_h as an input of $\mathcal{A}_{BC_h}$. If P'_h is an instance of B_h (C_h), then P'_{h-1} is an instance of F_{h-1} (A_{h-1}) by Proposition 5, respectively.

– C_h **and** D_h**:** Suppose we can distinguish C_h and D_h in polynomial time. We will show that this distinguisher $\mathcal{A}_{CD_h}$ can be used to solve 2-party DDH problem on group G. Note that $\alpha^r, \alpha^{r'}$ are independent variable from $view(h-1, X_L, T_L)$ and $view(h-1, X_R, T_R)$. Suppose we want to test whether $(\alpha^a, \alpha^b, \alpha^c)$ is a DDH triple or not. To solve this problem, we generate two key trees T_1 and T_2 of height $h-1$ with distributions X_1 and X_2, respectively. Now we generate a new distribution

$$P'_h = (view(h-1, X_1, T_1), view(h-1, X_2, T_2), \alpha^a, \alpha^b, \alpha^c).$$

If P'_h is an instance of C_h (D_h), then $(\alpha^a, \alpha^b, \alpha^c)$ is an valid (invalid) DDH triple, repectively.

– D_h **and** E_h**:** Suppose one can distinguish D_h and E_h in polynomial time. We will show that this distinguisher $\mathcal{A}_{DE_h}$ can be used to solve Binary Tree DDH problem with height $h-1$. Suppose We want to decide $P'_{h-1} = (view(h-1, X', T'), r')$ is an instance of Binary Tree DDH problem or r' is a random number. To solve this problem, we generate another tree T'' of height $h-1$ with a distribution X''. Note that we know complete secret and public informations of T''. Using P'_{h-1} and (T'', X''), we can generate a distribution

$$\begin{aligned} P'_h &= (view(h-1, X', T'), view(h-1, X'', T''), \alpha^{r'}, \alpha^{r''}, (\alpha^{r'})^{r''}) \\ &= (view(h-1, X', T'), view(h-1, X'', T''), \alpha^{r'}, \alpha^{r''}, \alpha^{r'r''} \end{aligned}$$

where $r'' \in_{\mathcal{R}} G$. Note that since we generate r'', we can compute $(\alpha^{r'})^{r''}$. Now we put P'_h as an input of $\mathcal{A}_{DE_h}$. If P'_h is an instance of D_h (E_h), then P'_{h-1} is an instance of F_{h-1} (A_{h-1}) by Proposition 5, repectively.

– E_h **and** F_h**:** Suppose one can distinguish E_h and F_h in polynomial time. We will show that this distinguisher $\mathcal{A}_{EF_h}$ can be used to solve Binary Tree DDH problem with height $h-1$. Suppose We want to decide $P'_{h-1} = (view(h-1, X', T'), r')$ is an instance of Binary Tree DDH problem or r' is a random number. To solve this problem, we generate another tree T'' of height $h-1$ with a distribution X''. Note that we know complete secret and public informations of T''. Using P'_{h-1} and (T'', X''), we can generate a distribution

$$\begin{aligned} P'_h &= (view(h-1, X', T'), view(h-1, X'', T''), \alpha^{r'}, \alpha^{K(h-1.X'', T'')}, (\alpha^{r'})^{K(h-1.X'', T'')}) \\ &= (view(h-1, X', T'), view(h-1, X'', T''), \alpha^{r'}, \alpha^{K(h-1.X'', T'')}, \alpha^{r'K(h-1.X'', T'')} \end{aligned}$$

where $r' \in_{\mathcal{R}} G$. Note that since we generate r', we can compute $(\alpha^{r'})^{K(h-1.X'', T'')}$. Now we put P'_h as an input of $\mathcal{A}_{EF_h}$. If P'_h is an instance of E_h (F_h), then P'_{h-1} is an instance of F_{h-1} (A_{h-1}) by Proposition 5, repectively.

A.3 Group Key Secrecy

Before considering the group key secrecy, we briefly examine key freshness. Every group key is fresh, since at least one member in the group generates its random key share uniformly for every membership change[12]. The probability that new group key is same as any old group key is negligible due to bijectiveness of $(f \circ g)$ function.

[12] Recall that insider attacks are not our concern. This excludes the case when an insider intentionally generates non-random number.

We note that the root (group) key is never used directly for the purposes of encryption, authentication or integrity. Instead, such special-purpose sub-keys are derived from the root key, e.g., by applying a cryptographically secure hash function to the root key, i.e. $H(group\ key)$ is used for such applications.

As discussed in Section 3, decisional group key secrecy is more meaningful if sub-keys are derived from a group key. Decisional group key secrecy of TGDH protocol is somewhat related to DTGDH assumption mentioned in Section A.2. This assumption ensures that there is no information leakage other that public blinded key informations.

We can also derive the sub-keys based on Shoup's hedge [27]: Compute the key as $H(group\ key) \oplus \mathcal{H}(group\ key)$ where $\mathcal{H}$ is a random oracle. It follows that in addition to the security in the standard model based on DTGDH assumption, the derived key is also secure in the random oracle model [9] based on Computational Tree-based Group Diffie-Hellman assumption.

A.4 Key Independence

We now give an informal proof that TGDH satisfies forward and backward secrecy, or equivalently key independence. In order to show that TGDH provides key independence, we only need to show that the *view* of the former (prospective) member to the current tree is exactly same as the *view* of the passive adversary respectively, since this shows that the advantage of the former (prospective) member is same as the passive adversary and by Theorem 8.

We first consider backward secrecy, which states that a new member who knows the current group key cannot derive any previous group key. Let M_{n+1} be the new member. The sponsor for this join event changes its session random and, consequently, previous root key is changed. Therefore, the *view* of M_{n+1} with respect to the prior key tree is exactly same as the *view* of an outsider. Hence, the new member does not gain any advantage compared to a passive adversary.

This argument can be easily extended to the merge of two or more groups. When a merge happens, sponsor in each tree changes its session random. Therefore, each member's *view* on other member's tree is exactly same as the *view* of a passive adversary. This shows that the newly merged member has exactly same advantage about any of the old key tree as a passive adversary.

Now we consider the forward secrecy, meaning that a passive adversary who knows a contiguous subset of old group keys cannot discover subsequent group keys. Here, we consider partition and leave at the same time. Suppose M_d is a former group member. Whenever subtractive event happens, a sponsor refreshes its session random, and, therefore, all keys known to leaving members will be changed accordingly. Therefore, M_d's *view* is exactly same as the *view* of the passive adversary.

This proves that TGDH provides decisional version of key independence.