TECHNICAL RESEARCH REPORT

Key Distribution Protocols for Multicast Group Communication in MANETs

by Maria Striki and John S. Baras

CSHCN TR 2003-10
(ISR TR 2003-17)

01-09-03

Report for Key Distribution No2:

Introduction

Network Security for Wireless communications is a topic that gets more and more attention every day. The ongoing number of users involved in transactions that require the utmost security, or the development of wireless multicast services that might be paid or not such as cable TV, secure audio and visual broadcasts, secure conferencing, military command and control, draws the attention of the research on security for wireless multicasting. The subtle issue in wireless multicasting is that even though a message from the source is broadcast in a distance and range determined from the power transmission limits and bandwidth of the device, we only want this message to be received from a special group of users that is certified to obtain the message, or that acquires the credentials that allow this group and only this, to get the message. What we do is encrypt the messages intended for a user or for a group of users with some special keys so that only the members of this group that acquire the appropriate keys to decrypt the message can obtain it. This is where Network Security comes in. It provides us with a variety of protocols and policies in general, for encrypting and decrypting messages. However, since the number of multicast groups of particular services continually grows we are not interested so much in providing solely security isolated from scalability of protocols. It is obvious that if there is not scalability in so large a system, then the security on them is redundant. So, since we have so many users, we need to reduce the amount of communication overhead, to reduce the storage and computation cost that come along, so that the transactions are fast and as cheap as possible. The devices have limited storage, computational and communication capacities. Devices also acquire limited bandwidth, capacity, computational and transmission power. This is why the communication overhead, the computation and storage cost are an issue and have to be reduced as much as possible to make the systems of today scalable to groups of perhaps millions of users.

Research focuses on the following aspects for scalable wireless multicast ad-hoc networks: Key Management and particularly Key Distribution, Protocols for Public Key Encryption/ Decryption and user authentication. Users can be pre-authenticated by a Certification Authority (CA) before they start transactions within a group. We still have to check whether a user is authenticated or not every time we wish to communicate with this user. In this work I assume that users are already authenticated. The key distribution plays a very important role in manipulating the communication, the storage and the computation costs. Two popular key management protocols as of today are: GKMP (Group key Management Protocol) - a heavily centralized one, and the Logical Tree Key Protocol. In what follows we develop their functionality principles, their advantages and disadvantages. Then, we do the same for more popular protocols like OFT, ELK, GDH.1, GDH.2, GDH.3, 2^d-Octopus and hybrid extensions etc. In this work we will attempt to combine these protocols with all the possible combinations into a two-layered hybrid scheme that exploits advantages of both protocols and reduces possibly their overhead. We will develop models of these combinations, redefine their function principles and we will do a mathematical analysis in order to derive all costs. The analytical formulae of these models will provide us with the tools for comparisons among them, and push the research towards the best behaving models. Perhaps some entirely new key management scheme could be developed that further reduces overhead. Numerous constraints that are inherent on secure wireless ad-hoc communications limit the variety of Key Distribution schemes that could be robust and scalable in a Mobile Ad-Hoc Wireless Network (MANET).

Another improvement in scalability might come from the selection of Public Key Encryption/ Decryption schemes that are going to be used as we are going to explain later in this work. Public Key encryption/decryption (e.g. RSA) is far slower and more expensive than Symmetric key encryption/decryption (e.g DES). Re-keying is very expensive and should be reduced as much as possible, without destroying the security of the group. We assume that the members, the Group Security Agents (GSAs) and Group Security Controllers (GSCs) are already authenticated.

Background and Notation

Motivation for Multicast: Efficiency in network resource utilization.
Goal: Only valid group members have access to current group data
Requirements:
–Perfect Forward Secrecy: The deleted entity should have no access whatsoever in messages exchanged among nodes of its previous group after it has been deleted.
–Perfect Backward Secrecy: An entity that has been added to a group should have no access in previous communication among members of its current group.
-Keys must be changed at every membership update and also periodically (re-keying).

Policy for the PKI

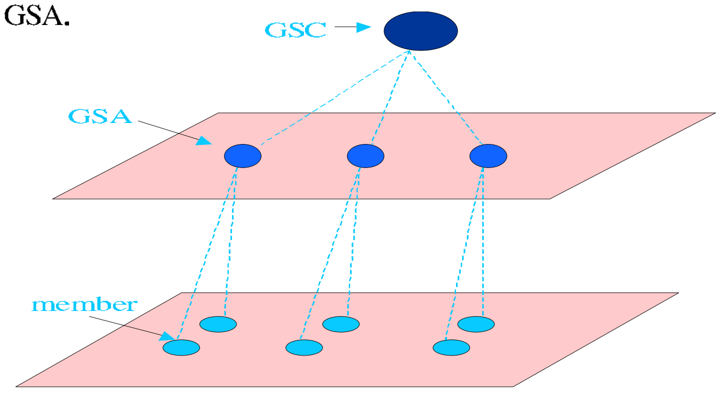
Of course, we assume that there exist a standard PKI according to which nodes can securely communicate with each other. For that, we use the RSA Public Key Encryption/Decryption method. According to this method, each node is assigned a public and a private key. The public key is published for all users, and the private key is known only to the particular member. So, any other member that is intending to communicate with this node encrypts a message with the node's public key, and the node decrypts the encrypted message with its private key. However, this method is very expensive and we would like to find a way to achieve cheaper and faster communication among nodes without jeopardizing the security of the system. What can be done, is a compromise between the slow and secure RSA method and the simpler, faster, but less secure symmetric Encryption/Decryption method (the sender and the receiver use the same secret key for encryption and decryption of the message respectively). The symmetric encryption is less secure mainly because the secret key can be compromised relatively easy in comparison to RSA secret key. Thus, we do as follows: We are going to use the symmetric key method for the communication of messages among the group, but the keys that are going to be used for the symmetric method are going to be communicated to the members with the use of Public Key Encryption/ Decryption method. Thus, for every new member, the GSC creates a new secret key. It encrypts it with the node's RSA public key and sends this encrypted message to the node. The node, decrypts the message containing the secret key with its private key known only to itself and thus gets the secret key that is known both to the GSC and itself, and with which it will be communicating privately with the GSC from now on. Thus, we use the Public Key method as a secure channel to communicate the secret keys to the nodes that are going to be used by the Symmetric Key method. This way we achieve faster and cheaper communication and computations without harming the security of the system.

PARAMETERS:

n: number of users, *K*: length of the key, *d:* number of descendants of a tree node, h: height of the tree. C_r: random function generator, computational cost per PKI encryption/decryption *(CPE /CPD),* computational cost per symmetric key encryption/decryption *(CSE /CSD), p*: probability of inserting a new member or GSA or deleting a member or GSA.

Two-Level Hierarchical Model

First level: GSCs – GSAs.
Second level: GSA- Members.
Hierarchical Key Management.
Different probability/capacity.
Independent subgroups.



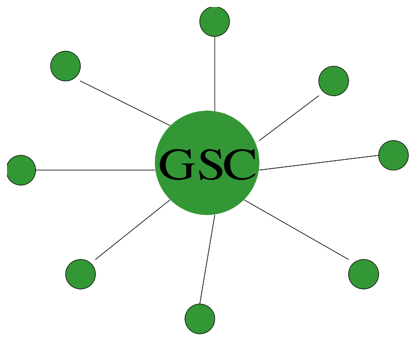
Independent subgroup key management.
Update limited to subgroup.
K^1_{ij}: GSA to member K^2_i: GSA to cluster, K^3: GSC to members.

MANET Constraint: Existence of robust nodes with sufficient capabilities to become group leaders. The two level hybrid scheme exploits the diversity of the battlefield: links key distribution to network topology, hierarchy, (un)predicted member mobility, routing. The network consists of heterogeneous nodes that result in producing links of variable qualities, uni/bi-directional asymmetric paths, larger bandwidth resources at higher tiers (satellites), restricted at lower (cellphones, laptops), different physical/communication mobility levels. At the low end mobility is more rapidly changing and higher degree of self-organization is observed. Nodes often need intermittent connectivity to reach others at the higher end. The scheme we designed models these environmental variations so that the first (second) level of hierarchy represents nodes at the higher (lower) end.

The Hybrid Hierarchical Key Scheme has adaptability to the network limitations and topology. It can use a combination of protocols to achieve either substantial communication overhead reduction, or lower the computation cost, or both. What is most important is that this model can support almost all protocols at any time. Given the parameters of the network it can select the most appropriate for the time schemes for each of its levels.

GKMP (Group Key Management Protocol)

- •Single Group Controller GSA
- •Key pair for member i: $GKP_i = \{SEK, KEK_i\}$
- •Simple, small storage versus large communication overhead for membership update.
- •Can be used as the organization scheme within each subgroup of GSA in the hierarchical model.



Evaluation of Cost for GKMP

Initially the GSC creates the n secret keys it is going send to the n members of the group via the Public Key method. Then, it creates a group session key for multicast communication. It communicates the session key to each member individually using the symmetric encryption method this time. The session key is encrypted with the private key it was just earlier sent to each member.

GSC Storage Cost: The GSC needs to store the secret key of each member as well as the session key. Thus, the GSC needs to store $(n+1)$ keys.

Member Storage Cost: Each member needs only store the secret key with which it will individually communicate with the GSC, and the session key. Thus, each member stores two keys.

Initial GSC Computation: The GSC creates $(n+1)$ new keys: the n secret keys of each member and the session key. The cost for RSA random key generation is denoted by the function C_r (prime random numbers random key generation). Then, the GSC does a public encryption for each of the n secret keys, with cost C_{PE} each (C_{PE}: Cost for Public Key Encryption). It then unicasts the session key to each member, encrypted via the Symmetric Encryption method, and thus the GSC does n symmetric encryptions with cost C_{SE} each (C_{SE}: Cost for Symmetric Encryption).

Initial Member Computation: Each member first decrypts the secret key sent to it by GSC using the Public Key Decryption method with cost C_{PD}. Then, it decrypts via the Symmetric Key Decryption method with cost C_{SD}, the session key sent to it afterwards by the GSC.

Initial Communication: The GSC communicates first the n secret keys to the n members and then sends individually to every member the session key (n unicast messages). Thus, the total Initial Communication Cost is $2nK$.

Add GSC Computation: When a new member is added to the group the GSC must change its session key to ensure that the new member can have no access to previous group communication (backward secrecy). First it creates a new secret key and sends it to the new member via the Public Key encryption method. Then it creates a new session key. It broadcast the new session key to all members but the new, encrypted with the previous session key, so that only the previous members of the group can decrypt the new session key and unicasts the new session key to the new member encrypted via the Symmetric Key Encryption method with the secret key it previously sent to it.

Add Member Computation: The new member must decrypt via the Public Key Decryption method the secret key sent to it by GSC and then, it must decrypt via the Symmetric encryption method the session key sent to it afterwards by the GSC. The rest of the members need only decrypt the new session key sent by the GSC and encrypted with their secret keys via Symmetric Encryption.

Add Communication: The GSC communicates a secret key to the new member and then it broadcasts the new session key to all members but the new and also unicast it to the new member. Thus, the communication cost here is $3K$, where K is the length of any key in bits.

Delete GSC Computation: When a new member is deleted from the group, the GSC must change its session key to ensure that the evicted member can have no access to future group communication (forward secrecy). The GSC creates a new session key (cost for creating a key by applying the random prime number key generator: C_r), communicates the new session key to each remaining member individually, encrypted via the symmetric method with the secret key of each member. Thus, the GSC does (n-1) symmetric encryptions.

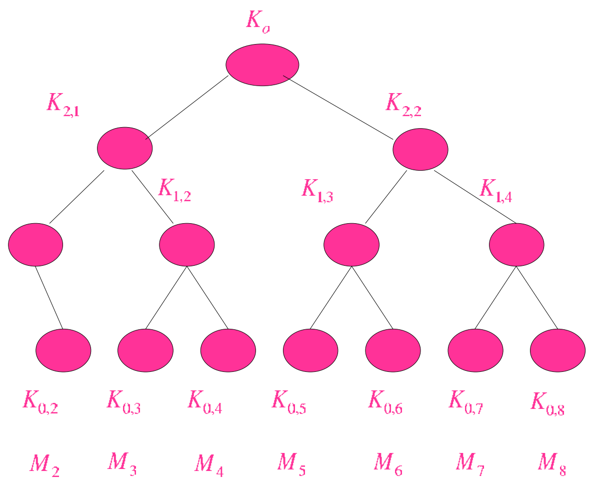
Delete Member Computation: Each member just decrypts the new session key sent to it via the symmetric key method.

Delete Communication: The GSC sends the new session key to the remaining (n-1) members.

Tree Based Key Management

This scheme extends the logical Tree hierarchy. The leaf nodes are the group members and the group key is the key associated with the root of the tree. Each member knows all keys from its leaf node up to the root node, but no other node in the tree.
Member joins group: key server authenticates it, assigns it to a leaf node, it receives all keys on the key path to the root. All keys it receives are independent from any previous keys (preserve backward secrecy). Key server replaces all keys on the node member's key path with fresh keys and sends each of these new keys to the group on a "need to know" basis.

Member leaves group: all keys the evicted member knows will be changed (ensure forward secrecy), keys replaced sequentially from the leaf up to the root key. Leaves are efficient because they only require updating $log_d(n)$ keys, n is the number of members, assuming a balanced tree.



Example:
Key path of member M2 is nodes associated with the keys {*K5, K2, K1*}.New member M4 joins in empty leaf. Server chooses new keys on the path of M4 and sends K'7, K'3, K'1 to M4 over secure link. To update the path server broadcasts the following messages to the group: *{K'3}K6, {K'1}K'3, {K'1}K2*. Member K3 leaves the group. Key server updates K1, K3 and generates new keys: K'1, K'3. It then broadcasts: $\{K'3\}_{K'7}$, $\{K'1\}_{K'3}$, $\{K'1\}_{K2}$

Key update- member 1 leaves:

$$\{\{K_{1,1}, K_{2,1}, K_o\}_{K_{0,2}},$$
$$\{K_{2,1}, K_o\}_{K_{1,2}}, \{K_o\}_{K_{2,2}}\}$$

Evaluation of Cost for CBT:
Initially the GSC (associated with the root of the tree) creates the n secret keys it is going send to the n members of the group via the Public Key method. Each member is associated with a leaf of the tree. Then, the GSC creates secret keys for all internal nodes of the tree including itself. It communicates to each member the keys associated with the nodes that belong to the path of the member from its leaf up to the root. Thus, the GSC communicates approximately h secret keys to each member (h is the height of the tree) to each member via the symmetric encryption encrypted with the secret key it previously sent to each of them. The key associated with the node that is the root of the tree is going to be the session key for all multicast group communication.

GSC Storage Cost: The GSC needs to store all the keys of the tree. *We have to note here that the secret key associated with an internal node may be used by the GSC to encrypt a message intended to be received and successfully decrypted only by those members that contain this internal node in their paths from the leaf up to the root.* The number of all nodes in a balanced d-tree is: $d^0+d^1+d^2+\ldots+d^h = (d^{h+1}-1)/(d-1) = (dn-1)/(d-1)$.

Member Storage Cost: Clearly, each member needs to store all (h+1) keys associated with the nodes that lie at the path of the leaves up to the root.

Initial GSC Computation: The GSC creates as many new keys as the number of nodes in the tree, thus (dn-1)/(d-1) new keys by the random prime key generator with cost C_r. Out of these keys, it encrypts via the public key encryption method the n keys that it is going to communicate to the members. The rest of the nodes, namely (dn-1)/(d-1)–n = (n-1)/(d-1), are sent to the members encrypted via the symmetric encryption method. In order to spare communication cost, the symmetric encryption takes place gradually as follows: internal nodes that belong to the same level of the balanced tree are simultaneously encrypted. Each such node is encrypted by the GSC d times with the secret keys of the d nodes that reside in the same path as the particular internal node and belong to the exact below level of the tree. Then, the GSC broadcasts each such internal node d times. The members that can decrypt any one of those d messages are only those that contain in their path up to the root the nodes associated with the keys of which the internal nodes has been encrypted. Since these nodes belong to the immediately lower level of the tree, the members clearly already know the secret keys of those d nodes of the lower level and can thus decrypt one of these encrypted messages and derive the secret key of the particular internal node. This way, it suffices that each internal node of a particular level in the tree is encrypted d times with the secret keys associated with its children nodes and that this procedure is repeated for every successive level towards the root, to ensure that the appropriate keys have been sent to the appropriate members. Thus, the GSC has to do d (n-1)/(d-1) symmetric encryptions with cost C_{SE} each.

Initial Member Computation: Each member decrypts the secret key associated with its leaf sent by the GSC via the Public Key Encryption method. With this key it later decrypts the h keys sent to it by the GSC with the Symmetric Key Encryption method.

Initial Communication: The GSC sends to the n members their n secret keys associated with their leaves. Then, as we described before, it sends d (n-1)/(d-1) encrypted messages that will be decrypted by the members with the symmetric decryption method, and the appropriate members and only those will be able to decrypt the appropriate secret keys of their path up to the root.

Add GSC Computation: The GSC places the new member to an empty leaf if there is nay free leaf of else an existing leaf node x is split, the member associated with x is now associated with left(x), and the new member is associated with right(x). The keys of all nodes residing in the path of the new member from its leaf up to the root must be updated so that the member has no access in previous communications (backward secrecy). Thus h keys need to be updated and the GSC generates h+1 new keys. It sends to the new member its secret key with the Public Key Encryption Method. The rest h keys are sent to the new member encrypted via the symmetric key encryption method with the secret key it earlier received, but they are first sent to the members that include the updated nodes to their path from the leaves to the root. The new key of each node to be updated is broadcast to the members encrypted by the node's current secret key. This way, the updated keys can be decrypted only by the members that depend on them. Clearly, the GSC must do 2h symmetric encryptions with cost C_{SE}.

Add Member Computation: The new member must decrypt via the Public Key Decryption method the secret key sent to it by GSC and then, it must decrypt via the Symmetric encryption method the h keys sent to it afterwards by the GSC. The rest of the members decrypt from 1 to h such encrypted messages sent by the GSC, depending on how many of the nodes with updated keys reside in their paths from the leaves up to the root. Actually, n/2 members do only one, n/4 do 2,…,$n/2^h$ do h decryptions etc. In average, each member does 2 symmetric decryptions.

Add Communication: The GSC must send the secret key associated with the leaf of the new member once, and then it must unicast the h updated keys associated with the path of the new member encrypted with the secret key of the new member and it must broadcast those h updated keys to the rest of the members. Thus, the total communication cost is (2h+1) K.

Delete GSC Computation: The GSC must update all keys that are associated with nodes that belong to the path of the evicted member up to the root, so that the evicted member can have no longer access to future communication (forward security). The GSC creates h new keys to update this path. The updated keys must be sent to the members that include the updated nodes to their path to the root. Of course, we cannot send the new keys of the nodes in this path encrypted with their current keys that members already acquire, because those keys are known to the evicted member as well. So, for each updated node, the GSC sends its new key to all its (d-1) children nodes but the one child that is also updated, encrypted (via the symmetric encryption method) each time with the secret key of each of the (d-1) children nodes. The process is sequentially repeated for all the h updated nodes in this path up to the root, so the GSC does (d-1) h symmetric encryptions in total, with cost C_{SE} each.

Delete Member Computation: Each member only needs to do as many symmetric decryptions as the number of nodes that belong to their path to the root that have their keys updated. As in the addition case, in average each member does 2 such decryptions. The members that were siblings of the evicted member according to the tree scheme do h symmetric decryptions.

Delete Communication: The GSC, for each updated node, sends the new key encrypted to (d-1) children. It does the same for all h updated nodes, thus the total cost is (d-1) h K.

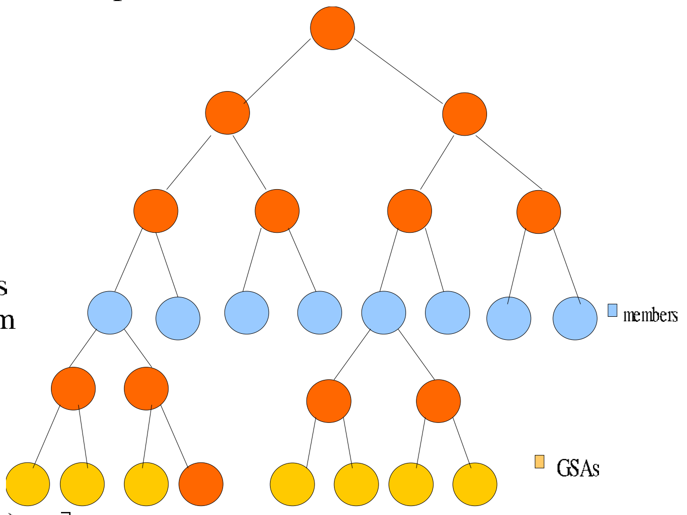
Parameters	GKMP	CBT
GSC Storage	(n+1) K	(dn-1) K/ (d-1)
Member Storage	2 K	(h+1) K
Initial GSC computation	(n+1) C_r +n C_{PE} +nC_{SE}	(dn-1)C_r /(d-1)+n C_{PE}+d(n-1)C_{SE} /(d-1)
Initial member computation	C_{PD} + C_{SD}	C_{PD} + h C_{SD}
Initial Communication	2 n K	(n + d(n-1)/(d-1)) K
Add GSC computation	2 C_r + C_{PE} + 2 C_{SE}	(h+1) C_r + C_{PE} + 2 h C_{SE}
Add members computation	C_{PD} + C_{SD}	C_{PD} + h C_{SD}
Add Communication	3 K	(2h + 1) K
Delete GSC computation	C_r + (n-1) C_{SE}	h C_r + (d-1) h C_{SE}
Delete member computation	C_{SD}	h C_{SD}
Delete communication	(n-1) K	(d -1) h K

Development of Hybrid Models and Analysis of Costs for each Model

First Scheme Architecture: Single Key Tree, Two Groups of Members.

n_1: # GSAs (clusters)
p_1: probability of GSA addition/deletion
n_2: # members in each cluster.
p_2: probability of member addition/deletion.
p1<p2 (p1, p2: motion frequencies).

So in this case we have an unbalanced tree:
$h_1 = -\log_d p_1$ and $h_2 = -\log_d p_2 < h_1$
The GSC distributes the keys itself both to GSAs and to members. So keys are decrypted both from GSAs and from members respectively.



Storage Cost:

GSC:
$$\left\lceil \left(\frac{d^{h_2+1}-1}{d-1} + n_1 \frac{d}{d-1} \cdot \frac{d^{h_1-h_2}-1}{d^{h_1-h_2}} \right) K \right\rceil = \lceil N_C \cdot K \rceil$$

GSA: $\left\lceil (h_1+1)K \right\rceil$ Member: $\left\lceil (h_2+1)K \right\rceil$

Derivation of N_c:
The first term is the storage cost for the tree of members only. The height of the tree is h_2, so the total number of nodes in this tree and thus the number of keys the GSC needs to store is: $(d^{h2+1}-1)/(d-1)$. Members only are associated with the leaves of a balanced tree with height h_2. The trees of GSAs have their root to any of the members (selected at random). We select any member among all that maintain paths to GSAs. This member is the root to a balanced tree of height $h_1–h_2$. Each leaf of such tree that is not empty is associated with a GSA. As we already know the GSC must store the keys of all nodes of the tree. The total number of nodes in this tree is: $(d\ d^{h1-h2}–1)/(d-1)$. Each such tree has a maximum number of GSAs: d^{h1-h2}. However if the number of GSAs is n_1 then it is obvious that the number of such GSA trees required is: n_1/d^{h1-h2}. Since the GSC has to store keys of all nodes of all the trees, but with this calculations we have taken into account twice those members that also belong t the trees of GSAs. Obviously the number of these members is: n_1/d^{h1-h2}, or for each GSA tree, we need to subtract one node and thus the total number of keys to be stored in each GSA tree is: $(d\ d^{h1-h2}–1) / (d-1)-1= d(d^{h1-h2}-1)/(d-1)$. Thus, the total storage cost for the GSC is Nc, as shown in the equation

Initial Computation Cost:

GSC: $\left\lceil N_C C_r + (n_1 + n_1 n_2) C_{PE} + (N_C - 1) C_{SE} \right\rceil$

GSA: $\left\lceil C_{PD} + h_1 C_{SD} \right\rceil$ Member: $\left\lceil C_{PD} + h_2 C_{SD} \right\rceil$

Derivation of the computation cost for GSC:

Initially N_c new keys will be generated in GSC by the random generator C_r. The GSC first sends the secret keys of the leaves (leaves that correspond to members and GSAs, thus totally $n_1+n_1n_2$ leaves) with the Public Encryption method. Then, GSC sends the secret keys of all the rest of the nodes encrypted with the secret keys of members it previously sent, using the Symmetric Encryption method. In other words, we encrypt the keys of nodes along the path of a member or GSA with the secret key previously sent to this member or GSA respectively. The communication of these symmetric key encrypted messages takes place successively for every level of the tree with direction from the bottom (GSAs) to the top (root), as we have discussed in the typical case of CBTs. The only difference here, is that the members do not communicate their private keys that correspond to the member leaves to their nodes that lie below them, as we would expect. Thus, the keys of the parent nodes of these members that acquire paths to GSAs must be communicated both to the appropriate members and to the nodes that are children of these members. Thus the GSC encrypts the key of such a node d (for its children members themselves) + d (times the number of GSA trees that lie below its children members) times instead of d.
As we have discussed in the CBT cases, the GSC needs to do $d (n1\ n2 -1) / (d-1) = d (d^{h2}-1) / (d-1) = ((d^{h2+1}-1) / (d-1))-1$ symmetric encryptions for the appropriate keys to be communicated to the appropriate members. For the GSA trees the GSC behaves the same: we have to note here that even if the key of the physical root of a GSA tree (namely the key of a member) is not encrypted and communicated d times to the nodes of GSA trees, the key of the parent of such member plays exactly the role of the physical root: it is encrypted and broadcasted d times to the particular GSA tree. Thus, in analogy to the typical CBT case, the GSC needs to do $d(n1-1)/(d-1)$ or $d(d^{h1-h2}-1) / (d-1)$ symmetric encryptions for each such GSA tree. Summing up, the overall number of symmetric encryptions the GSC needs to do is: N_C-1.

Each GSA decrypts its private key sent with use of the Public (asymmetric) method, and then decrypts all the keys that formulate the path up to the root (where the session key is) for the particular GSA with the use of the symmetric method.
The same is the case with each member. The difference is that the GSA decrypts a path of height h_1 whereas the member decrypts a path of height h_2.

Initial Communication Cost: $\left\lceil \{N_C + n_1(n_2+1) - 1\}K \right\rceil$

The communication cost is the number of bits that have to be communicated from GSC to all the GSAs and members so that the proper keys are distributed. For the distribution of keys to all the leaves using the public key method we need: $(n_1 + n_1 n_2)$ K bits. For the distribution of all the keys necessary to all the interior nodes using the symmetric encryption method we need: (N_c-1) K bits.

We assume that the probability for inserting a new GSA or a new member into the group (1/2) is the same as the probability for deleting a GSA or a member from the group (1/2).

- *Average Operating Cost:* Average Computation Cost for the addition or deletion of a GSA or member to the hybrid scheme.

Average Operating Cost for GSC:

$$\frac{1}{2}\{n_1 p_1((h_1+1)C_r + C_{PE} + 2h_1 C_{SE}) + n_1(n_2-1)p_2((h_2+1)C_r + C_{PE} + 2h_2 C_{SE})\} +$$
$$\frac{1}{2}\{n_1 p_1(h_1 C_r + (d-1)h_1 C_{SE}) + n_1(n_2-1)p_1(h_2 C_r + (d-1)h_2 C_{SE})\}$$

Both GSAs and members are viewed as "members" from the perspective of the GSC. In the case of a new GSA or member inserted into the group the GSC is responsible for creating and distributing the appropriate encrypted keys both for the case of a GSA addition (first term) and for the case of a member addition (second term). These two terms are derived exactly as we have analyzed in the CBT protocol for member addition. The third and fourth term represent the computation cost of a GSC for deleting a GSA or a member from the tree respectively. Similarly, the terms are derived exactly as we have analyzed in the CBT protocol for the case of member deletion.

Average Operating Cost for GSAs: $n_1 p_1 (C_{PD} + h_1 C_{SD})$

The task of the GSA in this scheme is the following: When we insert a new GSA it needs to decrypt the private key sent from the GSC via the public key encryption method and then decrypt the private keys associated with its path up to the root sent by the GSC using the symmetric key encryption method. The update for the changed keys is done by broadcasting the new keys encrypted with the keys of the nodes that are siblings to the nodes associated with the updated keys. When we delete a GSA, add or delete a member no actions have to be taken by any other GSA, since the GSC is responsible to carry out the latter operations in this scheme.

Average Operating Cost for Members: $n_1(n_2-1)p_2(C_{PD} + h_2 C_{SD})$
Similar is the course of thought for deriving this formula for the addition or deletion of a member.

Average Communication Cost:

$$\frac{1}{2}\{n_1 p_1 (2h_1+1)K + n_1(n_2-1)p_2(2h_2+1)K\} + \frac{1}{2}\{n_1 p_1 (d-1)h_1 K + n_1(n_2-1)p_2(d-1)h_2 K\}$$

The first and the second terms are the communication cost for adding a new GSA or member to the tree respectively. The third and fourth terms are the communication cost for deleting a GSA or member from the tree. All four terms derive straightforward from the cost analysis of CBT. Again, GSAs and members are viewed by the GSC as two kinds of members.

Second Scheme Architecture: GSC to GSAs Key Tree, Each Cluster GKMP

The tree is balanced with $h_1 = \log_d n_1$,

Storage Cost:

GSC: $\left\lceil \frac{(dn_1-1)}{d-1} K \right\rceil$ GSA: $\left\lceil 2K + n_2 K + (h_1+1) K \right\rceil$ Members: $\lceil 3K \rceil$

The GSC stores the secret keys of all the nodes of the tree. The GSA holds the path of the keys up to the root (GSC) in its tree. It needs additionally store the secret key with which it communicates with the GSC individually as we have seen in the analysis of CBT costs, the n_2 secret keys it communicates to each member individually using the symmetric encryption method and the session key it needs to communicate with all the members of its cluster.

Members store their private keys with which they communicate with the GSC or GSA in order to get their session key using the symmetric encryption method. Apart from that, the members store the session key that they share with the GSA, and the session key they share with the GSC.

Observation: At this point, we have to make another assumption as well: where are the public keys of GSAs or members published, and how do the GSC, GSAs and members access them? Along with this issue of course, goes the issue of entity authentication. Usually, when two entities A and B with to securely communicate there is a third trusted entity (we can say that it is the certification authority (CA)) that has issued the public and private keys to every entity. Thus, for every entity a designated pair of RSA public and private keys is assigned by the CA. The public key of this entity is published along with the member's identification. This mechanism in general ensures that entity A knows the correct public key for entity B and is certain that member B only can decrypt with its private key the encrypted message sent by A intended for B. Thus, entity authentications ensures that entity A communicates with legitimate entities only that have previously contacted the CA, and more specifically with the entities it actually intends to communicate. However, this mechanism does not protect against compromised entities. Entity Authentication in MANET environment where sometimes no trusted third members can be assumed, or even if such members can be assumed, they might not be accessible at all times due to the mobility of nodes or due to bandwidth constraints is not so easy. In this work, we assume that there is a mechanism that ensures that all members that participate in key distribution are already authenticated. We are going to analyze Entity Authentication for MANETs in a future work. In the current framework we are going to assume that the GSC acquires the capabilities required for playing the role of a CA. We assume that all public keys of entities in the network are published at the GSC. We have further assumed that the GSAs in the general case have access to the GSC at any time (availability of the GSC for the GSAs is assumed in terms of services and in terms of bandwidth for the up-link and the down-link). This means that the

GSAs can communicate with the GSCs and ensure that a particular member is authenticated and get its authentic public key as well. Furthermore, we assume that the GSC monitors the network for new or evicted (from the whole network) members, authenticates the new members, assigns public keys to members that do not have, and at regular intervals refreshes the certificates members carry with them (again this issue is not addressed here). Members might not have immediate access to the GSC through the up-link, however GSC is assumed to access all members in the general case through downlink communication. This is why members communicate with their local GSAs. It is assumed that all members that belong to a GSA are connected with it. If a member looses connection with its GSA, it is considered evicted from the particular GSA and either it joins another existing one, joins a new one, or is totally evicted from the network.
During the formulation of a new group, the leading GSA is going to request from the GSC the public keys of the joining members. It needs to store these keys only until it distributes to the members its own secret keys. Afterwards it does not need to store or request them anymore. Furthermore, the GSA will request from the GSC the public keys of the members that are added to an existing group. The GSC automatically communicates to the proper GSA the information it requires every time such an event occurs. We assume that for the communication of public keys and authentication certificates no further bandwidth is required. Their communication might take place along with signaling messages exchanged among GSC and GSAs at regular intervals, or "HELLO" messages of the routing protocol, or from the MAC etc. Again, we are not addressing this issue in the current work.

GSC Computational Cost: $\left\lceil \frac{(dn_1-1)}{d-1} C_r + \frac{d(n_1-1)}{d-1} C_{SE} + n_1 C_{PE} \right\rceil$

GSA Computational Cost: $\left\lceil C_{PD} + h_1 C_{SD} + (n_2+1) C_r + n_2 (C_{PE} + C_{SE}) \right\rceil$

Member Computational Cost: $\left\lceil C_{PD} + C_{SD} \right\rceil$

Communication Cost: $\left\lceil \left(n_1 + \frac{d(n_1-1)}{d-1} \right) K + 2 n_1 n_2 K \right\rceil$

The GSC creates (dn$_1$-1)/(d-1) new keys with the function generator Cr. The GSC is responsible only for the distribution of keys to GSAs, and the term for the computational cost is derived straight from the CBT cost formulas.

The GSA decrypts the secret key sent to it with the public encryption method and then decrypts the keys associated with its path up to the root sent using the symmetric encryption method. The GSA creates the secret keys of all its members as well (n2), encrypts them and distributes them to the members according to the cost formulas of the GKMP protocol.

The member just decrypts the key sent to it via the public encryption method and the key sent to it via symmetric encryption method.

The communication cost is the summation of the number of keys sent to GSAs from the GSC and to members from their leader GSAs respectively.

Average Operating Cost for GSC:

$$\frac{1}{2}\{n_1 p_1 ((h_1+1)C_r + C_{PE} + 2h_1 C_{SE})\} + \frac{1}{2}\{n_1 p_1 (h_1 C_r + (d-1)h_1 C_{SE})\}$$

The first term is the cost for adding a new GSA to the tree of GSAs. In this scheme the GSC is not involved with the addition and deletion of members. These operations are up to the GSA to handle. The second term is the cost of deleting a GSA from the tree.

Average Operating Cost for GSA:

$$\frac{1}{2}\{n_1 p_1 (C_{PD} + h_1 C_{SD}) + n_1 (n_2 - 1) p_2 (2C_r + C_{PE} + 2C_{SE})\} +$$

$$\frac{1}{2}\{n_1 p_1 ((n_2 + 1)C_r + n_2 C_{PE} + n_2 C_{SE}) + n_1 (n_2 - 1) p_2 (C_r + (n_2 - 1) C_{SE})\}$$

The first and the second term represent the cost for adding a new GSA or a new member respectively, according to the cost formulas of GKMP and CBT protocols. The third and fourth term represent the cost for deleting a GSA or a member respectively.
When a GSA is to be deleted (third term) the GSC is responsible for removing it and for sending the appropriate update keys to the appropriate nodes in the tree, but another GSA must recruit the members of the evicted GSA. So, either there is a free GSA to which no cluster has been assigned yet, or a member of the cluster is randomly selected as GSA. The new GSA recruits all n_2 members and builds a new cluster by creating and distributing the appropriate keys according to GKMP.

Average Operating Cost for Members:

$$\frac{1}{2}\{n_1 (n_2 - 1) p_2 (C_{PD} + C_{SD})\} + \frac{1}{2}\{n_1 p_1 (C_{PD} + C_{SD}) + n_1 (n_2 - 1) p_2 C_{SD}\}$$

The addition term represents the cost for adding a member to the cluster (according to the GKMP formulas) and the deletion term the cost for deleting a GSA from the tree or evicting a member from the cluster. When a GSA is evicted then its members join in another cluster. The first part of the deletion term that represents the cost for the member joining in another cluster if its GSA is deleted, an event with probability p_1. The last deletion term represents the overhead on the members when a member is deleted from its group.

Average Communication Cost:

$$\frac{1}{2}\{n_1 p_1 (2h_1 + 1)K + n_1 (n_2 - 1) p_2 (3K)\} + \frac{1}{2}\{n_1 p_1 ((d - 1) h_1 K + 2 n_2 K) + n_1 (n_2 - 1) p_2 (n_2 - 1) K\}$$

The average communication cost sums up the keys that are exchanged both in the case of a GSA or member addition and in the case of a GSA or member deletion respectively. The terms of the equation are derived from the cost analysis of CBT and GKMP.

Third Scheme:
GSC to GSAs Key Tree, each Cluster Key Tree

$h_1 = \log_{d1} n_1$ $h_2 = \log_{d2} n_2$, each tree is balanced.
d_1 = fan out of upper tree d_2 = fan out of lower tree

Storage Cost:

GSC: $\left\lceil \frac{(d_1 n_1 - 1)}{d_1 - 1} K \right\rceil$ GSA: $\left\lceil (h_1 + 1)K + \frac{(d_2 n_2 - 1)}{d_2 - 1} K \right\rceil$ Members: $\left\lceil (h_2 + 1) K \right\rceil$

Since both levels of hierarchy are simple trees, all costs are derived from the CBT cost analysis.
Observation: If we decide that we want the public keys of members stored in the GSA, we have to add to the storage cost of GSA the number n_2 of these public keys of members.

The GSA needs to store all keys of the tree it creates with the members of its cluster and also the keys associated with its path up to the root in the GSC tree.

The storage cost for each member is its own individual secret key and the keys of the nodes that belong to its path up to the root of the tree.

Computational Cost:

$$\text{GSC: } \left[\frac{(d_1 n_1 - 1)}{d_1 - 1} C_r + \frac{d_1 (n_1 - 1)}{d_1 - 1} C_{SE} + n_1 C_{PE} \right]$$

$$\text{GSA: } \left[C_{PD} + h_1 C_{SD} + \frac{(d_2 n_2 - 1)}{d_2 - 1} C_r + \frac{d_2 (n_2 - 1)}{d_2 - 1} C_{SE} + n_2 C_{PE} \right] \qquad \text{Members: } \left[C_{PD} + h_2 C_{SD} \right]$$

The GSA decrypts the appropriate keys sent to it from the GSA and simultaneously encrypts the appropriate keys it needs to send to its tree of members, as the GSC encrypts the appropriate keys to send to the tree of GSAs according to the CBT scheme.

$$\text{Communication Cost: } \left[\left(n_1 + \frac{d_1 (n_1 - 1)}{d_1 - 1} \right) K + n_1 \left[n_2 + \frac{d_2 (n_2 - 1)}{d_2 - 1} \right] K \right]$$

Again, this cost can be easily derived from the communication cost we calculated for the second scheme, for the part of the communication that involves the tree key management scheme. In this case we have two tree key management schemes, one in the first level of hierarchy (between the GSC and the GSAs) and one in the second level of hierarchy (between the GSA and the members). After this observation it is easy to derive the formula above for the communication cost.

Average Operating Cost for GSC:

$$\frac{1}{2}\{ n_1 p_1 ((h_1 + 1) C_r + C_{PE} + 2 h_1 C_{SE}) \} + \frac{1}{2}\{ n_1 p_1 (h_1 C_r + (d_1 - 1) h_1 C_{SE}) \}$$

In this scheme the GSC interacts with the GSA only as in the previous scheme. The first term is the operating cost of GSC for adding a GSA and the second and third term are the operating costs for deleting a GSA from the tree. It has been shown in CBT how these terms are derived.

Average Operating Cost for GSA:

$$\frac{1}{2}\{ n_1 p_1 (C_{PD} + h_1 C_{SD}) + n_1 (n_2 - 1) p_2 ((h_2 + 1) C_r + C_{PE} + 2 h_2 C_{SE}) \}$$
$$\frac{1}{2}\{ n_1 p_1 (\frac{d_2 n_2 - 1}{d_2 - 1} C_r + n_2 C_{PE} + \frac{d_2 (n_2 - 1)}{d_2 - 1} C_{SE}) + n_1 (n_2 - 1) p_2 (h_2 C_r + (d_2 - 1) h_2 C_{SE}) \}$$

The first term of the communication for addition case, represents the addition of a new GSA and the second term the addition of a member. The third and fourth terms represent the deletion of a GSA and of a member respectively. When a GSA is evicted then its members join in another cluster. The first part of the deletion term that represents the cost for the member joining in another cluster if its GSA is deleted, an event with probability p_1.

Average Operating Cost for Member:

$$\frac{1}{2}\{n_1(n_2-1)p_2(C_{PD}+h_2C_{SD})\}+\frac{1}{2}\{n_1p_1(C_{PD}+h_2C_{SD})+n_1(n_2-1)p_2h_2C_{SD})\}$$

The first term is the cost for addition of a member to a cluster, the second and third terms are the cost assigned on the members when their GSA is deleted and the cost for a member eviction respectively.

Average Communication Cost:

$$\frac{1}{2}\{n_1p_1(2h_1+1)K+n_1(n_2-1)p_2(2h_2+1)K\}+\frac{1}{2}\{n_1p_1((d_1-1)h_1K+n_2K+d_2\frac{n_2-1}{d-1}K)+n_1(n_2-1)p_2(d_2-1)h_2K\}$$

The average communication cost in this case derives from the addition of a GSA or a member and also from the deletion of a GSA or a member. We just assume that the two levels of hierarchy do not interfere with each other.

Fourth Scheme/ Architecture: Single Key Tree, single Group of Members:

p: same for all members.
Now, h= - log p, p=1/(n_1 n_2) balanced tree:

Storage Cost:

GSC: $\left\lceil \frac{d^{h+1}-1}{d-1} \right\rceil$ GSA: $\left\lceil (h+1)K \right\rceil$ Members: $\left\lceil (h+1)K \right\rceil$

The GSC stores all leaf nodes of the tree. They are associated with both GSAs and members. GSAs and members need only store the nodes of the tree that belong to their path from the leaf to the root.

Computational Cost:

GSC: $\left\lceil \frac{d^{h+1}}{d-1}C_r+(n_1+n_1n_2)C_{PE}+\left(\frac{d^{h+1}}{d-1}-1\right)C_{SE} \right\rceil$

GSA: $\left\lceil G_{PD}+hC_{SD} \right\rceil$ Members: $\left\lceil G_{PD}+hC_{SD} \right\rceil$

Communication Cost: $\left\lceil \left\{\frac{d^{h+1}-1}{d-1}+n_1(n_2+1)-1\right\}K \right\rceil$

All cost functions derive from the cost functions for the CBT protocol.

Average Operating Cost for GSC:

$$\frac{1}{2}\{n_1 p((h+1)C_r + C_{PE} + 2hC_{SE}) + n_1(n_2-1)p((h+1)C_r + C_{PE} + 2hC_{SE})\} +$$

$$\frac{1}{2}\{n_1 p(hC_r + (d-1)hC_{SE}) + n_1(n_2-1)p(hC_r + (d-1)hC_{SE})\}$$

The first and the second terms are the costs for the addition of a GSA or of a member respectively, and the third and fourth terms are the costs for the deletion of a GSA or of a member respectively. The GSC in this scheme views GSAs and members exactly the same, and places both at the leaves of a tree where all paths acquire the same height h. They represent the bottom level of a single hierarchy, they both reside on the leaves of the tree. The costs of the formulas are derived from CBT.

Average Operating Cost for GSA: $n_1 p(C_{PD}+hC_{SD})$

Since the GSAs are placed in the leaves of the trees with no hierarchy under them, they behave exactly like simple members. They perform decryption of their private keys. The operations for addition and deletion are left on the GSC.

Average Operating Cost for Member: $n_1(n_2-1)p(C_{PD}+hC_{SD})$

Average Communication Cost:

$$\frac{1}{2}\{n_1 p(2h+1)K + n_1(n_2-1)p(2h+1)K\} + \frac{1}{2}\{n_1 p(d-1)hK + n_1(n_2-1)p(d-1)hK\}$$

Fifth Scheme/ Architecture:
GSC to GSAs GKMP, each cluster Key Tree

$h_2 = \log_d n_2$ (balanced tree) $\qquad n_1 p_1 + n_1 (n_2 - 1) p_2 = 1$

$\lceil 3K + K(\frac{d_2^{n_2}-1}{d_2-1}) \rceil$ Storage Cost:

GSC: $\lceil (n_1+1)K \rceil$ GSA: $\lceil 3K + K(\frac{d_2^{n_2}-1}{d_2-1}) \rceil$ Members: $\lceil (h+1)K \rceil$

The members store the private keys of nodes that belong to their path in the tree up to the root, and also each member has the option to store a private key sent to it directly from the GSC. This burdens the GSC with more storage cost and more computation cost and also the initial communication cost becomes larger. This additional feature might be needed for the downlink communication only, if for any reason the GSC needs to have direct inexpensive communication with the member (e.g. re-authenticate it, provide it with some certificate etc). The mechanism of authentication is going to be described in a future report, and it is out of the scope of the current research for the moment. However, in that case the modified costs involving this feature are the following:

GSC Storage Cost: $\left\lceil (n_1+1)K+n_1(n_2-1)K \right\rceil$

GSC Computation Cost: $\left\lceil (n_1+1)C_r+n_1(C_{PE}+C_{SE})+n_1(n_2-1)C_r+n_1(n_2-1)C_{PE} \right\rceil$

Member Computational Cost: $\left\lceil C_{PD}+hC_{SD}+C_{PD} \right\rceil$

Communication Cost: $\left\lceil 2n_1K+(n_1n_2+n_1\frac{d(n_2-1)}{d-1})K+n_1(n_2-1)K \right\rceil$

We can observe that these costs are much higher now. If we decide to adopt this scheme we must have a very good reason for doing so. Allowing the GSC to communicate frequently with the members destroys the hierarchy of the scheme and it does not work as it was intended to anyway. The most important advantage of this scheme is that membership changes can be dealt with in a localized manner by the GSAs. And this is particularly useful in MANETs when high mobility restricts the communication among distant nodes, and hop by hop communication is greatly encouraged also, since nodes might acquire limited bandwidth, energy and resources. As for the GSC, it is generally assumed to be a powerful entity such as a satellite, but even so, because of the delays encountered in satellite systems, it is better that the members communicated more frequently with the GSA allocated to them, than with the satellite. For all these reasons, we still insist that the hierarchical scheme used as described until now, is the most efficient scheme from every point of view.

In what follows we continue our analysis according to all the previous assumptions:

Computational Cost

GSC: $\left\lceil (n_1+1)C_r+n_1(C_{PE}+C_{SE}) \right\rceil$

GSA: $\left\lceil (C_{PD}+C_{SD})+\frac{(dn_2-1)}{d-1}C_r+n_2C_{PE}+\frac{d(n_2-1)}{d-1}C_{SE} \right\rceil$ Members: $\left\lceil C_{PD}+hC_{SD} \right\rceil$

The GSA decrypts the secret key and the session key sent to it by the GSA, and computes and encrypts the appropriate keys for the members of its tree. All costs derive from the cost analysis for GKMP and CBT.

Communication Cost: $\left\lceil 2n_1K+(n_1n_2+n_1\frac{d(n_2-1)}{d-1})K+n_1(n_2-1)K \right\rceil$

The communication cost is the number of the keys sent from GSC to GSAs and the keys sent from each GSA to the members of their tree group.

Average Operating Cost for GSC: $\left\lceil \frac{1}{2}\{n_1p_1(2C_r+C_{PE}+2C_{SE})\}+\frac{1}{2}\{n_1p_1(C_r+(n_1-1)C_{SE})\} \right\rceil$

The addition term represents the cost for adding a new GSA to the tree of GSC. The deletion term is the cost of deleting a GSA from the tree. The costs are derived from the analysis of GKMP and CBT.

Average Operating Cost for GSA:

$$\frac{1}{2}\{n_1p_1(C_{PD}+C_{SD})+n_1(n_2-1)p_2((h+1)C_r+C_{PE}+2hC_{SE})\}+$$
$$\frac{1}{2}\{n_1p_1(\frac{(dn_2-1)}{d-1}C_r+n_2C_{PE}+\frac{d(n_2-1)}{d-1}C_{SE})+n_1(n_2-1)p_2(hC_r+(d-1)hC_{SE})\}$$

The first and second terms are the costs for adding a GSA or a member to the scheme. The third and the fourth terms represent the cost for deleting a GSA or a member. The members that were left without GSA need to find a new leader, so the third term is the cost for connecting the new GSA to these members under a tree scheme.

Average Operating Cost for Member:

$$\frac{1}{2}\{n_1(n_2-1)p_2(C_{PD}+h_2C_{SD})\}+\frac{1}{2}\{n_1p_1(C_{PD}+h_2C_{SD})+n_1(n_2-1)p_2h_2 2C_{SD})\}$$

The first and second terms are the costs for adding a GSA or a member to the scheme. The third and the fourth terms represent the cost for deleting a GSA or a member. The third term represents the cost required for a member with evicted GSA to join another group, and get the appropriate keys from another GSA.

Average Communication Cost:

$$\frac{1}{2}\{n_1p_1 3K+n_1(n_2-1)p_2(2h_2+1)K\}+\frac{1}{2}\{(n_1p_1(n_1-1)K+n_2K+d\frac{n_2-1}{d-1}K)+n_1(n_2-1)p_2(d-1)h_2K\}$$

The first term is the communication cost for adding a new GSA into the GKMP scheme. The second term is the cost for adding a member into the tree scheme of a GSA. The fourth term is the cost for deleting a member from the tree of the GSA. The third term is the cost of deleting a GSA from the GKMP scheme and also establishing an initial communication between a new GSA and the members of the deleted GSA.

Sixth Scheme/Architecture: GSC to GSAs GKMP, each cluster GKMP

Storage Cost:

GSC: $\lceil (n_1+1)K \rceil$ GSA: $\lceil 3K+n_2K \rceil$ Members: $\lceil 2K \rceil$

In this scheme, the GSA needs to store the private key that it uses to communicate individually with the GSC and the session key of the GSC group, the session key with the members group and the private keys it distributed initially to the members.

Again there is the possibility for the members to store a third key coming directly from the GSC. This is useful for the same reasons that we have stated earlier. If we decide to exploit this feature then the members store 3K bits. We also need to modify the GSC Storage Cost, the GSC computational Cost, the Member Computational Cost and the total Communication Cost as follows:

GSC Storage Cost: $\lceil n_1K+n_1n_2K \rceil$

GSC Computational Cost: $\lceil (n_1+1)C_r+n_1(C_{PE}+C_{SE})+n_1n_2C_r+n_1n_2C_{PE} \rceil$

Member Computational Cost: $\lceil 2C_{PD}+2C_{SD} \rceil$

Communication Cost: $\lceil 2n_1K+2n_1n_2K+n_1n_2K \rceil$

In what follows we proceed according to our initial assumptions.

Computational Cost:

GSC: $\lceil (n_1+1)C_r+n_1(C_{PE}+C_{SE}) \rceil$

GSA: $\lceil C_{PD}+C_{SD}+(n_2+1)C_r+n_2(C_{PE}+C_{SE}) \rceil$

Members: $\lceil C_{PD}+C_{SD} \rceil$

Communication Cost: $\lceil 2n_1K+2n_1n_2K \rceil$

Average Operating Cost for GSC: $\frac{1}{2}\{n_1p_1(2C_r+C_{PE}+2C_{SE})\}+\frac{1}{2}\{n_1p_1(C_r+(n_1-1)C_{SE})\}$

The addition term stands for the cost of adding a new GSA to the group of GSAs. The second term is the cost of deleting a GSA from the tree. These costs are derived from the GKMP analysis.

Average Operating Cost for GSA:

$$\frac{1}{2}\{n_1p_1(C_{PD}+C_{SD})+n_1(n_2-1)p_2(2C_r+C_{PE}+2C_{SE})\}+$$
$$\frac{1}{2}\{n_1p_1((n_2+1)C_r+n_2(C_{PE}+C_{SE}))+n_1(n_2-1)p_2(C_r+(n_2-1)C_{SE})\}$$

The first term represents the cost for the addition of a GSA to the GKMP. The second term is the cost for the addition of a member in the GKMP of its GSA leader. The third term represents the cost for the deletion of a GSA. The members that were left without GSA need to find a new leader, so this is the cost for connecting the new GSA to its members under the GKMP scheme. The fourth term represents the cost for deleting a member for the GKMP group of the GSA.

Average Operating Cost for Member:

$$\frac{1}{2}\{n_1(n_2-1)p_2(C_{PD}+C_{SD})\}+\frac{1}{2}\{n_1p_1(C_{PD}+C_{SD})+n_1(n_2-1)p_2C_{SD}\}$$

Average Communication Cost:

$$\frac{1}{2}\{n_1p_1 3K+n_1(n_2-1)p_2 3K\}+\frac{1}{2}\{(n_1p_1(n_1-1)K+n_1p_1 2n_2K)+n_1(n_2-1)p_2(n_2-1)K\}$$

Other Schemes

It is obvious that we can incorporate many more key distribution protocols to the Hybrid Two-Level Key Management Scheme. We can apply any of the protocols that we analyze in the rest of the report either to the first level (GSC to GSAs) or to the second level (GSA to members). The most important factors that determine which combination of protocols results in the best performance of our Hierarchical Scheme are among all: the individual performance of each of the applied key distribution protocols separately, the number of entities the key distribution protocol is applied to (n_1, n_2), and the frequency of mobility (p_1, p_2) of nodes at any level.

The number of nodes involved is very important for the performance of each protocol. Some protocols are not scalable to a large number of nodes, on the other hand some protocols may behave better than others when the number of nodes is small and worse when this numbers increases.

Key Distribution Protocols are distinguished in two categories: those based on a trusted entity to distribute keys (e.g. GKMP, CBT, OFT) and the contributory, based on key agreement (e.g. GDH.2, Octopus). The Contributory protocols are preferred when no agreed common secrets or trust among parties exist. They imply no single point of failure, but they are not scalable and they are inherently costly. The Non-Contributory protocols are preferred when trusted, robust central controller exist. They are scalable, efficient, but they have the single point of failure problem. However, the structure of our Hierarchical Hybrid Scheme assumes that the GSC is robust and trusted. Thus, for the upper level, centralized schemes are encouraged. For GSAs, trust may not always be assumed, as it is for the case of GSC, or may not even exist at all. We should not here that the GSC monitors all GSAs. Centralized or contributory schemes could be used for the lower level, depending on network assumptions, topology, group size and mobility.

Furthermore, the mobility rates of nodes are another factor that determines the applicability of key distribution protocols to the two-level hierarchical scheme. The most robust protocols allow higher mobility levels in the network without breaking down as often as the rest of protocols do. Usually, the non-contributory protocols are the most robust ones.

Another issue to consider is how the overall performance of the hybrid scheme is affected when two or more protocols overlap. For example, when two different key distribution protocols are applied to the two levels, the computation and communication cost of the GSA derives from the overlap of those two different protocols. So, in order to minimize the overall computation and communication cost we have to take into account not only the issue of how the protocols operate individually but also of how they interact when combined (e.g. for the calculations of GSA). One protocol may present high computation cost for GSC and low for members, whereas another protocol may present low computation cost for GSC and high for members. If we combine these two protocols so as the second one to be applied to the first level and the first one to the second level we end up with higher cost for the GSA, in comparison to what we would end up with for the GSA if we combined them the opposite way. However, the second combination would result in higher cost for the GSC after all. As for the total communication cost (derives from the GSC, the GSAs and the members) we would have to conduct a performance evaluation to be able to decide on the best combination given the parameter values.

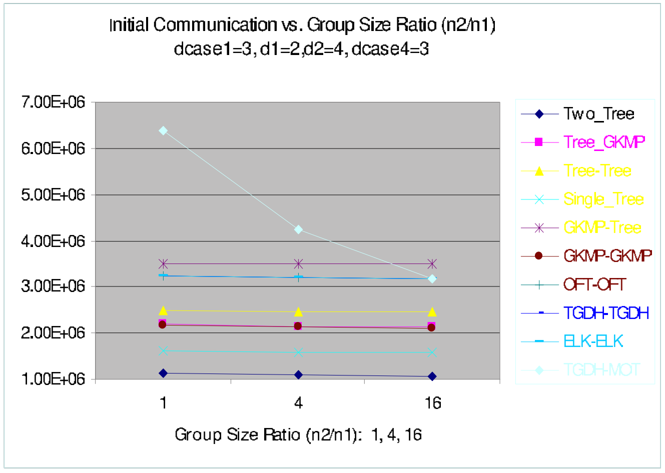
Taking all these issues into account and making most of the logical combinations of the protocols that have been examined we will come up with the combinations of protocols that are best for the two - level hybrid scheme. The most efficient protocols are OFT, TGDH, CBT and MOT. The existence of the GSC ensures that centralized schemes can be used for this model.

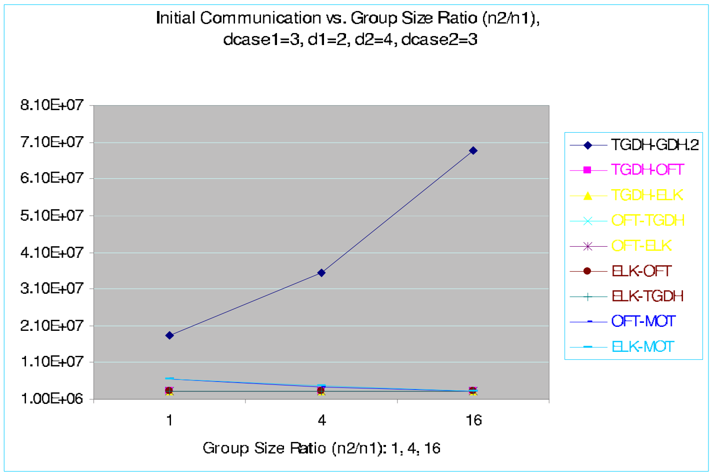
We have evaluated the following schemes for our comparison:

1. Single Key Tree, two Member Groups
2. GSC to GSAs Key Tree, clusters GKMP
3. GSC to GSAs Key Tree, clusters Key Tree
4. Single Key Tree, single Members Group
5. GSC to GSAs GKMP, clusters Key Tree
6. GSC to GSAs GKMP, clusters GKMP
7. GSC to GSAs OFT, each cluster {OFT, Key Tree, GDH.2, 2d-Octopus, TGDH}
8. GSC to GSAs TGDH, each cluster {OFT, Key Tree, GDH.2, 2^d -Octopus, TGDH}

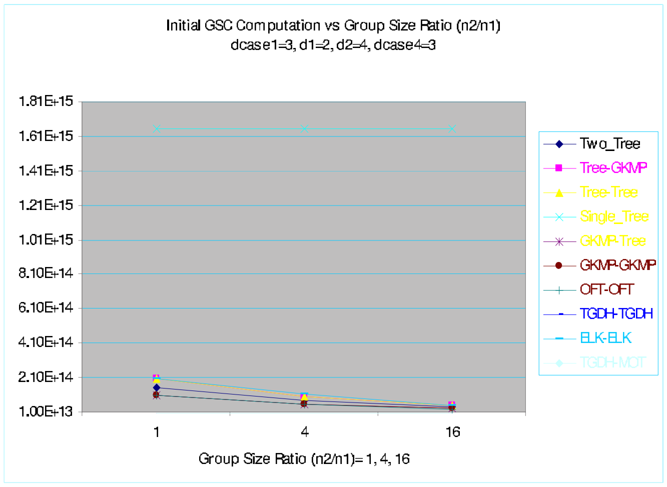
Graphic Results of Our Comparison

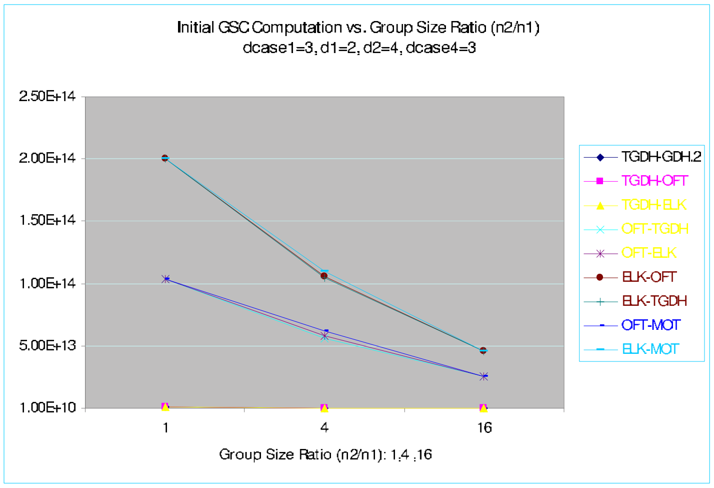
Initial Costs



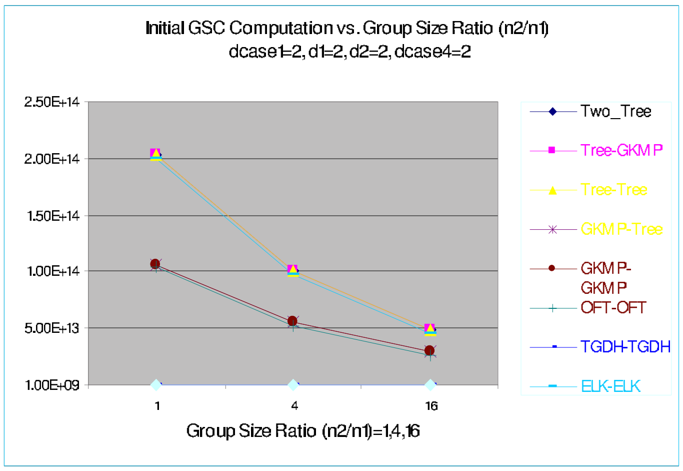


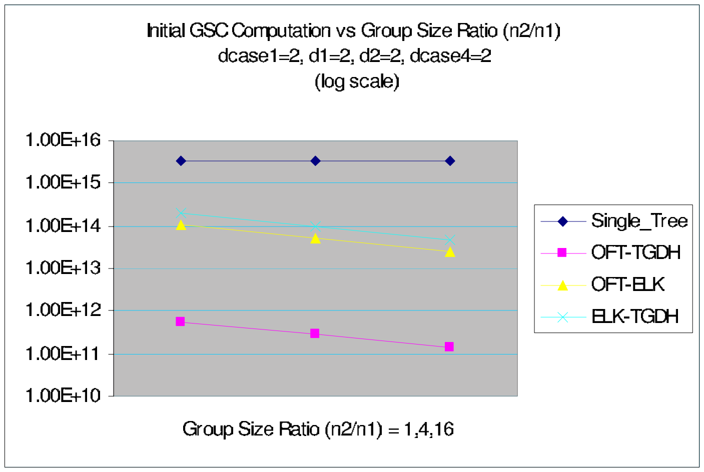
Figures 1, 2: Initial Comm/tion cost vs. Group Size Ratio (n2/n1): SinlgeTree, TwoTree and ELK-TGDH reduce Initial Communication the most. Not very sensitive to Group Size Ratio but for TGDH-MOT and TGDH-GDH.2 (presents the worst performance)



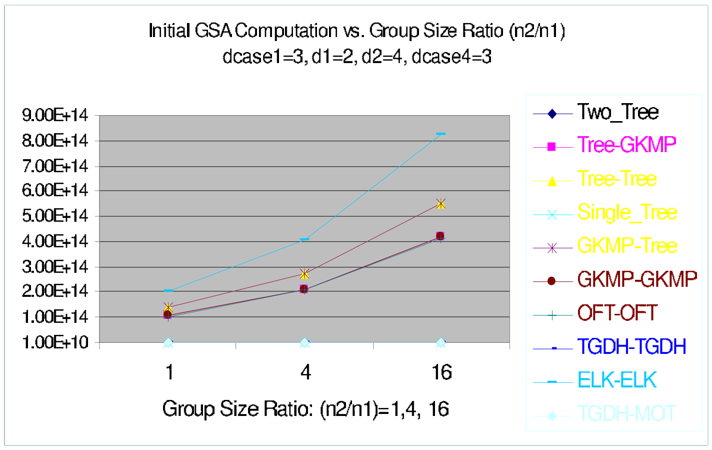


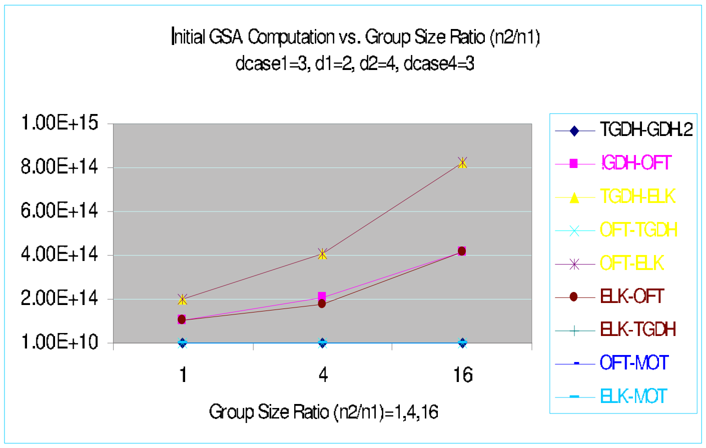
Figures 3, 4: Initial GSC Computation cost vs. Group Size Ratio (n2/n1): TGDH-ELK, TGDH-OFT followed by OFT-OFT, ELK-ELK, GKMP-GKMP reduce Initial Comm/tion the most. Single-Tree presents the worst performance, not sensitive to Group Size Ratio



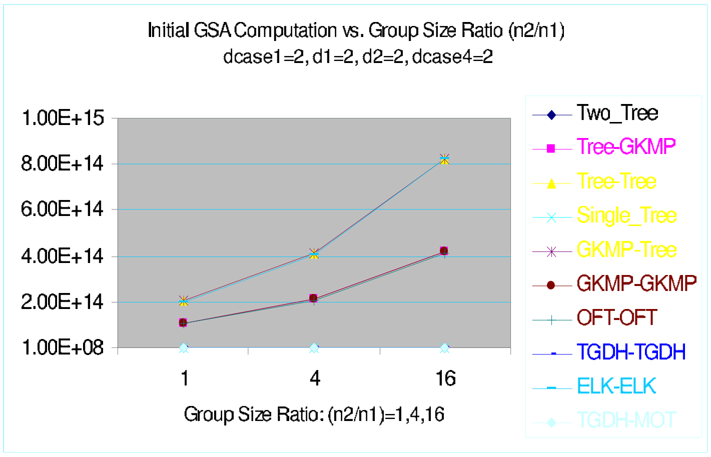


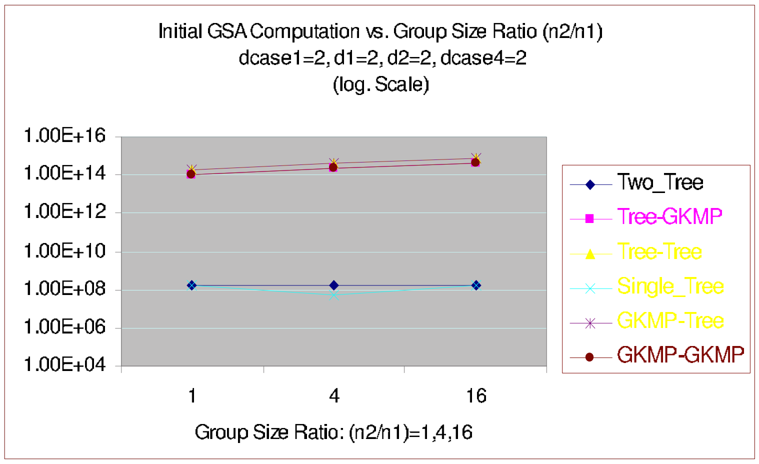
Figures 5, 6: Initial GSC Computation cost vs. Group Size Ratio (n2/n1) for different descendants in the trees: TGDH-ELK, TGDH-OFT followed by OFT-OFT, ELK-ELK, GKMP-GKMP reduce Initial Communication the most. Single-Tree presents the worst performance and is not sensitive to Group Size Ratio



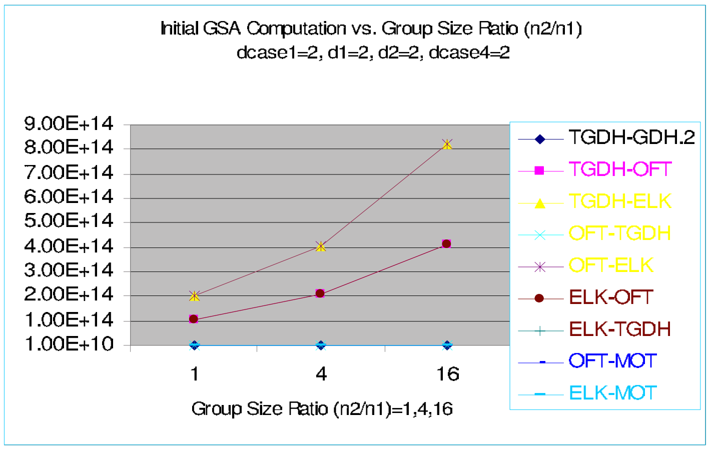


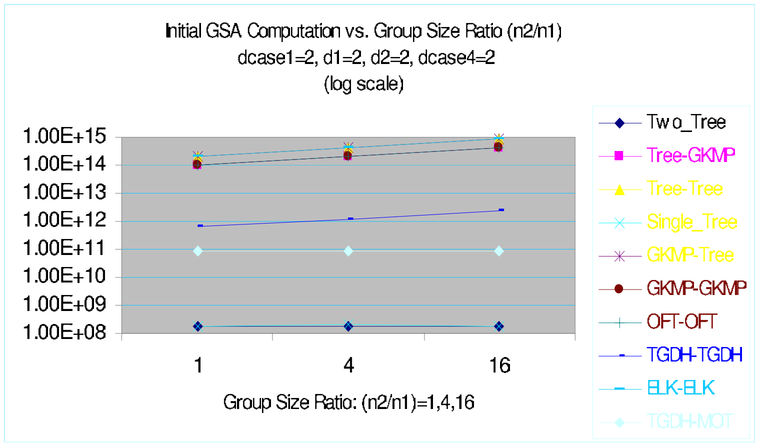
Figures 7, 8: Initial GSA Computation cost vs. Group Size Ratio (n2/n1): TGDH-MOT, OFT-MOT, ELK-MOT followed by GKMP-GKMP, OFT-OFT reduce Initial GSC Computation the most. Combinations with second component ELK present the worst performance.





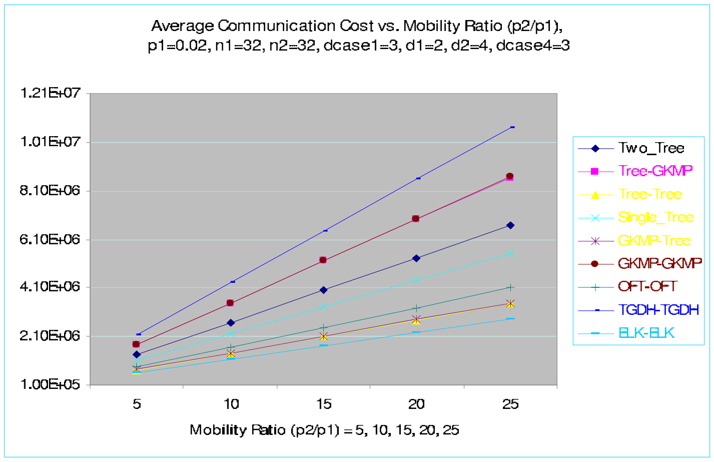
Figures 9, 10: Initial GSA Computation cost vs. Group Size Ratio (n2/n1) for different descendants in the trees:TGDH-MOT, OFT-MOT, ELK-MOT followed by GKMP-GKMP, OFT-OFT reduce Initial GSC Computation the most. Combinations with second component ELK present the worst performance. Costs sensitive to n2/n1

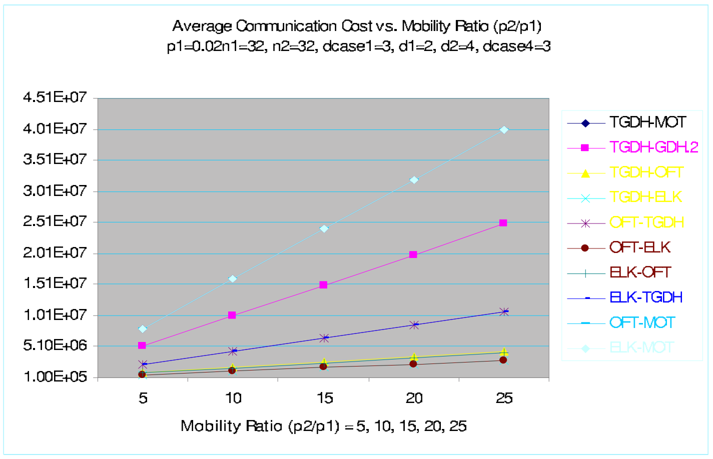




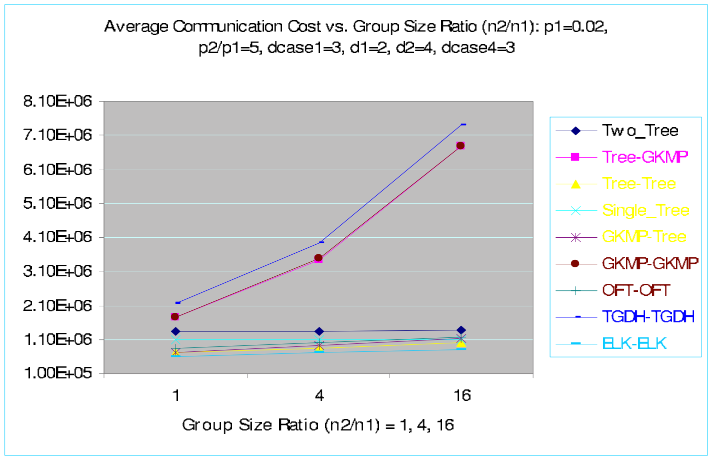
Figures 11, 12: Initial GSA Computation cost vs. Group Size Ratio (n2/n1) for different descendants in the trees: TGDH-MOT, OFT-MOT, ELK-MOT, TwoTree, SingleTree, followed by GKMP-GKMP, OFT-OFT reduce Initial GSC Computation the most. Combinations with ELK as 2nd scheme present worst performance.

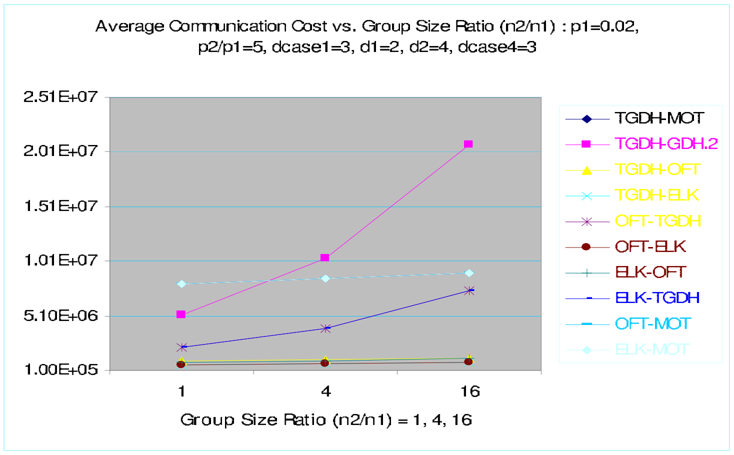
Operating Costs



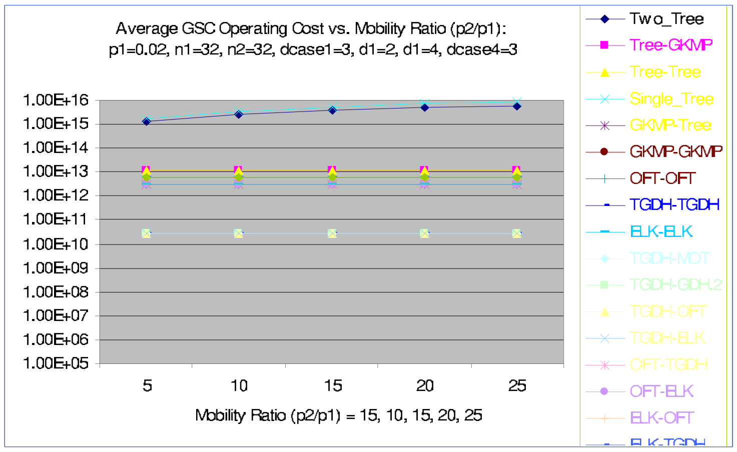


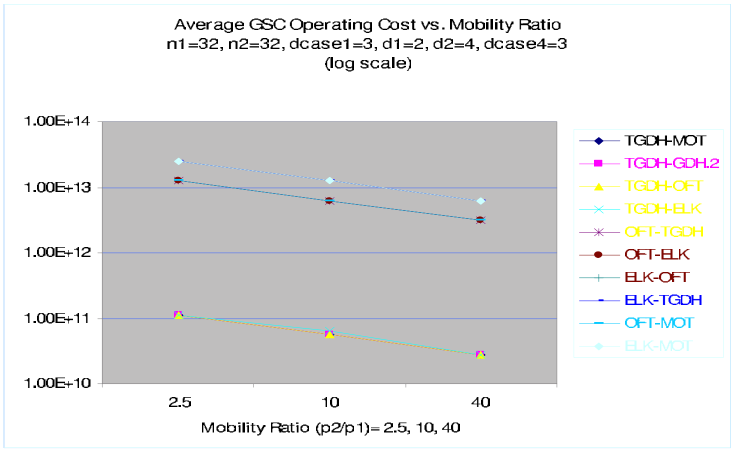
Figures 13, 14: Average Comm/tion cost vs. Mobility Ratio (p2/p1):combinations with second component OFT and ELK followed by GKMP-GKMP, Tree-GKMP reduce Average Comm/tion the most. Combinations with TGDH, Tree-GKMP, Tree-Tree present the worst performance.Costs sensitive to (p2/p1)



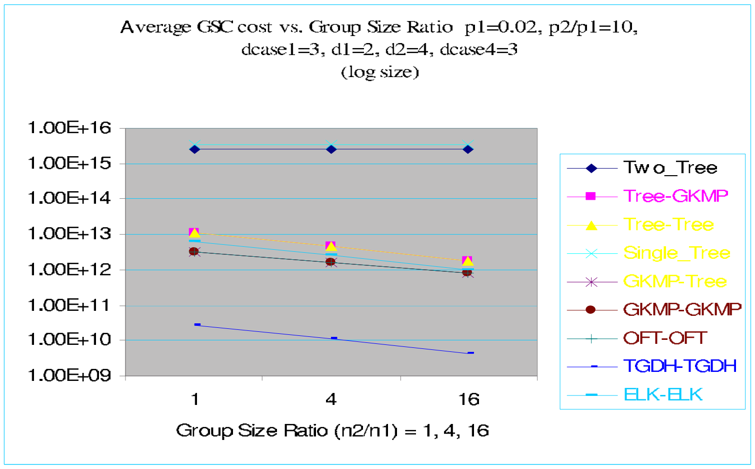


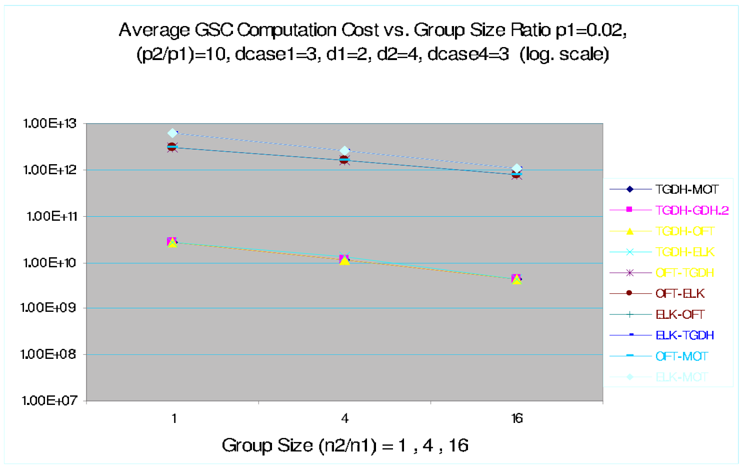
Figures 15, 16: Average Comm/tion cost vs. Group Size Ratio (n2/n1): combinations with second component OFT and ELK followed by GKMP-GKMP, Tree-GKMP reduce Average Comm/tion the most. Combinations with TGDH, Tree-GKMP, Tree-Tree present the worst performance. Costs sensitive to (n2/n1)



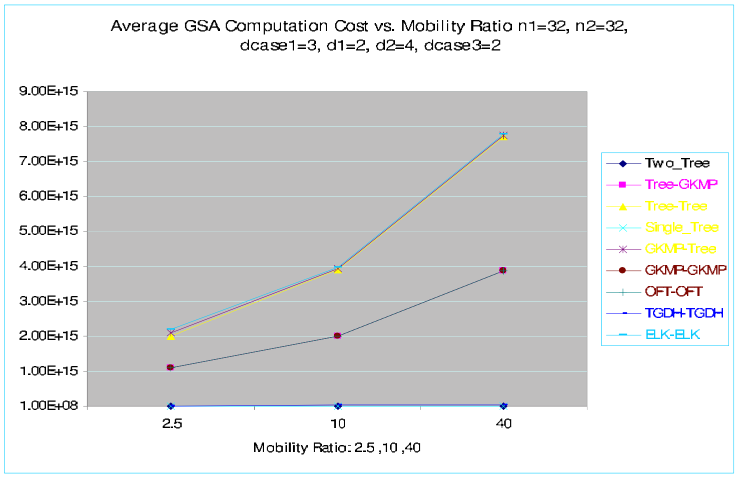


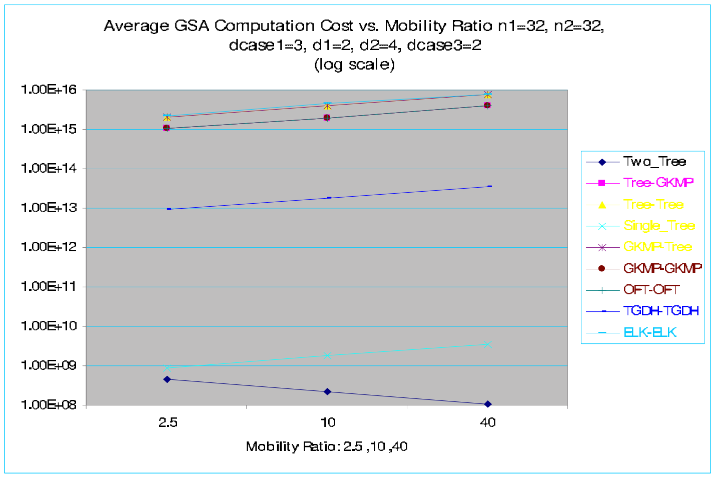
Figures 17, 18: Average GSC Operating cost vs. Mobility Ratio (p2/p1):combinations of TGDH followed by OFT-OFT, ELK-ELK, GKMP-GKMP, Tree-GKMP reduce Average GSC Operating cost the most. Single-Tree and TwoTree present the worst performance. All cost sensitive to the mobility ratio.



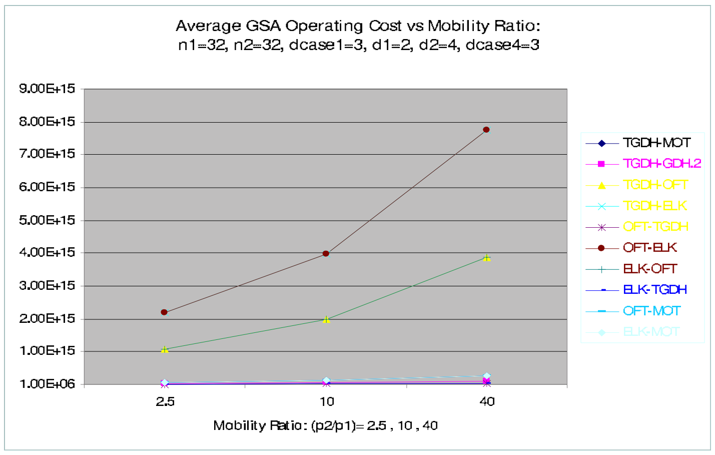


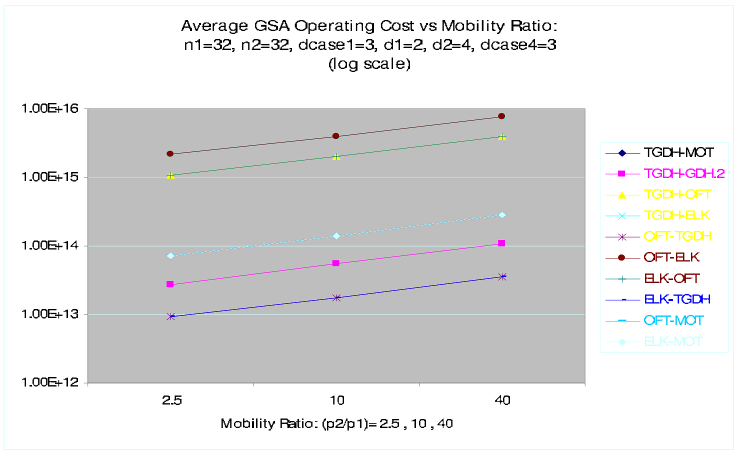
Figures 19, 20: Average GSC Operating cost vs.Group Size Ratio (n2/n1): Combinations of TGDH followed by OFT-OFT, ELK-ELK, GKMP-GKMP, Tree-GKMP reduce Average GSC Operating cost the most. Single-Tree and TwoTree present the worst performance. Most costs sensitive to Group Size Ratio



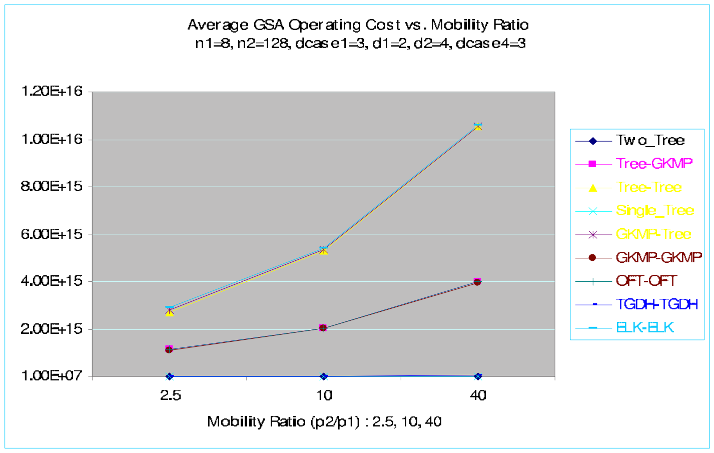


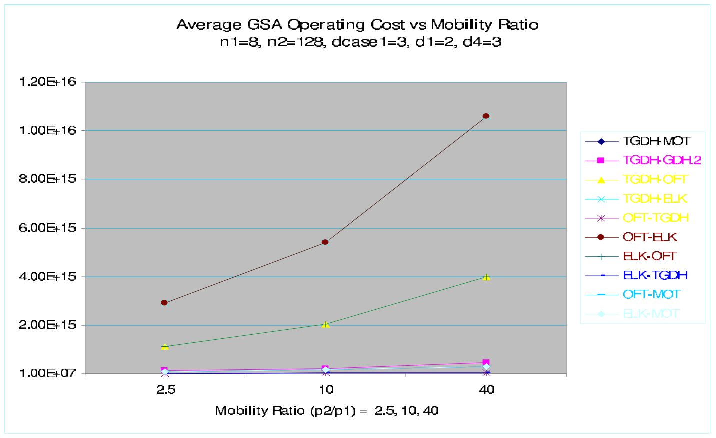
Figures 21, 22: Average GSA Operating cost vs. Mobility Ratio (p2/p1) for Group Sizes (n1, n2)=(32, 32): SingleTree, TwoTree, followed by combinations of TGDH reduce Average GSA Operating cost the most. Combinations of ELK present the worst performance. All costs sensitive to Mobility Ratio



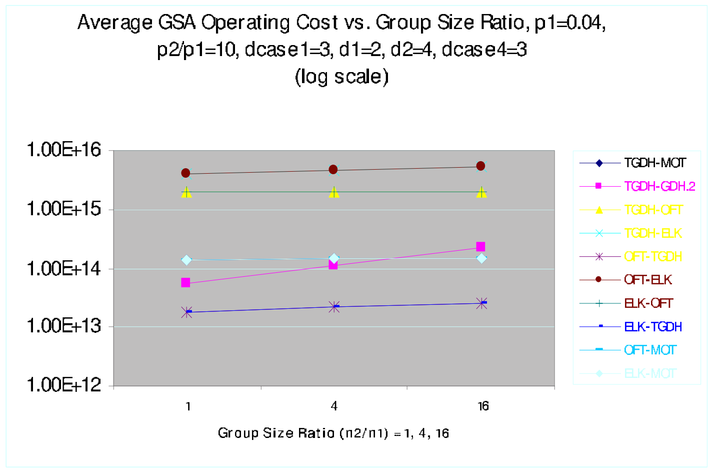


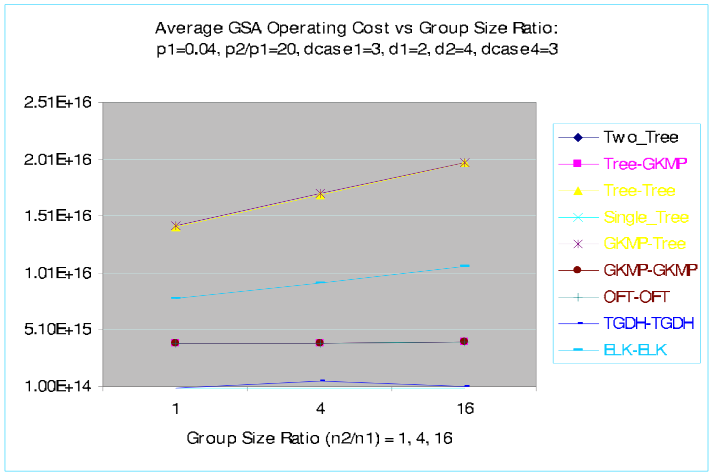
Figures 23, 24: Average GSA Operating cost vs. Mobility Ratio (p2/p1) for Group Sizes (n1, n2) = (32, 32): SingleTree, TwoTree, followed by combinations of TGDH reduce Average GSA Operating cost the most. Combinations of ELK present the worst performance. All costs sensitive to Mobility Ratio



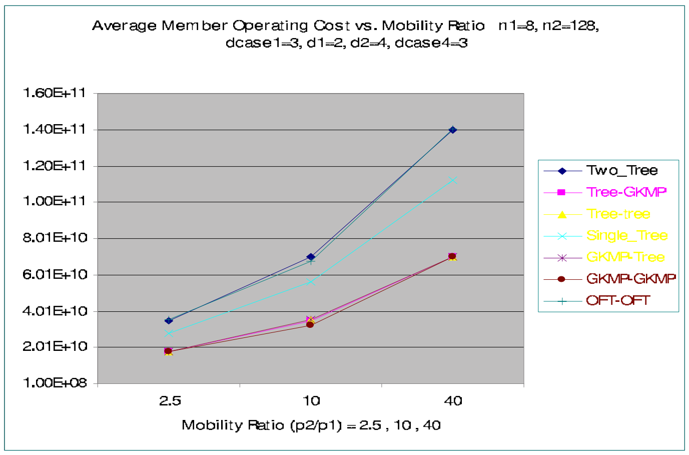


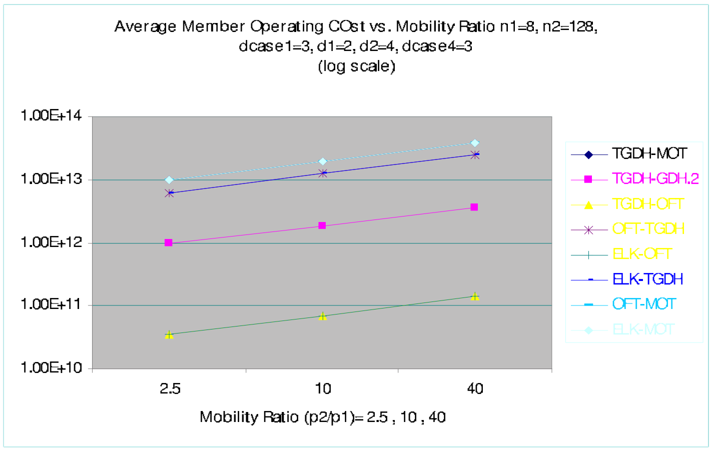
Figures 25, 26: Average GSA Operating cost vs. Mobility Ratio (p2/p1) for different group sizes (n1, n2)= (8, 128): SingleTree, TwoTree, followed by combinations of TGDH reduce Average GSA Operating cost the most. Combinations of ELK present the worst performance. All costs sensitive to Mobility Ratio





Figures 27, 28: Average GSA Operating cost vs. Group Size Ratio (n2/n1) for different mobilities p1=0.04, p2=0.4: SingleTree, TwoTree, followed by combinations of TGDH reduce Average GSA Operating cost the most. GKMP-Tree and Tree-Tree, and combinations that have OFT or ELK as second component present the worst performance. All costs sensitive to Group Size Ratio





Figures 29, 30: Average Member Operating cost vs. Mobility Ratio (p2/p1) for Group Sizes (n1, n2)=(8, 128): Tree-GKMP, GKMP-GKMP, followed by combinations with OFT as second component reduce Average Member Operating cost the most. Combinations that have ELK as the second component present terrible performance. All costs sensitive to Mobility Ratio

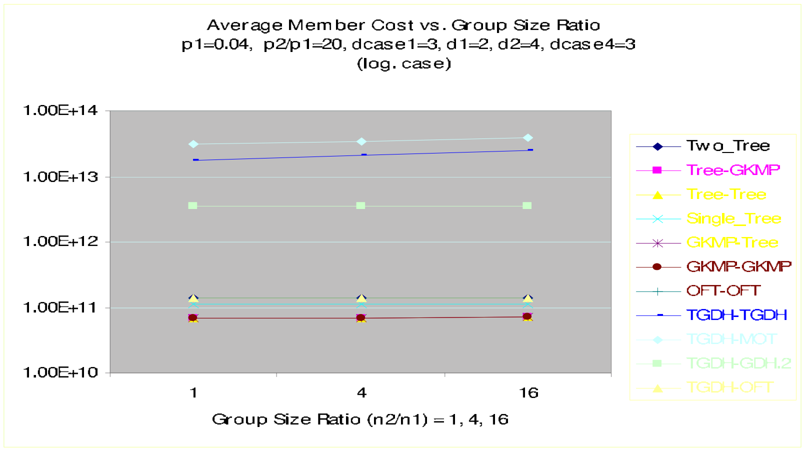


Figure 31: Average Member Operating cost vs. Group Size Ratio (n2/n1) for probability ratio p2/p1=20: Tree-GKMP, GKMP-GKMP, followed by combinations with OFT as second component reduce Average Member Operating cost the most. Combinations that have ELK as the second component present terrible performance. All costs sensitive to Mobility Ratio

Observations - Discussion

The Scheme presents *Adaptability* to network limitations and topology. It may use a combination of protocols to achieve either substantial communication overhead reduction, or lower the computation cost or both. It may support a wide range of protocols at any time. Given the network parameters, such as *users mobility frequencies*, user characteristics (processing capabilities, bandwidth, battery life), *overall number of users* and *number of users of certain categories*, it can select the *most appropriate* key distribution protocols for both its levels and combine them to a *general very efficient hybrid scheme*.

For schemes 7-19 we did not use the Tree protocol anymore, because the performance of OFT is better or equivalent in every aspect.

For our evaluation we assume:
$n_2/n_1 = t$: (1, 4, 16) and n_1=(32, 16, 8), n_2 =(32, 64, 128), $n_1 * n_2 = 1024$
$p_2/p_1 = y$: (5, 10, 15, 20, 25) or {(2.5, 10, 40) and p_1 = (0.08, 0.04, 0.02), p_2 = (0.2, 0.4, 0.8)}

Operating Costs *are very sensitive* to t and y, all costs are sensitive to values *K* and consequently to all C_{PE}/C_{PD} , C_{SE}/C_{SD} , C_r and C_{rr} costs. Naturally, metrics *are sensitive* to p_1, p_2,,n_1, n_2 alone.

Certain metrics are *sensitive* to the choices for number of descendants: d_1, d_2, $dcase_1$, $dcase_4$ of the trees of the appropriate schemes respectively.

The general performance of the *Two-Level Hybrid Scheme* is much better when no contributory protocols are applied (which is expected). However, certain combinations of non-contributory with contributory schemes reduce the performance more than certain cases that combine two non-contributory protocols do.

Different combinations of protocols may prevail for different costs depending on the partial performance of each protocol for the particular cost. Among undeniable winners for most of the cases are combinations of OFT and of TGDH protocols with other efficient protocols with respect to a particular cost.

Example: The first six schemes combine *CBT* and *GKMP* protocols in different ways:
For all costs of the *Two-Level Hybrid Scheme* but for the Average Communication Cost (critical in MANETs), *GKMP-GKMP* and *Tree-GKMP* seem to be prevailing. For the Average Communication Costs however clearly *Tree-Tree* and *GKMP-Tree* are the winners. For GSC Initial and Operative Costs *SingleTree* presents the worst behavior (since it is loaded with the tasks of GSAs), but presents much better behavior along with *TwoTree* with respect to Communication Costs (it spares the communication between GSC and GSAs).

Calculation of C_r, C_{PE}, C_{SE}, C_{PD}, C_{SD} for different models of Public Key Infrastructure (PKI) and symmetric key encryption/decryption

Up to now we have done the major part of the work. We have taken two very popular protocols in network security for key distribution, the GKMP (Group Key Management Protocol), and the Logical Tree Scheme and we have combined these two protocols together in order to derive new hybrid key management protocols. Our aim is to analyze these hybrid protocols for all the possible combinations and model their behavior in respect to how costly the communications among users within those protocols are. This requires a very thorough grasp of how the protocols work individually, and in combinations. In order to derive all the cost functions we have to define exactly how our model hybrid scheme behaves in each case and design it step by step. So we developed an analytical model for each model that is supposed to be simulating the actual behavior of users in such a network and the communication, computational and storage cost they would produce by their actions.

Until now we used GKMP protocol and the Logical Tree Scheme. However, they both have advantages and disadvantages in comparison to each other. GKMP has less storage cost but more communication overhead than the Logical Tree Scheme. The latter reduces substantially the communication overhead, particularly in the cases that a component is deleted, at the expense however of storage and computational cost. So, the aim of this analytical work is to come with a hybrid model that exploits the advantages of the two schemes. We want to provide analytical functions that model the behavior of entities using these schemes so that we can have an actual means of comparison, and ultimately decide which scheme among all and under what circumstances manages to minimize communication, computational and storage cost. At some cases we will see that some of the hybrid schemes perform better (some of the cost functions are cheaper) and some not. However, with careful inspection we conclude that the results seem in accordance with the models.

What remains to be done is to determine the values of the variables: C_r, C_{PE}, C_{SE}, C_{PD}, C_{SD}.
The C_r variable is the random generator of a key. The C_{PE} and C_{PD} variables depend upon the Public Key Infrastructure (PKI) that we use and much research is allowed in this field.

The PKI is the rules of the protocol and the assumptions upon which we will base how will exchange keys via the public key encryption/decryption method. It has to do with the topology of the keys with the nature of the Certification Authority that authenticates the users, with the amount of trust placed on each entity, with the kind of authentication we use and above all what mechanism we choose for public key exchange. The most popular public key encryption/ decryption mechanisms currently are: RSA (Rivest, Shamir, Adleman), DH, GDH1, GDH2, GDH3, Burmester-Desmedt, Elliptic Curves, El Gamal Digital Signatures, ELK (Efficient Logical Key), OFT (One Way Function Protocol), Blom Key Distribution Scheme.

The Public Key Encryption/Decryption is called Asymmetric Encryption/Decryption as well, because the sender sends a message encrypted with key K1 that only the sender acquires, and the receiver decrypts it with key K2<>K1. So this mechanism is resistant to security attacks as eavesdropping for instance but the drawback is that it is very expensive, very slow due to the fact that it includes many exponentiations, and if we try to use it for the exchange of messages the enormous overhead of communication makes it inefficient and non-scalable.

The other mechanism for the exchange of messages among users is the private key encryption/decryption method or symmetric key encryption/decryption method. It is called symmetric because we use one key to carry out the encryption and its inverse to decrypt the message. This is an inexpensive, fast operation in comparison to the asymmetric one, but it is much more vulnerable to attacks, and requires very secure channels and many assumptions in order to be used in practice.

The mechanism that is used today for encryption/decryption is the best compromise of the two protocols. The exchange of messages is performed with the symmetric key encryption/decryption method, which is fast and inexpensive. To make sure that the private keys used in the symmetric protocol are safe from attacks and as securely distributed as possible, we choose the asymmetric method initially when we formulate the topology of the PKI to distribute those private keys. Of course we assume that every node is provided with its own private key and has published a public key that corresponds to its private, regardless of the topology of nodes and the mechanism of communication used. However, things are not that simple. Each time a node is added or deleted from a group and we have to update keys for security reasons, then we have to use the public (asymmetric) method again.

This is why so many protocols have been developed for the Public Key Exchange. The methods for symmetric key exchange are quite a few and similar to each other. In fact, what is used today is DES (Data Encryption Standard) algorithm and its variations: 3-DES, and other algorithms like IDEA, AES etc.

We want to avoid re-keying as much as possible because it is very expensive. If we could have the same performance in terms of security using instead of Rinjdael (standard for RSA) the RD5 for example, or a one-way function in general we would have a great computation gain. This is suggested in the ELK, OFT protocols and some more we are going to develop.

Algorithms for finding large primes:

The Theorem for Prime Numbers states that the number of primes less than n, P(n), as n tends to infinity is given by: $P(n) = n/\log_e n$. For simplicity we symbolize $\log_e n = \log n$.

The density of primes is thus $dP(n)/dn= 1/\log n - 1/\log^2 (n)$.
For large n the second term is small and we can omit it. So, for n=512 we get density of primes = (1:512) /log2=1:355
This suggests that primes be found by trial and error. Choose a large (uneven) integer n, test it for primality, if the test fails make change and try again. With 512 -bit integers we would expect success in 355/2= 178 attempts.
Generally the average number of attempts is: $\log_2 n \times (1/2\log 2) = 0.7 \times \log_2 n$

Existing algorithm due to Rabin and Miller (implemented by Knuth) (Algorithm P) for Primality test:

A Primality Test provides an efficient probabilistic algorithm for determining if a given number is prime [14], [15]. It is based on the properties of strong pseudoprimes. Given an odd integer n, let $n=2^r s+1$ with s odd. Then choose a random integer a: $1 \leq a \leq (n-1)$. If $a^s =1 \pmod n$ or a $^{2j\ s}$ =-1 (mod n) for some $0 \leq j \leq r-1$, then n passes the test. A prime will pass the test for all a.
The test is very fast and requires no more than $(1+o(1)) \log_2 n < 2 \times \log_2 n$ modular multiplications. Unfortunately, a number that passes the test is not necessarily prime. Monier and Rabin (1980) have shown that a composite number passes the test for at most 1/4 of the possible bases a.

Calculation of Key Generation cost: C_r

Overview of key generation
Two random numbers must be generated. For 512-bit key we get two 256-bit random numbers and 2^{254} possible numbers are obtained from 254 random bits.
The whole security of the system depends on the fact that the program must be capable of generating each and every one of these 2^{254} numbers with more-or-less equal probability. Next, from each random number generate a prime number. Starting from the given number, you look at each successive odd number to see if it is prime. A simple sieve is used to eliminate obvious cases, using a table of the first 1028 prime numbers. Anything that survives the sieve is subjected to Fermat's test (if `(x`$^{p-1}$`mod p)<>1`, then `p` is not prime). Next, multiply the two (alleged) primes P and Q to give the modulus N, and from P, Q and N the RSA public and private keys can be derived in the standard manner. One of the most popular secure pseudo-random number generators is the Blum-Blum-Shub (BBS) pseudo-random bit generator:

1. Calculate the operations for choosing x relatively prime to n. Pick up an odd number x<n. According to the Theorem of Prime Numbers we can find a relative prime number after length(n)/ (log2×2) attempts (this will be done only the first time we use the BBS pseudo-random number generator).
2. Calculate $x_j = x^2_{j-1} \pmod n$
3. Take b_j to be the least significant bit of x_j
4. Go to step 2. Derive the new x_j and take the new b_j and repeat as many times as the length of the bits we wish for our key.

A way to speed up this slow operation is: after every multiplication extract the k least significant bits of x_j. As long as $k \leq \log_2 \log_2 n$, the scheme is cryptographically secure. Assuming that length(x)=length(n) and selecting length(n)=K we find that for step 2 we need multiplication of K bits each time. So we need K^2 operations. We repeat step 2 for (K/k) times in order to derive K random generated bits. k is fixed s.t $k \leq \log_2 K$. So, the total number of operations for generation of one key is: $(K/(2\times\log 2)) + (K/k) \times K^2$.

$Cr = (K\times 0.7) \times K^2$ (this term is used the first time that the BBS pseudo-random generator is initiated, it can even be precalculated, so in fact it does not add to the overall complexity)+ $(K/k) \times K^2 = (K^3\times 0.7)+ (1/k) \times K^3$, where $1 \leq k \leq \log_2 K$

Remark: In general this particular algorithm generates a random key with computational complexity $O(K^3)$. In general there are algorithms that generate random keys with computational complexities that vary. In the bibliography we have encountered upon algorithms with the following complexities: $O(K^4)$, $O(\log_2 K \times K^3)$, $O(\log_2 K \times K^2)$, etc. However, we decide to present this particular algorithm because it is among the most popular for generating random keys. Another remark here is that this algorithm just generates random but not prime numbers. In what follows we are going to demonstrate an algorithm that generates random prime numbers.

Estimation of C_g

C_g is the cost for the one-way function that blinds a key. Generally, it is suggested that it can be calculated using the MD5 or the SHA method [10], [11]. We used MD5 to estimate its complexity.

MD5 takes a message and turns to multiple to 512 bits. If the length of the key is K, $T=\lceil \frac{K}{512} \rceil$ is the number times we need to multiply 512 bits (sixteen 32-bit words), to construct the padded message that we are going to use in the procedure.

In brief, MD5 makes four passes over each 16-byte chunk of the message. Each pass has a slightly different method of mangling the message digest. The message is processed in 512-bit blocks. Each stage computes a function based on the 512-bit message chunk and the message digest (128-bit quantity) to produce a new intermediate value for the message digest. At the end of the stage, each word of the mangled message digest is added to its pre-stage value to produce the post-stage value that becomes the pre-stage value for the next stage. The value of the message digest is the result of the output of the final block of the message. Every stage is repeated for all T 512-bits chunks of the message. In every stage a separate step is taken for each of the 16 words of the message. During every such step we have addition of 5 terms (including the previous blocks, the message digest and a constant). In two of the four stages the message digest function has complexity cost of 3x32 bits, and in the rest two stages 2x32 bits. The rest of the terms as well as the output of the digest function is 32 bits. The left rotates imply complexity of 32 bits as well.

Complexity for each chunk (512-bit) of message for all the 16 blocks of 32 bits for all stages:

1st stage:(4+3) 32+32, 2nd stage: (4+3) 32+32, 3d stage: (4+2) 32+32, 4th stage: (4+2) 32+32. Totally we have: 16x32x30 = 15360.

The complexity for all chunks of the message is approximately: Tx15360 bits = $\lceil \frac{K}{512} \rceil$x15360 = 30 K. So we can roughly estimate C_g =30K. The certain conclusion is that C_g is linear to the length of the key and also a cheap and secure operation if we use the MD5 method. So C_g = 30K

Calculation of the C_{PE}, C_{PD}:

RSA Algorithm:

Plaintext M is encrypted in blocks

Encryption: $C = M^e \bmod n$ Decryption: $M = C^d \bmod n$

Public key = <n, e> Private key = <n, d>

Requirements: There exist <n, e, d> such that $M = M^{ed} \bmod n$ for all M < n. It is easy to calculate M^e and C^d for all M < n. It is infeasible to find d given n and e

C_{PE}:

We gererate via the Random Key Generation method, two numbers. In order to use them for the RSA method we want these numbers p, q to be prime. We have already seen that the complexity for the generation of these two random numbers (not necessarily primes) p and q is: $(1/k) \times (\log_2 p)^3$, where $1 \leq k \leq \log_2 p$ and $(1/m) \times (\log_2 q)^3$, where $1 \leq m \leq \log_2 q$ respectively.
Thus, we need to select two large prime numbers p, q where $p \times q = n$. By method of trial and error we try to see which p, q will pass the *Primality Test*. We have already calculated that we need $\text{length}(n) \times 0.7$ trials in average to find number n that fulfills all the requirements. Each different n is candidate to pass the *Primality Test*, which gives complexity: $(1+o(1)) \times \log_2 n < 2 \times \log_2 n$ modular multiplications. Each modular multiplication has computational complexity $(\text{length}(n))^2$. So, the total complexity to derive the appropriate prime n given any random number m is: $1.4 \times ((\text{length}(n))^4)$. However, we need to consider the complexity for deriving this random number m in the first place. We have already calculated that this complexity is: $(1/k) \times (\log_2 m)^3$. Since we need $0.7 \times \text{length}(m)$ trials in average, we need to derive a random number $0.7 \times \text{length}(m)$ times, thus the overall complexity for deriving the required p and q random numbers are: $(1/k) \times 0.7 \times (\log_2 p)^4 + 1.4 \times (\log_2 p)^4 = ((0.7/k)+1.4) \times (\log_2 p)^4$ and $((0.7/k)+1.4) \times (\log_2 q)^4$ respectively. We also know that $\log_2 n = \log_2 p + \log_2 q$. Thus, $\log_2 p < \log_2 n$ and $\log_2 q < \log_2 n$ and we can use this bound to calculate the overall complexity of generating the random primes p and q. Thus, the overall complexity is: $((1.4/k)+2.8) \times (\log_2 n)^4 = ((1.4/k)+2.8) \times K^4$
We have just calculated the complexity for generating random primes to be used for the RSA key encryption/decryption algorithm.

Proceeding to the next steps, we want: $e \times d = 1 \mod (p-1) \times (q-1)$. If e is known we can derive d with O(1) calculations.
So we need to calculate the encryption of the session key with length K.
In many versions of RSA, e is assumed fixed with length K_e. A popular value for e is $2^{16}+1$ or 3.
We find that in the worst case we perform 2^*K_e modular multiplications and 2^*K_e divisions, and $2^*\log K_e$ other operations thus the total complexity is approximately $(4K_e+2\log K_e) \times K^2$ assuming that $\text{length}(n)=\log_2 n=\text{length}(m)=K$. Each of these operations has a cost analogous to the length of the message m (in our case the length K of the session key). At every step we multiply K either with itself or with the outcome of a previous multiplication truncated with modulo n so the result is always less than n. Roughly the total number of operations are: $(4 \times K_e + 2 \times x\log_2 K_e) \times (\text{length}(n))^2$.
So the final computation cost for the C_{PE} variable is: $C_{PE} = (4 \times K_c + 2 \times \log_2 K_c)$ x K^2

Observation: From equation:$e \times d = 1 \mod (p-1) \times (q-1)$ we can see that $K_e + K_d \times \geq \log_2 p + \log_2 q = K$. Thus, the selection of K_e affects the selection of K_d and vice versa. The most popular value for e is 3. Generally it is preferred d to be very large for the decryption algorithm, since the larger the d the more difficult is for an attacker to break the algorithm. In the case that we select e=3, we select a number d with size $K_d \approx K$. This is why, in the RSA algorithm, the encryption is much faster than the decryption.

C_{PD}:

For the decryption the same idea is adopted. Things however are slightly easier. We already have n, e, d and we only need to carry out the decryption of the message: $m = c^d \mod n$. If we use Shamir's assumption for the unbalanced RSA that $c^d \mod n$ can be reduced to $r^{dl} \mod p$ where $d1 = d \mod \phi(p)$ and $r = c \mod p$, we achieve a speedup of length(p). According to Shamir's proposition we have roughly:
$n=p \times q$ but also $\lg n = 10 \times \lg n'$ and $\lg p = 2 \times \lg p'$, where the same equation used to hold for n', p': $n' = p' \times q'$. From these equations we see that $\lg n/\lg p > 5$, and this is to say that the use of mod p instead of mod n produces as decryption cost more than 25 times less than the encryption cost:

A similar idea for the decryption used in the Chinese Remainder Theorem: we can speed decryption up to four times by computing: c^d mod p and c^d mod q instead. The Chinese Remainder Theorem then allows us to deduce c^d mod pq = c^d mod n. The idea of the Chinese Remainder Theorem is used for the decryption only, since p and q are known only to the member that decrypts the message. In the case of encryption, the member encrypting a message with the public key does not know p and q (unless it has created the public-secret pair).
However, this operation doesn't necessarily make the decryption faster than the encryption. In some asymmetric key systems such as ECC and Braid Method, the decryption speed is faster than the encryption speed. In systems like NTRU and RSA encryption is faster. In RSA this is due to the choice of e that is usually a small number. However, in RSA the encryption speed up achieved by the efficient choice of e for the encryption, is larger than the decryption speed up achieved by the Chinese Remainder Theorem for the decryption. As we mentioned earlier if we select e=3 then we have to select a number d with size $K_d \approx K$.

Thus for the decryption cost we easily derive the following: $C_{PD} = ((4/25)\times K_d+(2/25)\times \log_2 K_d) \times K^2$
Finally: $C_{PD} = ((4/25)\times K_d+(2/25)\times \log_2 K_d) \times K^2$

Special case: If e=3 then $C_{PD} =((4/25)\times K+(2/25)\times \log_2 K)\times K^2=O(K^3)$ whereas $C_{PE}=O(K^2)$.

DES and calculation of C_{SE}, C_{SD}

DES is a *block cipher*: it operates on plaintext blocks of a given size (64-bits) and returns ciphertext blocks of the same size. Thus DES results in a *permutation* among the 2^64 possible arrangements of 64 bits. Each block of 64 bits is divided into two blocks of 32 bits each, a left half block L and a right half R. DES operates on the 64-bit blocks using *key* sizes of 56- bits. The keys are actually stored as being 64 bits long, but every 8th bit in the key is not used. DES encrypts and decrypts data in 64-bit blocks, using a 64-bit key. It takes a 64-bit block of plaintext as input and outputs a 64-bit block of ciphertext. It has 16 rounds, meaning the main algorithm is repeated 16 times to produce the ciphertext. It has been found that the number of rounds is exponentially proportional to the amount of time required to find a key using a brute-force attack. So as the number of rounds increases, the security of the algorithm increases exponentially

We will calculate the number of computations by following the algorithm step by step:

```
DES(X,K,E)
X is a 64-bit input value, K is a 64-bit key, E is TRUE if we are encrypting, else FALSE
(1) Apply permutation IP to X, resulting in value LR (64 bits).
(2) Apply selector PC1 to K, resulting in value CD (56 bits).
(3) For ROUND = 1, 2, ..., 16:
(3a) if E is TRUE then   if SHIFTS[ROUND] = 1
     then apply permutation LSH1 to CD (in place) else apply permutation LSH2 to CD (in place)
(3b) Apply selector PC2 to CD resulting in value KI (48 bits)
(3c) If E is FALSE then if SHIFTS[17-ROUND] = 1
     then apply permutation RSH1 to CD (in place) else apply permutation RSH2 to CD (in place)
(3d) Apply selector E to LR, resulting in value RE (48 bits).
(3e) XOR RE with KI, resulting in value RX (48 bits).
(3f) Break RX into 8 6-bit blocks, replace the i-th block Yi with the result of looking up Yi in S-
     box i. Concatenate these 4-bit results together to get the 32-bit value SOUT.
(3g) Apply permutation P to SOUT resulting in value FOUT (32 bits).
```

(3h) Replace the left half of LR with the XOR of FOUT and the left half of LR.
(3i) If ROUND is not 16, apply permutation SWAP to LR (in place).
(4) Apply permutation IPINV to LR resulting in value OUT (64 bits).
Return OUT as the result of this DES operation.

We have permutation of the initial 64-bit input, then 16 DES rounds for each one of which we generate the per round keys and finally a last permutation of the 64-bit output. These permutations are not random, they have a specific structure so they require a fixed number of operations: $C_{permute}$. To generate the per round keys we do an initial permutation of the 56 useful bits of key, to generate a 56-bit output which it divides into two 28- bit values, called Co, Do. These permutations again require a fixed number of operations: $C_{permute}$
We permute 24 of those bits as the left half and 24 as the right half of the per round key. A completely randomly chosen permutation of k bits would take about $k\times\log_2 k$ bits. So we have two permutations that take $24\times\log_2 24$ bits each. From every bit we produce $\log_2 24$ more bits.
In a DES round in encryption the 64 bits are divided into two 32-bit halves called L_n and R_n. The output generates 32- bit quantities L_{n+1} and R_{n+1}. Their concatenation is the 64-bit output of the round. We have to make extra calculations only to produce $R_{n+1.}$ From R_n and per round key of 48-bits we produce an outcome of 32 bits which is XORed with L_n This procedure requires about 8 operations for the 32 bits extension to 48 bits, 64 for the mapping in S boxes, 32 for the last XORing. So, totally we need about $O(2\times C_{permute}+24\times\log_2 24 +16\times 144)$ operations for a DES encryption. It is proven that we need the same amount of operations for the DES decryption.
However we might want a key length or data length larger than 64 bits. We still can use the DES method for larger encrypted messages or larger encryption keys the following way. We will break the whole decryption into chunks of 64 bits and will use DES to encrypt each block. If the key has more than 64 bits we can break the encryption key into chunks as well and perform the encryption of the message. In our case this would be the appropriate thing to do because the session key produced initially from RSA would have length larger than the length of the key produced by DES. So, we need to perform DES encryption and decryption K/64 times roughly.

Now we can calculate the computation costs C_{SD} and C_{SE} that are going to be roughly the same. So:
$C_{SE}=C_{SD}= (K/64)\times O(2\times C_{permute} + 24\times\log_2 24 +16\times 144)$ $C_{SE}=C_{SD} < (K/64) \times 2500$
We see that C_{SE}, C_{SD} have cost linear to the key length so we can attribute to them the final form: $C_{SE}=C_{SD}= C_{DES} \times K$, where $35\leq C_{DES} \leq 80$

3-DES: It is a variation of simple DES. 3-DES provides double the security of DES because it literally uses the DES encryption scheme, then the DES decryption scheme and again the DES encryption scheme to provide the final encrypted key: If m is a 64-bits plaintext message then 3-DES produces the following encrypted message: $E_{k1}(D_{K2}(E_{K3}(m))))$ or $E_{k1}(E_{K2}(E_{K3}(m))))$ depending on which version of 3-DES we are using. This message may have length from 40 to 198 bits. Its complexity is roughly 3 times the complexity of DES.

Exponentiation

Exponentiation is an operation with cubic complexity: if the size of number that is involved (modulus) is doubled, the number of operations increases by a factor of $2^3 = 8$.
Exponentiation is done by looking at the bit pattern of the exponent as successive powers of 2, and then successively squaring the argument and multiplying with modulus as necessary [14], [15]. At worst this involves $2\times l$ modular multiplications or $4\times l$ multiplications and divisions where l is the length of the exponent. Thus, 16 bit exponentiation involves $64 \text{ x } (32)^3 = 2^{21} = 2\text{x}10^6$ operations. The general form is: $4\times l\times(2\text{\^{}}(\log_2 l+1))^3 = 4\times l\times(2\times l)^3=32\times l^4$.

In the case where the exponent and the modulo N have the same size K in bits this complexity could be expressed as: $32\times K^4$
The complexity of this general formula for modular exponentiation can be further refined as we see in what follows:
A 512 bit implementation of RSA in software on most computers will take about one second. But if exponentiation is done under a modulus, things are a lot easier as we see from the following example:
Suppose we want to take 299 to the 153rd power, under a modulus of 355. The first thing we do is to note that 153 equals 128 + 16 + 8 + 1 (binary decomposition), and that therefore,
299^153 = 299^128×299^16×299^8×299^1.
Now, we can compute the term on the far right; it's 299. Under the modulus, it's still 299. But knowing the one on the right, we can compute the one next to it: Doubling the powers by squaring the numbers, and applying the modulus to intermediate results, gives:
299 mod 355 is 299. (299 = 299^1 mod 355)
299^2 = 89401, but 89401 mod 355 is 296. (296 = 299^2 mod 355)
296^2 = 87616, but 87616 mod 355 is 286. (286 = 299^4 mod 355)
286^2 = 81796, but 81796 mod 355 is 146. (146 = 299^8 mod 355)
146^2 = 21316, but 21316 mod 355 is 16. (16 = 299^16 mod 355)
16^2 = 256, and 256 mod 355 is 256. (256 = 299^32 mod 355)
256^2 = 65536, but 65536 mod 355 is 216. (216 = 299^64 mod 355)
216^2 = 46656, but 46656 mod 355 is 151. (151 = 299^128 mod 355)

Thus, 299^153 mod 355 = (151 ×16 ×146 ×299) mod 355 which now looks a whole lot easier. At this point total multiplications/divisions are in the worst case: $4\times l\times n^2$, where n is the modulo.

Now we proceed by calculating the product and applying the modulus to intermediate results:
151 ×16 = 2416, but 2416 mod 355 is 286.
286 ×146 = 41756, but 41756 mod 355 is 221.
221 ×299 = 66079, but 66079 mod 355 is 49.
Thus, 299^153 mod 355 = 49.
For this part total operations are in the worst case: $2\times \log_2 l \times n^2$

And that is how you take a large number to a large power under a large modulus using the most efficient algorithm. The constant application of the modulus operation to intermediate results prevents you from having to deal with any number larger than the square of the modulus in any case.
So, if l is the exponent size and n the modulo size then the total number of operations is: $(4\times l+2\times \log_2 l)\times n^2$. The general complexity is: $O(l\times n^2)$. For the case that the exponent and the modulo have the same size we get total complexity of: $O(n^3)$.

In the following Table an approximate estimation of the cost (in bits) of the public encryption/decryption parameters C_{PE} and C_{PD} respectively, of the symmetric encryption/ decryption parameters C_{SE} and C_{SD} respectively, of key generation C_r, of exponentiation C_E and of hashing C_g operations that are used for the estimation of computational costs is presented.

C_r	$C_{rr}=(K\times \log 2/2) + (K/k) \times K^2$ pseudo- random number generation $C_r= ((1.4/k)+2.8) \times K^4$ pseudo-random generation for primes (used for RSA keys)
C_{PE}	$(4\times K_e+2\times \log_2 K_e) \times K^2$
C_{PD}	$((4/25)\times K_d+(2/25)\times \log_2 K_d)\times K^2$

C_{SE}	$C_{DES} \times K$
C_{SD}	$C_{DES} \times K$
C_E	$(4\times l+2\times \log_2 l)\times K^2$
C_g	$30\times K$

Table 5: Costs of parameters used in the protocols.

For the asymmetric encryption/decryption we use the RSA method and for the symmetric encryption/decryption we use the DES method:
$C_{SE}=C_{SD}= C_{DES} \times K$, where $35\leq C_{DES} \leq 80$, $1\leq k\leq \log_2 K$
Most common selections for K_e and K_d are: $K_e =2$, $K_d \approx< K$.
For the exponentiation we assume that l is the size of the exponent and K is the size of the modulo n. We use this expression for the complexity among the variations existing in the current bibliography.

Computing C_{PE} and C_{PD} with the Diffie-Hellman Algorithm

The Diffie-Hellman (DH) public key cryptosystem predates RSA and is the oldest public key system still in use. It offers better performance than RSA for what it does. Basically it is used for private key establishment. DH allows two individuals to agree on a shared key even though they can only exchange messages in public.
Description:
There are numbers p (prime which is about 512 bits), number g<p. They can be known beforehand and can be publicly known.
Each user chooses a 512-bit number at random and keeps it secret. Assume that our two users are named A and B. A's secret number is S_A and B's secret number is S_B. Each raises g to their secret number, mod p. So A computes some number T_A and B computes some number T_B. They exchange their Ts. Finally, each raises the received T to their secret number. $T_A = g^{SA}$ mod p and $T_B = g^{SB}$ mod p. So A computes T_B^{SA} mod p and B computes T_{Aa}^{SB} mod p. They will both compute the same number since: $T_B^{SA} = (g^{SB})^{SA} = g^{SB\,SA} = g^{SA\,SB} = (g^{SA})^{SB} = T_A^{SB}$ mod p.
Nobody else can calculate $g^{SA\,SB}$ in a reasonable amount of time even though they know g^{SA} and g^{SB}.
Now, if we want to do encryption with DH the procedure is the following:
Both parties compute a public key, which consists of the three numbers <p, g, T>, where $T=g^s$ mod p, for the private key S. So, assume that B has published a $<p_B, g_B, T_B>$. If A wants to send to B a encrypted message, A picks a random number S_A , computes g_B^{SA} mod p_B and computes $K_{AB} = T_B^{SA}$ mod p_B, and uses that as the encryption key to share with Bob. So A uses K_{AB} to encrypt the message according to any secret key cryptographic technique, and sends the encrypted message along with g_B^{SA} mod p_B to B. B raises g_B^{SA} mod p_B to B's own secret S_B, and thereby calculates K_{AB} which enables him to decrypt the message.
We can see that encryption of messages with DH is not cheaper than RSA, in fact it ultimately leads to RSA or a similar technique and it can be even more expensive.
The good part is that we can use simple DH to exchange only the private key that the two users are going to use for then on, since they can exchange messages with the symmetric key method. In this case we don't have to compute C_{PE} and C_{PD} but only a cost for the shared key that is going to be the same for both users. DH is much cheaper than RSA because we don't have to find any primes now. We only need to do two exponentiations. The cost for that is already know, so roughly we need:
$(4 \text{ x } K_c+2\lg K_c) \text{ x } p^2$. Since p=k in terms of length we can see that the complexity of the method is still $O(K^2)$, even though the method is cheaper.
The same is the case with other similar methods that have exponentiations such as El Gamal Digital Signatures, Elliptic Curves etc etc.

Current research is focused on developing group mechanisms for reducing the cost of PKI. In such methods the session key establishes among the members of a group without any prior private keys between two members. Such methods are: GDH.1, GDH.2, GDH.3, Burmester-Desmedt, ELK, OFT. We will analyze most of them to get an idea of their potential.

Protocols for Group Communication

Group Key Distribution GDH.1

This is a quite simple and straightforward protocol. It consists of two stages: up-flow and down-flow. The purpose of up-flow stage is to collect contributions from all group members. As shown in the figure, M_i on the up-flow is to compute $a^{N1 \ldots Ni-1}$ the highest numbered incoming intermediate value- to the power of N_i, append it to the incoming flow and forward all to M_{i+1}. For example M_4 receives the set $\{a^{N1}, a^{N1\ N2}, a^{N1\ N2\ N3}\}$ and forwards to M_5: $\{a^{N1}, a^{N1\ N2}, a^{N1\ N2\ N3}, a^{N1\ N2\ N3\ N4}\}$.

$$\begin{tikzcd} M_i \arrow[rr, "\{\alpha^{\Pi(N_k \mid k \in [1,j])} \mid j \in [1,i]\}"] & & M_{i+1} \end{tikzcd}$$

Stage 1 (Upflow): Round i; $i \in [1, n-1]$

$$\begin{tikzcd} M_{n-i} & & M_{n-i+1} \arrow[ll, "\{\alpha^{\Pi(N_k \mid k \notin [i,j])} \mid j \in [1,i]\}"'] \end{tikzcd}$$

Stage 2 (Downflow): Round $(n-1+i)$; $i \in [1, n-1]$

To summarize, the up-flow state each group member performs one exponentiation and an up-flow message between M_i and M_{i+1} contains i intermediate values. The final transaction in the up-flow stage takes place when the highest-numbered group member M_n receives the up-flow message and computes $(a^{N1 \ldots Ni-1})^{Nn}$ which is the intended group key K_n. After obtaining K_n, M_n initiates the down-flow stage. In this final stage each M_i performs i exponentiations: one to compute K_n and (i-1) to provide intermediate values to subsequent (lower-indexed) group members. For example, assuming n=5, M_4 receives a down-flow message: : $\{a^{N5}, a^{N1\ N5}, a^{N1\ N2\ N5}, a^{N1\ N2\ N3\ N5}\}$. First, it uses the last intermediate value in the set to compute K_n. Then, it raises all remaining values to the power of N_4 and forwards the resulting set: $\{a^{N5\ N4}, a^{N1\ N5\ N4}, a^{N1\ N2\ N5\ N4}\}$ to M3. In general the size of down-flow decreases on each link; a message between M_{i+1} and M_i includes i intermediate values.

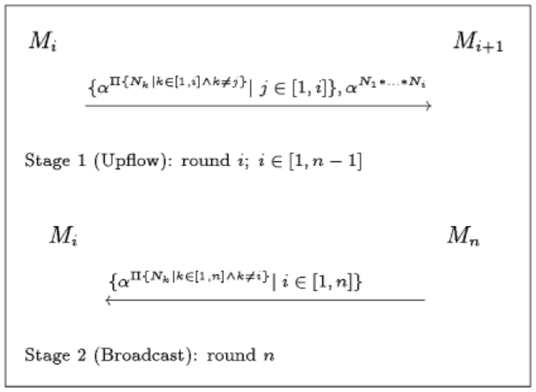
In summary, GDH.1 has the following characteristics:

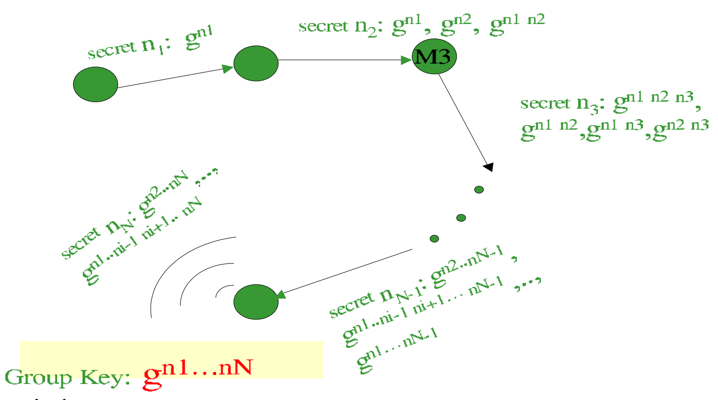
Rounds	Messages	Combined Message Size	Exp/s per M_i	Total Exps/s.
2(n-1)	2(n-1)	(n-1)n	(i+1), for i<n, n for M_n	(n+3)n/2-1

Group Key Distribution GDH.2

In order to reduce the number of rounds in GDH.1 we modify the protocol as shown in the following figure. The up-flow stage is still used to collect contributions from all group members; the only change is that each M_i now has to compose I intermediate values – each with (i1) exponents – and one cardinal value containing i exponents. For example M_4 receives a set: $\{a^{N1\ N2\ N3}, a^{N1\ N2}, a^{N1\ N3}, a^{N3\ N2}\}$ and outputs a set: $\{a^{N1\ N2\ N3\ N4}, a^{N1\ N2\ N3}, a^{N1\ N2\ N4}, a^{N1\ N3\ N4}, a^{N3\ N2\ N4}\}$

The cardinal value in this example is a $^{N1\ N2\ N3\ N4}$. By the time the up-flow reaches M_n the cardinal value becomes $a^{N1\ldots Nn-1}$. M_n is thus the first group member to compute the key K_n. Also, as the final part of the up-flow stage, M_n computes the last batch of intermediate values.
In the second stage M_n broadcasts the intermediate values to all members.


M_i
M_{i+1}
$\{\alpha^{\Pi\{N_k \mid k\in[1,i]\wedge k\neq j\}} \mid j\in[1,i]\}, \alpha^{N_1*\ldots*N_i}$
Stage 1 (Upflow): round i; $i\in[1,n-1]$
M_i
M_n
$\{\alpha^{\Pi\{N_k \mid k\in[1,n]\wedge k\neq i\}} \mid i\in[1,n]\}$
Stage 2 (Broadcast): round n


secret n_1: g^{n1}
secret n_2: g^{n1}, g^{n2}, $g^{n1\ n2}$
M3
secret n_3: $g^{n1\ n2\ n3}$, $g^{n1\ n2}$, $g^{n1\ n3}$, $g^{n2\ n3}$
secret n_N: $g^{n2..nN}$,..., $g^{n1..ni-1\ ni+1..\ nN}$
secret n_{N-1}: $g^{n2..nN-1}$, $g^{n1..ni-1\ ni+1\ldots\ nN-1}$,..., $g^{n1\ldots nN-1}$
Group Key: $g^{n1\ldots nN}$

In summary, GDH.2 has the following characteristics:

Rounds	Messages	Combined Message Size	Exp/s per M_i	Total Exps/s.
n	n	(n-1)(n/2+2)-1	(i+1), for i<n, n for M_n	(n+3)n/2-1

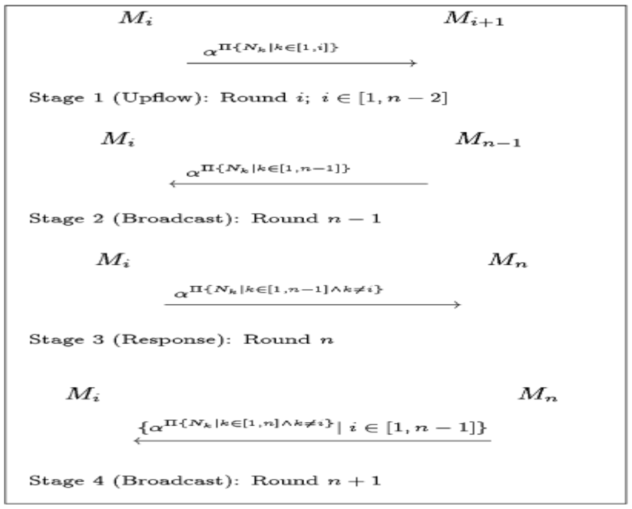
So, in GDH.2 more than in GDH.1, the highest-indexed group member M_n plays a special role by having to broadcast the last round of intermediate values. The main advantage of GDH.2 is due to its low number of protocol rounds; n as opposed to twice as many in GDH.1

Practical Considerations

GDH.1 and GDH.2 offer the following advantages:
1. *No a prior ordering of group members.* The sequencing and numbering can be done real time. The first participant automatically becomes M_1.
2. *No synchronization:* no clock or round synchronization is necessary.
3. *Small number of exponentiations:* it depends on the participant's index i. On the average, each M_i will perform n/2 exponentiations
4. *Minimal Total Number of messages:* at least n messages are required in any group key agreement protocol, i.e., each M_i has to contribute its own share of the key.
5. *Minimal Number of rounds for asynchronous operation (GDH.2):* In order to construct a true DH key-K_n = $a^{\pi\{Nk/K\in[1\ldots n]\}}$ - each participant needs to contribute its own exponent. Assuming that the protocol starts asynchronously (first round initiated by M_1), only one M_i can add its own exponent in a given round.
6. *Minimal number of messages sent/received by each participant (GDH.2):* Assume that there exist a protocol that constructs a DH key and requires each M_i (other than M_1 or M_n) to send one and receive one message. One possibility is that M_i receives a message before sending one. In that case, from the message received M_i must be able to construct K_n –since no further messages will be received. The message received must contain $a^{\pi(\{N_1,\ldots,N_n\}-N_i)}$, since there is no other way for M_i to construct K_n. This means that every M_j (j<>i) has already contributed its exponent N_j, and, hence, already received (except M_1) and subsequently sent, a message. Therefore M_i =M_n and I=n since this can only take place in the n (n-1)st round. If we assume that M_i sends a message before receiving one then M_i = M_1 because this protocol runs asynchronously, i.e., only M_1 can start the protocol.
7. *Implementation Simplicity:* just like 2-party DH, GDH.1 and GDH.2 require only the modular exponentiation operation and the random number generator for the protocol execution. So, given a black box realization of 2-party DH, GDH.1/2 can be implemented thereupon without any additional arithmetic operations.

Group Key Distribution GDH.3

In certain environments it is desirable to minimize the amount of computation performed by each group member. This is particularly the case in very large groups. Since GDH.1, GDH.2 both require (i+1) exponentiations from every M_i, the computational burden increases as the group size grows. The same, of course, is true for message sizes.



M_i
M_{i+1}
$\alpha^{\Pi\{N_k|k\in[1,i]\}}$
Stage 1 (Upflow): Round i; $i \in [1, n-2]$
M_i
M_{n-1}
$\alpha^{\Pi\{N_k|k\in[1,n-1]\}}$
Stage 2 (Broadcast): Round $n-1$
M_i
M_n
$\alpha^{\Pi\{N_k|k\in[1,n-1]\wedge k\neq i\}}$
Stage 3 (Response): Round n
M_i
M_n
$\{\alpha^{\Pi\{N_k|k\in[1,n]\wedge k\neq i\}} | i \in [1, n-1]\}$
Stage 4 (Broadcast): Round $n+1$

In order to address these concerns, GDH.3 protocol has been constructed. It consists of four stages. In the first stage we collect contributions from all group members similar to the up-flow stage in GDH.1. After processing the up-flow message M_{n-1} obtains $a^{\pi\{Nk/K\in[1,n-1]\}}$ and broadcasts this value in the second stage to all other participants. At this time, every M_i (I<>n) factors out its own exponent ad forwards the result to M_n. (That factoring out N_i requires computing its inverse N_i^{-1}. This is always possible if we choose the group q as group of prime order). In the final stage, M_n collects all inputs from the previous stage, raises very one of them to the power of N_n and broadcasts the resulting n-1 values to the rest of the group. Every M_i now has a value of the form $a^{\pi\{Nk/K\in[1,n]\wedge k\neq i\}}$ and can easily generate the intended group key K_n.

This protocol has two appealing features: Constant message sizes and constant and small number of exponentiations for each M_i (except for M_n with n exponentiations required)

Rounds	Messages	Combined Message Size	Exp/s per M_i	Total Exps/s.
n+1	2n-1	3(n-1)	4, for i<n, 2 for M_{n-1} n for M_n	5n-6

Membership Addition and Deletion for GDH.2 and GDH.3

Addition of new members in GDH.2 can be achieved as follows:

1.Assume that M_n saves the contents of the up-flow message (Stage 1, round n-1)

2.M_n generates a new exponent N_n' and computes a new up-flow message (using N_n' and not N_n):

{ $a^{\pi\{Nk/K\in[1,n]\wedge k\neq j\}}$ | j in [1,n]}, $a^{N1\ N2\ldots Nn-1\ Nn'}$ and sends it to the new member M_{n+1}.

3. M_{n+1} generates its own exponent and computes the new key $K_{n+1} = a^{N1\ldots Nn'\ Nn+1}$

4. Finally, as the normal protocol run, M_{n+1} computes n sub-keys of the form:

{ $a^{\pi\{Nk/K\in[1,n]\wedge k\neq j\}}$ }|j in [1,n]} and broadcasts to the other group members.

Member addition in GDH.3 is almost identical to that in GDH.2. M_n has to save the contents of the original Broadcast and Response messages. M_n generates a new exponent and with it, computes a new set of sub-keys which it forwards to the new member M_{n+1}. M_{n+1} computes the new key K_{n+1} and adds its own exponent to teach of the n sub-keys it received. Finally, M_{n+1} broadcasts the sub-keys and all members compute K_{n+1}.

The extensions to GDH.2/3 are quite straight-forward and require only to additional rounds per each new member. The new key, K_{n+1} is easily computable by all parties and retains the same secrecy properties as K_n. However, while all other group members compute K_{n+1} with a single exponentiation, M_n is required to perform n exponentiations n addition to generating a new exponent. This extra burden on M_n may be undesirable.

Protocol extensions for member deletion in both GDH.2 and GDH.3 are very similar to those for member addition. Let M_p be the evicted member of the group, with p<>n. M_n plays a special role by generating a new exponent N_n'. M_n computes a new set on n-2 sub-keys: { $a^{\pi\{Nk / K\in[1,n]\wedge k\neq j\}}$ } |j in [1,n], k<>p} and broadcasts them to all group members. Since $a^{N1 *,\ldots,Np-1,\ Np+1,\ldots,Nn-1,\ Nn'}$ is missing from the set of broadcasted sub-keys, the newly excluded M_p is unable to compute the new group key. In the event that M_n is to be evicted from the group, M_{n-1} assumes the special role as described above.

Burmester-Desmedt

Burmester and Desmedt present a much more efficient protocol that is executed in only three rounds:
1. Each user M_i generates its random exponent N_i and broadcasts $z_i = a^{Ni}$.
2. Every M_i computes and broadcasts $X_i = (z_{i+1}/z_{i-1})^{Ni}$
3. M_i can no compute the key $K_n = (z_{i-1})^{nNi} (X_i)^{n-1}(X_{i+1})^{n-2}\ldots X_{i-2} \bmod p$

The key defined here is different from the previous protocols, namely $K_n = a^{N1\ N2+N2\ N3+\ldots+Nn\ N1}$. The protocol is proven secure provided the DH problem is intractable. In summary, the BD protocol has the following characteristics:

Rounds	Messages	Combined Message Size	Exp/s per M_i	Total Exps/s.	Divisions per M_i
2	2n	2n	n+1	(n+1)n	1

An important advantage of BD protocol its cheap exponentiations. This makes for big savings in computation.

	ING	BD	BD*[4]	GDH.1	GDH.2	GDH.3
rounds	$n-1$	2	$2n-1$	$2(n-1)$	n	$n+1$
total messages	$n(n-1)$	$2n$	$2n-1$	$2(n-1)$	n	$2n-1$
combined msg size	$n(n-1)$	$2n$	$2n$	$n(n-1)$	$\frac{(n+3)n}{2}-3$	$3(n-1)$
messages sent per M_i	$n-1$	2	2	2 1 for M_1, M_n	1	2 1 for M_n
messages received per M_i	$n-1$	$n+1$	$n+1$	2 1 for M_1, M_n	2 1 for M_1, M_n	3 n for M_n
exponentiations per M_i	n	$n+1$	$n+1$	$i+1$	$i+1$	4 2 for M_{n-1} n for M_n
total exponentiations	n^2	$(n+1)n$	$(n+1)n$	$\frac{(n+3)n}{2}-1$	$\frac{(n+3)n}{2}-1$	$5n-6$
synchronization	Y	Y	Y	N	N	N
DH key	Y	N	N	Y	Y	Y
symmetry	Y	Y	Y	N	N	N

From the analysis of those protocols, and their mathematical models, we derived important information about their costs for different features. What is very obvious is that these protocols is some aspects as in rounds, in total messages in some of these protocols, in messages sent per M_i and in messages received per M_i seem to be superior from the GKMP and the Logical Tree based protocols. Even the operations of adding and deleting a member are not any more costly than in GKMP. In the latter case logical tree seems to be the winner. The only drawback of those protocols is that the exponentiations for M_i are too many. If we think back about GKMP and Logical Tree based protocol, each user needs to perform one exponentiation but for the GSA or GSC given the case. The total exponentiations in the case of GKMP were never more than 2n, and so is the case in Logical Tree based Schemes. However, the only way to evaluate which protocol performs better at this point is to incorporate those protocols into the six models that have been developed previously. It seems that it is a fruitful option, that we replace the GKMP scheme with one of this protocol in all six models and see if we have a better performance. We can do the same procedure for Logical Trees and draw our results. This is a long on going research because so many factors have to be taken into account and so many cases have to be analyzed. This was just a demonstration of the potential of the existing protocols, their capabilities and how they can merge with the existing ones and enhance the overall performance.

Evaluation of Cost for the GDH.2 protocol

GSC Storage: The M_n member plays the role of the GSC since it needs to store the whole up-flow message that consists of n intermediate values each with n-1 exponents and of 1 value of n exponents. So in essence, we can assume that the GSC stores n+1 messages of length less or equal to the number of bits of the ultimate key K.
Member Storage: need to store the session key only.
Initial communication: n rounds (n-1 unicast messages from the M_i member to the M_{i+1} member of length (i+1) K bits and one multicast message from member M_n to the rest n-1 members of length n K bits). The total message length in average is: $\sum_{i=1}^{n-1}(i+1)K$ +nK = $(n^2+3n-2)K/2$
GSC initial computation: The GSC (M_n) has to carry out n exponentiations and must compute the secret number N, which will be derived from the pseudo-random number generator C_{rr} for general numbers (for random numbers, not for primes).
Member initial computation: Each member M_i does (i+1) exponentiations of the same complexity each. We can say that in average the exponentiations a member carries out are n/2. And it does one more exponentiation when it gets the stream message in order to construct the session key K_n. It also generates its random secret by using the pseudo random number generator C_{rr}.
GSC addition computation: M_n generates a new exponent N_n' and computes a new up-flow message using N_n' not N_n and sends it to the new member $M_{n+1.}$ Now member M_{n+1} becomes the GSC and computes the new key K_{n+1}. And it computes and broadcasts the n sub-keys to the other group members. So this member computes n+1 intermediate values, it does n+1 exponentiations. Again, it also generates its random secret by using the pseudo-random generator.
Member addition computation: The member gets the broadcast stream and it only needs to do one exponentiation to get the key K_{n+1}, and one random number generation.
Add communication: The M_n member sends to the M_{n+1} member an up-flow message of (n+1)K length. Then M_{n+1} member broadcasts to all n members its n+1 intermediate values, of total length (n+1)K. So the total length of the add communication is 2(n+1) K.
GSC deletion computation: If member M_p is evicted then M_n that has asserted the role of the GSC computes a new set of n-2 sub-keys. What is missing is the term $\alpha^{N_1*...N_{p-1}*N_{p+1}...N_{n-1}*N_n'}$ $\alpha^{N_1*...N_{p-1}*N_{p+1}...N_{n-1}*N_n'}$ so that M_p cannot compute the new key. So the procedure is exactly like GSC add computations, only that now M_n has to do n-1 exponentiations and one random number generation C_{rr}.
Member deletion computation: it is exactly the same for member addition computation.
Delete communication: Here only member M_n is broadcasting a message of length n-1 to the rest of the members. So the total overhead for delete communication is: (n-1)K

Evaluation of Cost for the GDH.3 protocol

GSC Storage: The M_n member plays the role of the GSC since it needs to store the whole up-flow message that consists of n intermediate values each with n-1 exponents and of 1 value of n exponents. So in essence, we can assume that the GSC stores n+1 messages of length less or equal to the number of bits of the ultimate key K. M_n finally stores also the value computed initially by M_{n-1} of K bits. It is going to need it in the case of member addition.

Member Storage: need to store the session key only.
Member Initial Computation: Each member M_i does 1 exponentiation in the up-flow message from member M_i to M_{i+1}.then after it receives the broadcast message from M_{n-1} member it does one inverse exponentiation to eliminate its own exponent and forwards the result to member M_n or the GSC. M_n computes the final key by doing one random number and broadcasts the results to all the n-1 members that have to raise the appropriate value to their exponent to get the session key. We have to note here that the member has to compute its inverse exponent, which is an exponentiation too. So exponentiations per M_i are: 4 for i<(n-1), 2 for i=(n-1), n for i=n.
GSC Initial Computation: The GSC (M_n) has to carry out n exponentiations as we stated above, and one random number generation C_{rr}.
Initial Communication: Initially we have n-2 rounds (n-2 unicast messages from the M_i member to the M_{i+1} member of length (i+1) K bits). Then we have one broadcast message from member M_{n-1} to the rest n-1 members of length K bits. After that we have (n-1) unicasts from each member M_i to the member M_n of length K bits again, and then a broadcast from M_n of n-1 values of length K. The total message length in avg. is: $\sum_{i=1}^{n-2}(i+1)K$ +K+(n-1)K+(n-1)K=(n-2)(n-1)K/2+3(n-1)K = $(n^2+3n-4)K/2$
GSC addition computation: Exactly the same as in GDH.2
Member addition computation: Exactly the same as in GDH.2
Add communication: Exactly the same as in GDH.2
GSC deletion computation: Exactly the same as in GDH.2
Member deletion computation: Exactly the same as in GDH.2
Delete communication: Exactly the same for delete communication for GDH.2

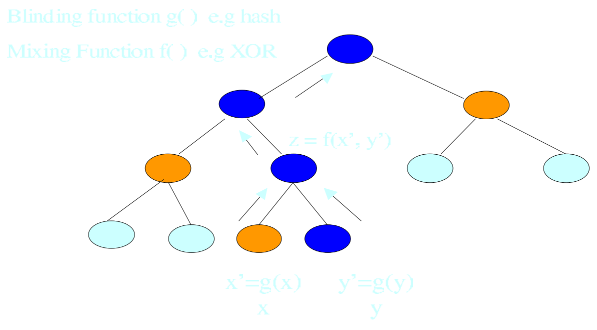
In this table we present the results of the evaluation the costs for protocols GDH.2 and GDH.3

Parameters	GDH.2	GDH.3
GSC Storage	(n+1)K	(n+2) K
Member Storage	K	K
Initial GSC computation	n C_E	n C_E
Initial member computation	n/2 C_E	4 C_E for i<(n-1), 2 C_E for M_{n-1}
Initial Communication	$(n^2+3n-2)K/2$	$(n^2+3n-4)K/2$
Add GSC computation	(n+1) C_E	(n+1) C_E
Add members computation	C_E	C_E
Add Communication	2(n+1) K	2(n+1) K
Delete GSC computation	(n-1) C_E	(n-1) C_E
Delete member computation	C_E	C_E
Delete communication	(n-1) K	(n-1) K

We can see that the two protocols are almost identical in performance. GDH.3 is more efficient than GDH.2 though because it requires much less member computation and the total number of exponentiations is less as well. This makes GDH.3 a fast, powerful protocol that could be used in the two level hierarchical scheme, in the place of GKMP protocol.

One Way Function Tree Scheme (OFT)

Last but not least, we are going to discuss a protocol for establishing a shared cryptographic key in large, dynamically changing groups. It



belongs to the same family of protocols (non-contributory) with the simple Tree Scheme. It achieves a new minimum in the number of bits that need to be broadcast to members in order to re-key after a member is added or evicted.

The number of keys stored by group members the number of keys broadcast to group when new members are added or evicted, and the computational efforts of group members, are logarithmic in the lumber of group members. The algorithm provides complete forward and backward secrecy.

The new group keying method uses one-way functions to compute a tree of keys, which is called the One Way Function Tree (OFT). The keys are computed up the tree, from the leaves to the root, the group manager maintains a binary tree, each node x of which is associated with two cryptographic keys, a node key k_x and a blinded node key $k_x' = g(k_x)$. The blinded node key is computed from the node key using a one way function g; it is blinded in the sense that a computationally limited adversary can know k_x' and yet cannot find k_x. The manager uses a symmetric encryption function E to communicate securely with subsets of group members.

Interior nodes of the tree have exactly two leaves. Every leaf of the tree is associated with a group member. The manager assigns and securely communicates a randomly chosen key to each member. The interior node keys are defined by the rule: $K_x = f(g(k_{left(x)}), g(k_{right(x)}))$, where left(x) and right(x) denote the left and the right children of the node x. The function f is a mixing function. The node key associated with the root of the tree is the group key, which the group can use to communicate with privacy and/or authentication. The security of the system depends on the following invariant: *Each member knows and maintains a list of the un-blinded node key on the path from its node to the root, and the blinded node keys that are siblings to its path to the root, and no other blinded or un-blinded keys.* This enables the member to compute the un-blinded keys along its path to the root, including the root key, which it also stores. If one of the blinded node keys changes and the member is told the new value, it can re-compute the keys on the path and find the new group key. The security of the group results primarily from the mixing function f. The addition of the one-way function g to blind internal keys gains an important functionality: each internal node key can be used as a communications subgroup key for the subgroup of all descendent members. This functionality is important to the efficiency of add/evict operations.

Adding or Evicting a Member

If the blinded key changes then its new value must be communicated to all of the members that store it. These members are associated with the descendants of the sibling s of x, and they know the un-blinded node key k_s. The manager encrypts k_x' with k_s before broadcast it to the group, provides the new value of the blinded key to the appropriate set of members, and keeps it from other members.

When a new member joins the group, an existing leaf node x is split, the member associated with x is now associated with left(x), and the new member is associated with right(x). Both members are given new keys. The old member gets a new key because its former sibling member knows its old blinded node key and could use this information in collusion with another group member to find an un-blinded key that is not on its path to the root. The new values of the blinded node keys that have changed are broadcast securely to the appropriate subgroups, as described above. In addition, the new member is given its set of blinded node keys, in a unicast transmission. In order to keep the height h of the tree as low as possible, the leaf closest to the root is split when the new member is added.

When the member associated with the node y is evicted from the group the member assigned to the sibling of y is reassigned to the parent p of y and given a new leaf key value. If the sibling s of y is the root of the sub-tree, then p becomes s, moving the sub-tree closer to the root, and one of the leaves of this sub-tree is given a new key (so that the evictee no longer knows the blinded key associated with the root of the sub-tree). The new values of the blinded node keys that have changed are broadcast securely to the appropriate subgroups, as described above. The number of keys that must be broadcast is equal to the distance from y to the root. During eviction, the evicted slot is pruned and a leaf node along the resulting path is changed. These changes propagate up the tree. A broadcast of h keys is required to update all the other members who had depended on the blinded node keys which have changed. In the addition case, unlike evict, we also have to unicast to the new member the blinded ancestral sibling keys the member needs to know, so additional h keys are also required.

LKH and OFT require about twice the size of broadcast to initialize than SKDC, since every key in a binary tree with n leaves must be communicated. Storage requirement of both LKH and OFT members are about $\log_2 n$ keys, while in SKDC members is exactly two keys. The storage requirement of LKH and OFT managers is about 2n, while that of SKDC managers is exactly n+1. Furthermore, OFT requires significantly fewer random bits than LKH. In particular, to add one member, in OFT the manager must generate only one new random key. By contrast in LKH the manager must generate h new keys. If key generation is performed in software, this difference could yield OFT a significant time advantage over LKH. The LKH and OFT distribute the computational cost of re-keying among the whole group, so that the group manager's burden is comparable to that of a group member.

Initialization	GKMP	LKH	OFT	Add Member	GKMP	LKH	OFT
Total Delay	n	2n	3n	Total Delay	n	2h	3h
#broadcast bits	n K	2nK+ h	2nK+ h	#broadcast bits	n K	2hK+h	hK+h
# unicast bits	0	0	0	# unicast bits	0	0	hK
Manager Comp/tion	n	2n	3n	Manager Comp/tion	n	2h	3h
Maxmember Comp/tion	1	h	2h	Maxmem Comp/tion	1	h	2h
# random bits gen.	K	2nK	nK	#random bits gen	K	hK	K

Evict Member	GKMP	LKH	OFT	Memory Usage	GKMP	LKH	OFT
Total Delay	n	2h	3h	Manager Storage	nK	2nK	2nK
# broadcast bits	n K+lgn	2hK+h	hK+h	MaxMemStorage	2K	hK	hK
# unicast bits	0	0	0				
Manager Computation	n	2h	3h				
Max. Member Comp/tion	1	h	2h				
# random bits gen.	K	hK	K				

Time and comm/tion usage of initialization, add-member, and evict-member, with GKMP, LKH, and OFT key-establishment methods. N: group size, K: size of a key in bits, h: height of the key tree (h = lg n when tree is balanced). While LKH has lower magnitude of total delay, the units of time for OFT are typically much smaller because keyless one-way functions are much faster than are keyed-encryption functions.

We model the secure channel by sending a secret key using the Public Key Encryption Method. Thus, the GSC encrypts the key with the Public RSA Key of the member, and the member decrypts the message with its private RSA key and derives the key intended to it.

GSC Storage Cost: 2nK if we don't assume a secure channel for the initial transmission of keys to the member. We need nK bits for storing the un-blinded keys and nK for storing the blinded keys of the members. If we use the public encryption method as secure channel, then we transmit the member un-blinded keys via symmetric encryption, and thus the GSC has to store the n keys used for the symmetric encryption as well, and this makes the total storage cost 3nK.

Member Storage Cost: Every member maintains the un-blinded key of the leaf it is associated with and a list of blinded node keys for all the siblings of the nodes along the path from its node to the root, including the root key, which it also stores. So, every member stores hK bits or h keys. So, a member decrypts its private key sent via the Public encryption method, and then decrypts the blinded

keys of the nodes whose parent or sibling nodes belong to the members' path up to the root via the symmetric encryption method.
Initial Communication: 2nK or 3nK depending on whether we incorporate the secure way of sending the initial un-blinded keys to members or not.
GSC/Member Add Computation: The new un-blinded key is sent to the member via the public encryption method, so the GSC encrypts it with cost C_{PE}. Then the GSC encrypts via the symmetric encryption method C_{SE} the h blinded keys that it is going to unicast to the new member and the h blinded keys that it is going to broadcast to the nodes that need these updated values. We have to note here that in the case that there is not a free leaf for a member to be virtually placed, we split an existing leaf x. The member associated with x becomes now left(x) and the new member becomes right(x). Both members are given new keys. So, in this case the GSC has to perform an additional C_r and an additional C_{PE}. However, if we decide that the tree will have a particular height and the number of members will never become more than a particular number n' then we can keep the previous formulation, otherwise the cost for a GSC will become: $2C_r + 2C_{PE} + h\ (C_{SE}+2C_g)$.
The member decrypts via the public key encryption method its un-blinded key and then via the symmetric encryption method the h un-blinded keys that the GSC sent to it. And then the blinded value of each such updated un-blinded key is calculated, with total cost $h\ C_g$.
Communication Cost for Addition: K for the un-blinded key sent to the new member, hK for unicast the h blinded keys to member and hK for broadcasting those h un-blinded keys to all the internal nodes that need their updated value.
GSC/Member Delete Computation: During eviction, the evicted slot is pruned and a leaf node along the resulting path is changed. The sibling member of the evicted one has its un-blinded key changed. These changes propagate up the tree. A broadcast of h keys is required to update all the other members who had depended on the blinded node keys that have changed. The new values of the blinded node keys that have changed are broadcast securely to the appropriate subgroups. So, after a member's eviction, the GSC creates a new key for the sibling of the evicted member with cost C_r, it encrypts the key in order to send it to the member using the public key encryption method and then it first calculates the updated blinded keys and after encrypts those h updated blinded keys it is going to broadcast to the appropriate subgroups. The encryption this time is performed via the symmetric encryption method. Now, we don't need to perform any unicast.
The member decrypts the new un-blinded key sent to it (this operation refers to the sibling of the evicted member) using the public key decryption method. Then it has to decrypt the h updated keys broadcast from the GSC with the symmetric decryption method. And then the blinded value of each such updated un-blinded key is calculated, with total cost hC_g.
Communication Cost for Deletion: K bits for the key sent via the Public key encryption method, and h K bits for the updated keys that are broadcast from the GSC to the members.

We analyzed the protocol OFT that belongs to the *non-contributory* family of protocols as the Logical Tree Scheme does but with additional features. It seems to be the most efficient among other protocols of either the contributory or the non-contributory family, and it would be very useful to incorporate this protocol in our two-level hierarchical scheme. This protocol could replace the Logical Trees either between GSC-GSAs or between GSA-Members.

ELK (Efficient Large-Group Key Distribution Protocol)

Due to the continuous increase in computation power, ELK has been designed to trade off computation for lower communication overhead. A member join protocol is designed that does not require any broadcast but requires that the server (or GSC in our case) computes a one-way function on all keys in each time interval. ELK also introduces hints, a technique which makes key updates smaller but requires receiver computation. A family of *pseudo-random functions* (PRFs) is defined

that uses key K on input M of length m bits and outputs n bits. $PRF^{(m->n)}$: K x $\{0,1\}^m$ -> $\{0,1\}^n$. We write $PRF_K^{(m->n)}(M)$. The following keys are derived from K_i as follows:
$K_i^\alpha = PRF_{K_i}^{(n->n)}, \{C_R\}_{K_L}$ $K_i^\beta = PRF_{K_i}^{(n->n)}$, $K_i^\gamma = PRF_{K_i}^{(n->n)}$, $K_i^\delta = PRF_{K_i}^{(n->n)}$.
In ELK, the key update is added directly to the data packets. Since space in data packets is limited, a small amount of key update information is added. Previous protocols had lengthy messages, so they could not use this approach. So, ELK features a method to compress key updates by trading off key update message size and receiver computation. The resulting key is small enough that the sender can piggyback in it data packets. *With those mechanisms, the majority of group members can recover from a lost key update if they receive the hints in a data packet.* The remaining small fraction needs to contact the key server through unicast. ELK is composed of two mechanisms: *key update, key recovery (through hints).* In this new key update protocol the left and right child keys contribute to update the parent key. The approach is similar to the OFT protocol, but here small hints allow legitimate members to reconstruct the key without the key update.

Assume that we want to update a key K of a node with children nodes whose keys are K_L and K_R. The new key K' is derived from K and contributions from both children. The left key K_L contributes n_l bits to the new key, which are derived by a pseudo-random function, using key K_L and applied to K. The contribution is symbolized $C_L = PRF_{K_L^\alpha}^{(n->n_1)}(K)$. Similarly, C_R is n_2 bits long and is derived from the right child key K_R and K as follows: $C_R = PRF_{K_R^\alpha}^{(n->n_2)}(K)$. We concatenate the two contributions to form a new key of length: n_l+n_2: $C_{LR}=C_L|C_R$. To compute K' we compute a pseudo-random function with C_{LR} as the key and the previous key K as the data: $K' = PRF_{C_{LR}}(K)$. Assume $n_l \leq n_2$. Since the security of the key distribution scheme is at most $O(2^n)$, then $n_l+n_2 \leq n$. The key update message needs to contain enough information so that the members on the left who know K_L can re-compute K' as well as the members on the right who know K_R .The left members can derive C_L themselves but they need C_R. Hence the key update message contains $\{C_R\}_{K_L}$ (n_2 bits long). Similarly, for the members on the right, the key update message contains $\{C_L\}_{K_R}$ (n_l bits long).

Instead of broadcasting the key update message that has length n_l+n_2 bits, legitimate members who know K and either K_L or K_R can also recover the new key K' from a hint that is smaller than the key update message, by *trading off computation for communication.* If we assume that a member can perform 2^x computations, we can construct a smaller key update that we call a hint. Consider the right-hand members that know K and K_R. They can derive the contribution C_L. If they would also have a checksum, they could brute-force the missing n_l bits of K' from the left side contribution. The hint message contains the key verification $V_{K'}$, which is derived from the new key $V_{K'} = PRF_{K'}(0)$ (n_3 bits long). The right-hand members compute the following keys: for each possibility for C_L they compute C_{LR}' and obtain a candidate key $K'' = PRF_{C_{LR}}(K)$. The member verifies the candidate key by checking against the key verification $PRF_{K''}(0)$ to see if it equals $V_{K'}$. $PRF_{K''}(0)$.

Setting $n_3 = n_l+1$ in the hint results in "half a false positive key" on average. The problem is that the procedure might deliver more than one candidate key, in which case some of them are false positives. So, we might expect to receive more than one false positive key next to the correct key. The advantage of the hint is that it is $2n_l-n_3$ bits shorter than the key update. In the ideal case $n_l = n_2$ and then the hint is determined by the security parameter.

The Initial Costs as well as the GSC/Member Storage Costs of ELK protocol are identical to the OFT with the exception that in ELK no blinding function g exists. For the Initial computation the GSC creates keys for all nodes of the tree, using the RSA key generation function: C_r.

GSC/Member Add Computation: For the case of a member addition no broadcasts are necessary. Initially, after selecting the location for the new member, the GSC communicates to it first its secret key with the Public Decryption method and then the keys of its path up to the root encrypted with the symmetric encryption method with the key that was just communicated to it. Until here, the case is similar to OFT addition. However, no update key messages are communicated to the rest of members. During member addition or key refreshment at given intervals, GSC updates the keys of all nodes in the tree. It computes K_i'= $PRF_{K_i}(K_G)$ for all keys K_i, and K_G'= $PRF_{K_G}(0)$ for the new group key. Each node with key K_i knows the current group key K_G also, computes the new group key and its own updated key K_i'. Thus each member computes the (h+1) updated keys of its path, with cost (h+1) C_{PRF}. The GSC updates all 2n keys, apart from encrypting the appropriate keys for the new member.

Add Communication: The GSC communicates the appropriate keys to the new member only. As we have seen before, it communicates (h+1) keys to the new member.

GSC Delete Computation: The GSC computes all the updated keys (h) in the path of the evicted member: K_i'= $PRF_{C_{LR}}(K_i)$ where $C_{LR} = PRF^{(n->n_1)}_{K^{\alpha}_{il}}(K_i) | PRF^{(n->n_2)}_{K^{\alpha}_{ir}}(K_i)$. Then, the updated message the GSC broadcasts contains $\{C_R\}_{K^B_L}$, and $\{C_L\}_{K^B_R}$, namely the encrypted right and left contributions for each updated key, 2h contributions in total. {*Only members that belong to the path of node associated with key K_L or K_R can derive $K_L{}^B$ or $K_R{}^B$ respectively and thus decrypt the C_R or C_L respectively they need. Since $K_L{}^B$=$PRF_{KL}(K_i)$ and $K_R{}^B$=$PRF_{KR}(K_i)$, the GSC derives $K_L{}^B$ from K_L with cost C_{PRF}(K/2) etc. So does the member*}. The GSC computes 2h contributions, thus C_{PRF} (K/2) bits in total. Before that, it has to compute K^A_R K^A_L, K^B_R, K^A_R. Again, the cost is 4h C_{PRF}(K/2). Along with that it performs 2h encryptions with the symmetric encryption method (DES), so we have 2h C_{SE}(K/2). The GSC attaches the hint message along with the data packets. So, we omit the cost of the hint message from the computations the GSC has to do in order to appropriately update the tree.

Delete Member Computation: *Note that the tree is a virtual structure. In fact, the member receives the keys that correspond to the internal nodes that reside in its path up to the root. However the member behaves as if those keys actually belong to the internal nodes throughout all its communication transactions. So, a member gets the key updates of the internal nodes in its path and decrypts them with the secret key decryption method.* Each node, depending on whether it is a left or right child, has to decrypt the right or left key update respectively in order to derive the key of its parent. The GSC computes all the keys that need to be updated. Instead of unicasting the whole keys that correspond to each internal node, it broadcasts part of those keys. For instance, members get $\{C_R\}_{K^B_L}$. Those members who know K_L (the node associated with partial key K_L lies on their path) can derive $K_L{}^b$, decrypt $PRF^{(n->n_2)}_{K^a_R}(K)$ from the key update and derive K'.

The computation cost for deriving K^B_L or K^B_R is C_{PRF}(K/2) in average. Each member must compute for each node in its path that is updated either K^B_L and K^A_L or K^B_R and K^A_R depending on whether the member is found in the left sub-tree of this node or in the right respectively. This computation results in 2 C_{PRF}(K/2). Then, each member that can derive K^A_L of a particular updated node, can then derive C_L with cost C_{PRF}(K/2) as well. Similar is the case for a member that knows key K^A_R of an updated node. Now we are going to see how many updated nodes lie in the path of each member up to the root. Actually n/2 members have 1 updated node, n/4 member2 have 2,…, n/2^h members have h. In average, each member has 2 updated nodes in its path up to the root. Apart from the previous costs, members must brute force the part of the key they do not acquire and verify it with the update they

receive and decrypt from the GSC. For this heavy computation we define a new variable C_{KEY}. In average: $C_{KEY} = 2^{K/2-1}C_{PRF}(K/2)$. So, the cost of C_{KEY} for the calculation of the contribution using the brute force method is additionally required, as well as the cost for verification which is $C_{PRF}(L)$, where $L<K/2$. Average Delete Member Computation: $2(3C_{PRF}(K/2)+C_{PRF}(L)+C_{SD}(K/2)+C_{KEY}(K/2))$. Max. Delete Member Computation: $h(3C_{PRF}(K/2)+C_{PRF}(L)+C_{SD}(K/2)+C_{KEY}(K/2))$.
Delete Communication: The GSC need only broadcast those 2h key updates. So, the delete communication is: $2h\times K/2$ bits = hK bits.

Parameters	ELK	OFT
GSC Storage	2n K	2n K/ 3nK
Member Storage	(h+1) K	(h+1) K
Initial GSC computation	$2(n-1) C_r + n C_{PE} + 2(n-1) C_{SE}$	$nC_r + n C_{PE} + n(C_{SE}+2C_g)$
Initial member computation	$C_{PD} + hC_{SD}$	$C_{PD} + h C_{SD} + h C_g$
Initial Communication	2nK / 3 n K	2n K / 3nK
Add GSC computation	$(h+1)C_r + C_{PE} + 2 h C_{SE} + 2nC_{PRF}$	$C_r + C_{PE} + h(C_{SE}+2C_g)$ / $2C_r + 2 C_{PE} + h(C_{SE}+2C_g)$
Add member computation	$C_{PD} + hC_{SD}$ for the new $(h+1)C_{PRF}$ for the rest	$C_{PD} + hC_{SD} + hC_g$, for two $hC_{SD} + hC_g$, rest
Add Communication	(h+1)K	(2h + 1) K
Delete GSC computation	$6hC_{PRF}(K/2) + 2hC_{SE}(K/2)$	$C_r + C_{PE} + h(C_{SE} + 2C_g)$
Delete member computation	$2(3C_{PRF}(K/2)+C_{PRF}(L)+C_{SD}(K/2)+C_{KEY}(K/2))$, $L>K/2$	$C_{PD}+hC_{SD}+hC_g$, for one $hC_{SD} + hC_g$, rest
Delete communication	h K	(h+1) K

Evaluation of C_{PRF}, C_{KEY}

A PRF function can be modeled with any encryption mechanism that passes the next–bit test and fulfills certain security requirements depending on the application. The authors have decided to model the PRF with RC5 encryption mechanism. RC5 is a very simple mechanism, invented by Ronald Rivest as a diversion of RSA. It is over four times faster than Rinjdael and much simpler. In the paper of Ronald Rivest it is suggested that RC5 can replace DES with an equivalent algorithm if we designate properly the input parameters of RC5: *word size in bits (w), number of rounds r (the expanded key table S contains t=2(r+1) such words), the number of bytes in the secret key K (b), and the secret key* of course. It is suggested that in applications such as key management choosing r=32 rounds is appropriate to ensure security and not to slow down applications. RC5 uses the following three primitive operations only and their inverses (all are linear but the variable rotations x<<y which take constant time): additions, subtractions, bit-wise exclusive-OR, left and right rotations.

The algorithm consists of three components: a key expansion algorithm, an encryption algorithm and a decryption algorithm. From the inspection of the three procedures we yield the following results for the complexity of RC5: We assume that the expanded key table S consists of two w-bit words.
The complexity of encryption approximately is: $r\times2\times(2\times w)=4\times r\times w$. Similarly, the complexity of decryption is: $r\times2\times(3\times w)=6\times r\times w$.

The key-expansion routine expands the user's key K to fill the expanded key array S, so that S resembles an array of $t=2\times(r+1)$ random binary words determined by K. The key expansion algorithm uses two magic constants, and consists of three algorithmic parts: copying the secret key into an array L of $8\times b/w$ words (b/w complexity), initializing array S to a particular fixed (key-independent) pseudo-random bit pattern, using an arithmetic progression modulo 2^w determined by

the magic constants (this has $t \times w$ complexity), mixing the user's secret key in three passes over the arrays S and L ($3 \times t \times 6 \times w = 18 \times t \times w$).
So, if we decide to use a key of length K, it consists of K/8 words. Assuming that we set w=64 for better security and $t=16 \times (K/8)$ (typical is that t=6.5(K/8)) then we have the following: r =K-1. So we can claim that r may have the size of K. We have to note here that r can also have the size of half of the length of the key (K/2) and still produce a secure scheme. Summing up the overall complexity for r=K and w=64 is: $10 \times r \times w + K/64 + 19 \times t \times w = 48 \times r \times w + K/64 + 38 = 48 \times 64 \times K + K/64 + 38 > 3072 \times K$ in the worst case. For w=32 and r=K/2 complexity is: $768 \times K$ and the protocol still performs very well. The important conclusion is that the operation is linear to the size of the key and thus quite chip. Generally however, the larger the r, the more secure the protocol is.

The C_{KEY} function as we stated in the analysis of ELK is a heavy computational function that has complexity of: $2^{K/2-1} C_{PRF}(K/2)$. We need to generate a new key (we assume we can use the PRF for that, or just start using exhaustively all combinations without any preferences and orientations). The testing of the new key is a straight-forward procedure, since it is a verification function, its complexity is O(K), where K is the key length. We assume that the verification can be done mainly with the aid of a single exclusive-OR operation. So for PRF the complexity of C_{KEY} function is: $2^{K/2-1} C_{PRF}$' else it is simply $2^{K/2-1}$.

We sum our results in the table that follows:

$C_{PRF}(K)$	CK, $800<C<3000$
C_{KEY}	$2^{K/2-1} C_{PRF}(K/2)$

Here we present the costs for all 4 protocols that we have discussed in more detail and we make a comparison of all their costs to address the advantages and disadvantages of each over the rest.

Parameters	ELK	OFT
GSC Storage	2n K/ 3nK	2n K / 3nK
Member Storage	(h+1) K	(h+1) K
Initial GSC computation	$2(n-1) C_r + n C_{PE} + 2(n-1) C_{SE}$	$nC_r + nC_{PE} + n (C_{SE}+2C_g)$
Initial member computation	$C_{PD} + hC_{SD}$	$C_{PD} + h C_{SD} + h C_g$
Initial Communication	2nK / 3 n K	2n K / 3nK
Add GSC computation	$(h+1) C_r + C_{PE} + 2 h C_{SE} + 2n C_{PRF}$	$C_r + C_{PE} + h(C_{SE}+2C_g)$ / $2C_r + 2C_{PE} + h(C_{SE}+2C_g)$
Add member computation	$C_{PD} + hC_{SD}$ for the new, $(h+1)C_{PRF}$ for the rest	$C_{PD} + hC_{SD} + hC_g$, for two $hC_{SD} + hC_g$ for the rest
Add Communication	(h+1)K	(2h + 1) K
Delete GSC computation	$6hC_{PRF}(K/2) + 2hC_{SE}(K/2)$	$C_r + C_{PE} + h(C_{SE} + 2C_g)$
Delete member computation	$2(3C_{PRF}(K/2)+C_{PRF}(L)+C_{SD}(K/2)+C_{KEY})$, $L<K/2$	$C_{PD} + hC_{SD} + hC_g$ for one $hC_{SD} + hC_g$ for the rest
Delete communication	h K	(h+1) K

Parameters	GKMP	CBT
GSC Storage	n K	(dn-1) K (d-1)
Member Storage	2 K	(h+1) K
Initial GSC computation	$(n+1) C_r + n C_{PE} + nC_{SE}$	$(dn-1) C_r /(d-1) + n C_{PE} + d (n-1) C_{SE}/(d-1)$
Initial member computation	$C_{PD} + C_{SD}$	$C_{PD} + h C_{SD}$

Initial Communication	2 n K	(n +d(n-1)/(d-1)) K
Add GSC computation	2 C_r + C_{PE} +2 C_{SE}	(h+1) C_r + C_{PE} + 2 h C_{SE}
Add members computation	C_{PD} + C_{SD} for the new C_{SD}, rest	C_{PD} + h C_{SD}, for one h C_{SD}, rest
Add Communication	3 K	(2h + 1) K
Delete GSC computation	C_r + (n-1) C_{SE}	h C_r + d h C_{SE}
Delete member computation	C_{SD}	h C_{SD}
Delete communication	(n-1) K	(d-1) h K

Comparison of GKMP, CBT, OFT, ELK protocols

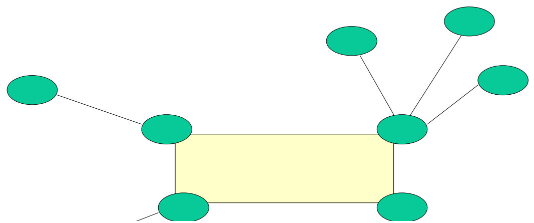
The broadcast size to initialize the GKMP is n. CBT, ELK and OFT require about twice that size of broadcast since every key in a binary tree with n leaves must be communicated. The Storage requirement of both CBT and OFT members are about $\log_2 n$ keys, while that of GKMP members is exactly 2 keys. The storage requirements of the GSC of CBT, ELK and OFT are 2n whereas that of GKMP is n.

Another advantage of OFT over CBT and ELK is that OFT requires significantly fewer random bits. In particular to add one member, in OFT the GSC must generate only one new random key. By contrast in CBT the GSC must generate h new keys, and in ELK 2n new keys. If key generation is performed in software, this difference could yield OFT a significant time advantage over CBT and ELK since in many practical applications entropy is a precious resource. In ELK the key generation algorithm is much cheaper than the key generation algorithm we use in OFT and CBT, but even so, the overhead for generating 2n keys for every member addition is to high, and particularly if we want to implement ELK in a MANET where bandwidth and nodes resources might be significantly limited.

A disadvantage of OFT over ELK and CBT is that in OFT the member computation for the addition of a member is heavier than this of the other protocols, since the member has to compute the blinded values of the un-blinded keys sent to it. However, we have just shown that the C_g computation is not as expensive as C_r and C_{PE}, so this is not a significant difference in performance. GKMP of course is the protocol that requires the least member computation, but the maximum GSC computation cost and has the maximum communication overhead. The add communication is about the same for OFT, ELK and CBT and it depends on the height of the tree, but for GKMP it is fixed to 3K, regardless of the group size.

For the GSC computation in the case of member deletion all protocols provide different mechanisms. ELK is more efficient in the sense that it applies chip pseudorandom and cryptographic functions to half the length of the keys, so it substantially reduces the GSC cost for the deletion. As we have already mentioned the pseudorandom functions PRFs for this application can be modeled with RC5, which is designed by Ronald-Rivest to be a much simpler and faster diversion of RSA. OFT is more efficient than CBT in the calculation of this cost as CBT must generate more new keys than OFT. Furthermore, OFT needs to evaluate the C_g function for the keys but it is not as costly as the C_r. The most expensive operation of all in terms of GSC computation for member deletion is this of GKMP. It has to encrypt a key for all n-1 members. For the delete member computation the most efficient is that of GKMP. CBT comes next, then OFT and the less efficient is the member of ELK which has a bunch of computations to perform. Last but not least is the Communication Cost in the case of member deletion. ELK and OFT are the winners, they behave almost the same while CBT produces double communication overhead. Clearly the looser in this case is GKMP with a considerable overhead, a major disadvantage for application of GKMP in a MANET environment.

What is important here is to decide which cost is the one we mostly want to reduce: storage cost, computation cost or communication cost. It seems that communication cost is the one we have to cater for the most.



Octopus Protocol (DH Based)

As we already know, contributory key distribution protocols interactively generate a common group key including an independent contribution of every group member. Each member of the group has a piece of information that has to reach every other member of the group. Therefore, the minimum number of exchanges or messages needed to distribute this information may serve as a lower bound for the number of exchanges or messages needed to distribute this information may serve as a lower bound for the number of exchanges or messages required by a contributory key distribution. In literature, Baker and Shostak have shown that the minimum number of required phone calls needed to distribute n pieces of information held by n different parties (*gossip problem*) is 2n-4, n $\geq$4. This result provides us with a lower bound for the total number of exchanges required by contributory key distribution protocols that do not activate broadcasts. Klaus Becker and Uta Wille introduce a DH-based protocol that matches the previous bound.

It has been proven that if h(n) denotes the minimum number of messages required by the gossip problem among n parties, then it holds that h(n)=2n-2.

For a contributory group key distribution system P for n parties without broadcasts the following have been proven: For the total number of messages $\phi_1(P)$ and exchanges$\phi_2(P)$, required by P it holds that $\phi_1(P) \geq$2n-2, $\phi_2(P) \geq$2n-4.

For a contributory group key distribution system P for n parties with broadcasts the following have been proven: For the total number of messages $\psi_1(P)$ and exchanges $\psi_2(P)$ required by P it holds that $\psi_1(P) \geq$n and $\psi_2(P) \geq$n.

It has been shown that the Octopus protocol requires only 2n-4 exchanges. For the broadcasting case, no further protocol has to be introduced because the lower bounds for the number of exchanges and the number of messages are both n in GDH.2. So, GDH.2 proves that the lower bound for the number of exchanges is sharp if broadcasting is possible.

The Octopus protocol works as follows:

It uses DH key computed in one round as a random input for the subsequent round. Therefore, it is further assumed that there is a bijection WRITE HERE PHI !!!!!!!!!!!!!!!from generator G into Z_q (field from which the participants choose their random secrets).

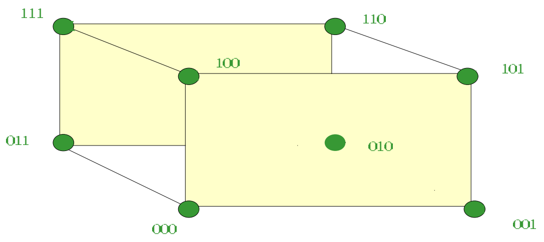
Four parties A, B, C, D may generate a group key using only four exchanges. First parties A and B and then parties C and D perform a DH key exchange generating keys α^{ab} and α^{cd}, respectively. Subsequently, A and C as well as B and D carry out a DH key exchange using as secret values the keys generated in the first step; i.e A(B) sends $\alpha^{\phi(a^{ab})}$ to C(to D) while C(D) sends $\alpha^{\phi(a^{cd})}$ to A(B) such that A and C (B and D) can generate the joint key $\alpha^{\phi(a^{cd})\phi(a^{ab})}$. In the Octopus protocol participants P_1, P_2, …, P_n generate a common group key by first dividing themselves into five groups. Four participants P_{n-3}, P_{n-2}, P_{n-1}, P_n take charge of the central control; denote these participants A, B, C, D respectively. The remaining parties distribute themselves into four groups: $\{P_i \mid i \in I_A\}$, $\{P_i \mid i \in I_B\}$, $\{P_i \mid i \in I_C\}$, $\{P_i \mid i \in I_D\}$, where I_A, I_B, I_C, I_D are pair-wise disjoint, possibly of equal size, and $I_A \cup I_B\, a^{\phi(a^{K(I_C \cup I_D)})\phi(a^{K(I_A \cup I_B)})} I_C \cup I_D = \{1,\ldots n-4\}$. P_1, …P_n can generate a group key in three steps as follows:

1. For all $X \in \{A, B, C, D\}$ and all $i \in I_X$, the party X generates a joint key k_i with P_i by performing a DH key exchange.
2. The participants A, B, C, D perform the 4-party key exchange described above using the values $a = K(I_A)$, $b = K(I_B)$, $c = K(I_C)$, $d = K(I_D)$, where $K(J) := \prod_{i \in J} \phi(k_i)$ for $J \subseteq \{1, \ldots, n-4\}$. Thereafter, A, B, C, D hold the joint and later group key K:= $a^{\phi(a^{K(I_A \cup I_B)})\phi(a^{K(I_C \cup I_D)})}$.
3. The step is described only for A. The parties B, C, D act correspondingly. For all $j \in I_A$, the participant A sends the following two values to P_j: $a^{K(I_B \cup I_A \setminus \{j\})}$ and $a^{\phi(a^{K(I_C \cup I_D)})}$. Now P_j is able to generate K; first P_j calculates $(a^{K(I_B \cup I_A \setminus \{j\})})^{\phi(k_j)} = a^{K(I_A \cup I_B)}$ and then $K = a^{\phi(a^{K(I_C \cup I_D)})\phi(a^{K(I_A \cup I_B)})}$.

This protocol requires n-4 exchanges to generate the DH keys k_i, four exchanges for the key agreement between A, B, C, D and finally n-4 messages to be sent from A, B, C, D to $P_1, P_2, \ldots, P_{n-4}$. Hence the protocol performs a minimum number of 2n-4 exchanges.

protocol	messages	exchanges	simple rounds	Synchr. rounds	broadcasts
Octopus	3n-4	2n-4	$2\left\lceil \frac{n-2^d}{2^d} \right\rceil \left\lceil \frac{n-4}{4} \right\rceil + 2$	4	-

Cost of parameters and operations in Octopus protocol



Cube and 2^d –Octopus protocol.

The number of simple rounds can be minimized by generalizing the idea of the 4-party key agreement described above. In general 2^d parties can agree upon a key within d simple rounds by performing DH key exchanges on the edges of a d-dimensional cube. In order to describe formally the cube protocol for 2^d participants, we identify the 2^d participants of the d-dimensional space $GF(2)^d$ and choose a basis $b_1, \ldots, b_d$ of $GF(2)^d$. Now the protocol may be performed in two rounds as follows:

1. In the first round, every participant $v \in GF(2)^d$ generates a random number r_v and performs a DH key exchange with participant $v+b_1$ using the values r_v and r_{v+b1}, respectively,
2. In the i^{th} round, every participant $v \in GF(2)^d$ performs a DH key exchange with participant $v+b_i$, where both parties use the value generated in round i-1 as the secret value for the key exchange.

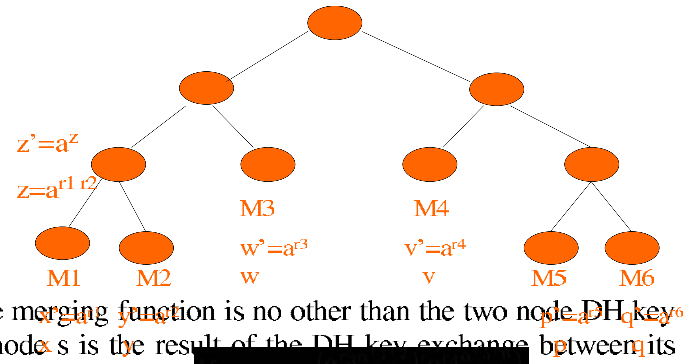
In every round, the participants communicate on a maximum number of parallel edges of the d-dimensional cube (in round i in the direction b_i). Thus every party is involved in exactly one DH exchange per round. Furthermore, all parties share a common key at the end of this protocol because the vectors $b_1, \ldots, b_d$, form a basis of the vector space $GF(2)^d$.
In order to formulate a protocol for an arbitrary number of participants ($<>2^d$), which requires a low number of simple rounds, the idea of the octopus protocol can be adopted again. In the 2^d – octopus protocol the participants act as in the octopus protocol with the only difference that 2^d instead of four parties take charge of the central control, whereas the remaining n-2^d parties divide into 2^d groups. In other words, in steps 1 and 3 of the octopus protocol, 2^d participants manage communication with the rest and in step 2 these 2^d parties perform the cube protocol for 2^d participants. If the number of participants is n and if d is the largest integer smaller than $\log_2 n$, then the 2^d – octopus protocol requires 1+d+1 $= \lceil \log_2 n \rceil + 1$ simple rounds. In general, we obtain the following values for the complexity of the cube and of the 2^d–Octopus protocol.

Protocol	messages	exchanges	Simple rounds	Syn. rounds	bc

2^d - cube	n d	n d/2	d	d	-
2^d - octopus	$3(n-2^d) + 2^d$ d	$2(n-2^d) + 2^{d-1}$ d	$2\left\lceil {n-2^d}/{2^d} \right\rceil$+d	2+d	-

The 2^d – Octopus protocol is particularly interesting because it provides a tradeoff option between the total number of messages or exchanges needed and the number of rounds. For d=2 (octopus protocol) the number of exchanges is optimal, whereas the number of simple rounds is comparatively high. On the other hand, if d satisfies $2^{d-1} < n < 2^d$, the number of simple rounds required is very low and the total number of messages is high. Furthermore, the protocol enables the group to decide how many participants should share control of the protocol.
With the 2^d–Octopus protocol a class of key distribution systems without broadcasting which matches the lower bound $\lceil \log_2 n \rceil$ for the total number of simple rounds if n is a power of 2 was introduced; otherwise the protocols require $\lceil \log_2 n \rceil + 1$ simple rounds. With respect to the bounds derived for the number of rounds required by contributory distribution protocols, there have been formulated protocols that exceed the bounds by at most one round. It remains an open question whether there exist, for n<>2^d, protocols (w. or w/o broadcast) that require only $\lceil \log_2 n \rceil$simple rounds.

TGDH protocol



The TGDH protocol resembles OFT a lot in general. The basic differences are the following: any member of the tree can act as a central controller depending on its position in the tree, a member knows all blinded keys of the tree at any given time, and in TGDH the merging function is no other than the two node DH key exchange. The secret key x of an internal node s is the result of the DH key exchange between its offspring left(s) with associated secret key y and right(s) wi[redacted]hen $x = \alpha^{yz}$, and the blinded key of node s is α^x. We assume that any member at any time can become group leader and broadcast a message to all the members of the group. The mobility of nodes as well as their bandwidth and computational limitations might make it impossible for a simple node to broadcast a message to all members. In such an environment nodes may be close at one time interval and further at another. Therefore, this protocol is weak in MANETs, unless the network is rather restricted so that nodes stay relatively close to each other throughout the entire multicast session so the bandwidth problem is resolved. If we assume that we separate a large group of nodes in 2^d subgroups of small size, each of which perform key distribution under the TGDH protocol, we can claim that we overcome most of these difficulties imposed by the MANETs.

The original paper does not define how the tree is originally formulated from a number of nodes that wish to be members of the same group. We could assume that this is being done gradually, by executing the join operation for each new member that is successively incorporated into the tree. Instead, we *propose a more efficient and less costly scheme*. During the initial construction of the tree half of the members become sponsors. However, each time we go up a level in the tree towards the root, the number of sponsors is reduced to half because a parental node inherits both sponsors of its children but it needs only one, so the other becomes obsolete. At each level, the sponsors compute the un-blinded and blinded key of their parent, and broadcast the latter to the rest of the members. Each member computes all keys in its path to the root and also gets all the blinded keys of the tree.

Initial Member/Sponsor Storage: Since every member stores all blinded keys of the tree, it stores (2^h-1) K= (n-1) K bits for all the blinded keys, and h K for all the un-blinded keys in its path to the root.

Initial Member/Sponsor Computation: The member/sponsor does 2 exponentiations for every computation of a node key in its path. It receives the blinded key of its co-path, raises it to the power of its own secret key and blinds the new key that is derived (one exponentiation more) in order to broadcast it to the rest of the members. There are h nodes in its path, so it does 2h exponentiations.

Initial Member/Sponsor Communication: Every member or sponsor broadcasts all blinded keys of its path (h blinded keys), since all members need to have all blinded keys of the tree.

Add sponsor computation: The sponsor generates a new intermediate node and a new member node. The new intermediate node will be the parent of the sponsor and of the new member node. The sponsor gets the blinded key of the new member, and raises it to the power of its own secret key. The result is the secret key of the parent node. Now the sponsor has to blind this new key as well (second exponentiation). Similarly, all the updated nodes in the sponsor path from the leaf to the root are calculated. Thus, 2h exponentiations are done by the sponsor. Then the sponsor sends all blinded keys to the new member only, and just the updated blinded keys of the tree to the rest of the members. The sponsor calculates the new group key while updating the necessary blinded keys to send to the rest of the members of the group.

Add member computation: The new member, upon receiving all the blinded keys of the tree, does h exponentiations (using the blinded keys of its co-path) to calculate the group key. It needs to calculate first the secret keys of its path. All the rest of the members have to do from 1 to h-1 exponentiations to compute the group key, since not all blinded keys change for them, and since they have already calculated the secret keys of their co-path. It is as follows: n/2 members does 1, n/4 do two,…, n/2^h do h exponentiations. In average each member does 2 exponentiations.

Add communication: The new member sends its blinded key to its sibling (sponsor). Then, the sponsor broadcasts the updated h blinded keys to all the members in the tree except for the new, and additionally unicasts all the n-1blinded keys of the tree to the new member.

Delete sponsor computation: The sponsor now is the right-most leaf node of the sub-tree routed at the leaving member's sibling node. The former sibling of the leaving member is promoted to replace the leaving member's parent node. The sponsor picks a new secret share, computes all keys on its key path up to the root, and broadcasts the new set of blinded keys to the group. This information allows all members to re-compute the new group key. The rest is exactly as in the add sponsor computation case. Thus the sponsor has to do 2h exponentiations.

Delete member computation: The members compute the new group key after having received the updated blinded keys that the sponsor broadcasts. As in the case of member addition exactly, every member in average does 2 exponentiations.

Delete communication: Sponsor broadcasts the updated h blinded keys to all members of the tree.

Cost Evaluation for Original and Modified 2^d–Octopus protocols: (O), (MO), (MOT).

The n group members are divided into 2^d subgroups of equivalent size. Each sub-group has one leader or else group security controller (GSC), e.g. the sponsor(s) in TGDH. We now modify the initial 2^d–Octopus protocol by replacing the 1st step where partial group keys are derived by the two party DH exchange, with GDH.2 / TGDH partial group key derivation. Subsequently we modify the 2nd and 3rd step of the original 2^d-Octopus protocol to the new hybrid protocols requirements and we analyze their performance. We denote the original 2^d-Octopus protocol as (O), the GDH.2 modified 2^d-Octopus as (MO), and the TGDH modified as (MOT) for simplicity. The papers on 2^d–Octopus protocol do not refer to the case of member addition/deletion. We analyze this case as well and derive the cost values for the (O), the (MO), and the (MOT) protocols.

Step1: Each subgroup establishes their sub-group key, or handles member additions/evictions exactly as indicated by GDH.2 and TGDH. We use a pseudo-random number generator C_{rr}, to create the secret shares of the three contributory protocols. This pseudo-random number generator is less complicated than the one we use to create number for the RSA system, C_r. At the end of the protocol evaluation we describe how to derive the complexity of the pseudo-random number generators.

(MO): Message length for 2^d GSCs: $2^{d-1}(\lceil n/2^d \rceil^2 + 3\lceil n/2^d \rceil - 2)$ K. Average exponentiations per GSC for $\lceil n/2^d \rceil$ nodes: $\lceil n/2^d \rceil$. Average exponentiations per member: $(\lceil \frac{n}{2^{d+1}} \rceil + 1)$, according to GDH.2 formulas for the initial phase. In the add phase the GSC does: $(\lceil \frac{n}{2^d} \rceil + 1)$ exponentiations and the member does one exponentiation.

(O): During the DH key exchange among members and their GSCs, all n-2^d members send one message to their GSCs, and the GSCs broadcast to members 2^d messages in total. This is so because we assume that the GSC sends the same share a^x for members that belong to the same subgroup. Each member j of course creates its own secret share, λ_j, so that the secret key that each member j shares with its GSC is: $a^{x\lambda_j}$. So, we get n messages in total. The GSC does $2\lceil (n-2^d)/2^d \rceil$ exponentiations in the initial phase and the member does two exponentiations in the same phase.

(MOT): In the initial phase as we have designed it, the maximum number of messages per sponsor is $\log\lceil n/2^d \rceil$. For every node that lies in the sponsor's path, one broadcast message is sent (the node's blinded secret key is sent to the rest of members) and two exponentiations are done (one in order to construct the secret key of this node, and one to blind the secret key). In total, there is an exchange of $2\lceil n/2^d \rceil$ communication messages per subgroup, and 2n for all 2^d subgroups, as we have already seen. Also, in total 4n exponentiations are required for all GSCs at the initial phase. In the case of addition, the sponsor does $2\log\lceil \frac{n+1}{2^d} \rceil$ exponentiations and the member does two. The subgroup size in the addition case is: $\lceil \frac{n+1}{2^d} \rceil$, so $\lceil \frac{n+1}{2^d} \rceil$ messages are communicated. In the case of deletion the size of the subgroup is $\lceil \frac{n-1}{2^d} \rceil$, and $\log\lceil \frac{n-1}{2^d} \rceil$ messages are communicated, according to the TGDH formula.

*Observation*1*:* The group G that provides the generator α is of order q and has length K bits (the length of the group key). Thus, rising α to more exponents does not change the number of bits of the resulting value (exponentiations are modular). The complexity of each exponentiation is denoted as C_E.

*Observation*2*:* In the example for the simple Octopus protocol we used d=2. However, for d>2, the 3d step should be modified as follows: each of the 2^d participants (GSCs) must communicate d values to its group, in analogy to the two values that are derived in the case of the original Octopus (d=2). For instance, for d=3, we have eight GSCs noted as: A, B, C, D, E, F,G and H, and we expect the following sequence of operations:
In the 1st round the DH keys derived are: a^{AB}, a^{CD}, a^{EF}, a^{GH} .
In the 2nd round the DH keys derived are: $a^{a^{AB}a^{CD}}$, $a^{a^{AB}a^{CD}}$, $a^{a^{EF}a^{GH}}$,$a^{a^{EF}a^{GH}}$
In the 3d round the DH key derived is: $a^{a^{a^{AB}a^{CD}}a^{a^{EF}a^{GH}}}$ for all four pairs (this is the final group key).

For (O) protocol, if we take GSC A for instance, it has to communicate to its member j the following parts of the group key at the initial phase so that member j only securely derives the group key (according to the example for Octopus, d=2): a^{AB/K_j}, $a^{a^{CD}}$, $a^{a^{a^{EF}a^{GH}}}$. Thus, the GSC must communicate d=3 messages to each member of its subgroup. For (MO) and (MOT) protocols, for any member that belongs to the subgroup of GSC A for instance, the three parts of the group key that are sent by A are: a^{B}, $a^{a^{CD}}$, $a^{a^{a^{EF}a^{GH}}}$.

At the 3d step for the initial phase, members of all three protocols must execute d exponentiations each, to compute the final group key. This is so because each member gets d parts of the key. It raises the first part to the power of its own secret contribution and gets the first outcome. It then raises the second part to the first outcome, and gets the second outcome etc. It finally raises the dth part of the key to the (d-1)th outcome and gets the group key. It does d exponentiations in total.

For (MO) and (MOT) protocols, during the addition/eviction events, (d-1) of the d values need not be broadcast anew since they remain unchanged, provided that they are already stored in every member. So, the GSC communicates only one value to the whole sub-group, the value each particular member requires in order to reconstruct the group key, which is the same for every member in the subgroup. The GSC in the group of which a change of membership occurred need not communicate anything to its members at the 3d step. The subgroup has calculated its updated subgroup key already, and all members of the subgroup can use the remaining (d-1) values they are storing to reconstruct the new group key. We illustrate these functions by continuing with the previous example where d=3. Assume that a change of membership occurs in the subgroup of GSC A and the new subgroup key is now χ . In this case, the members may not need to do all the d exponentiations as they did for the initial case. Actually the members that belong to the sub-groups of 2^{d-1} participants (E, F, G, H in our example) need to do only one, those of 2^{d-2} participants need to do two,…., those of 2 participants (C, D in our example) have do d-1, and those of the last two participants (A, B in our example) have to do d exponentiations. In average, members of each participant need to do two exponentiations in order to reconstruct the new group key in the case of addition/deletion. In the addition/deletion case where the subgroup key of GSC A is substituted we expect the following operations during the 2nd step:
In the 1st round the DH keys derived are: a^{BX} , a^{CD}, a^{EF} , a^{GH} .
In the 2nd round the DH keys derived are: $a^{a^{XB}a^{CD}}$, $a^{a^{XB}a^{CD}}$, $a^{a^{EF}a^{GH}}$,$a^{a^{EF}a^{GH}}$.
In the 3d round the DH key derived is: $a^{a^{a^{XB}a^{CD}}a^{a^{EF}a^{GH}}}$ for all four pairs (this is the new group key).

If we take GSC A for instance after the three rounds it should have stored the following values: χ, a^{B}, $a^{a^{CD}}$, $a^{a^{a^{EF}a^{GH}}}$, $a^{a^{a^{a^{XB}a^{CD}}a^{a^{EF}a^{GH}}}}$. Observe that the intermediate values stored remain unchanged from the previous time. The members of the subgroup of A will use these intermediate values to construct the group key. But the members already acquire these intermediate values as well as χ. Thus, the GSC that witnesses a membership update needs to communicate nothing to its sub-group at the 3^{d} step.

GSC B for instance after the three rounds it should have stored the following values: β, a^{X}, $a^{a^{CD}}$, $a^{a^{a^{EF}a^{GH}}}$, $a^{a^{a^{a^{XB}a^{CD}}a^{a^{EF}a^{GH}}}}$. Observe that all intermediate values stored remain the same but for one: a^{X}. This is the only value that the GSC needs to communicate to the members of its sub-group after the derivation of the group key. Then, these members will be able to reconstruct the group key themselves by doing d exponentiations.

GSC F for instance, after the three rounds it should have stored the following values: f, α^{E}, $a^{a^{GH}}$, $a^{a^{a^{XB}a^{CD}}}$ $a^{a^{a^{EF}a^{GH}}}$, $a^{a^{a^{a^{AB}a^{CD}}a^{a^{EF}a^{GH}}}}$. Observe that all intermediate values stored remain the same but for one, $a^{a^{a^{XB}a^{CD}}}$. This is the only value that the GSC needs to communicate to the members of its sub-group after the derivation of the group key. Then, these members will be able to reconstruct the group key themselves by doing one exponentiation.

The case of addition/eviction for the 3^{d} step when (O) protocol is applied presents the following differences: The subgroup key of each participant A, B etc is calculated only by the participant and not by the members, from the partial contributions of its members as we have already seen in the original protocol. Thus, using the same example as previously for (MO) and (MOT) protocols we observe the following:
For all participants (GSCs) except for A and B, one value only, the updated one, is broadcast to the members of their subgroups. The values that are broadcast are blinded values, and their free communication over the network causes no harm for the security of the system. However, both GSCs A and B need to communicate to their members the value a^{XB} which is not a blinded one. This value is to be known only to participants A and B. If a node obtains an un-blinded value that is not supposed to know, it can combine it with any of the blinded values that it can get, since they are freely broadcast, and derive more un-blinded keys and perhaps derive the group key. Members of groups of GSCs X and B need to obtain the un-blinded value a^{XB} however. So, in this case since there exist no subgroup key, we do as follows: for every member j with DH key K_j, its GSC communicates the value: a^{XB/K_j}. Clearly, only the particular member that knows K_j can derive the appropriate value to construct the group key. Thus, such a value is communicated individually to every member that belongs either to the subgroup that witnesses a membership update (subgroup of GSC A in this example), or to the subgroup of the GSC that is mate to the subgroup with the updated subgroup key, at the first round of the 2nd step (GSC B in the example, communicates with GSC A to derive a^{AB}).

GSC Storage:

(MO): The M_n member plays the role of the GSC since it needs to store the whole up-flow message that consists of n intermediate values each with n-1 exponents and of one value of n exponents. So in essence, we can assume that the GSC stores n+1 messages of length K. Since every GSC performs a DH key exchange with $\lceil {}^{n-2^d}/_{2^d} \rceil$ of the total members only, during the 1st step of the 2^d-Octopus protocol, the storage cost for the GSC is $\lceil {}^{n-2^d}/_{2^d} \rceil$. For the 2nd step, each participant needs at the worst case to store the value it computes during each of the total d rounds, so it stores d values. In total, the GSC stores K ($\lceil {}^{n-2^d}/_{2^d} \rceil$+d) bits.

(O): The GSCs store all the information that is stored by (MO) and additionally they could store some of the intermediate products needed to derive the values: AB/K_j, for each member j. These products are approximately $\lceil {}^{n-2^d}/_{2^d} \rceil$ for each GSC as we are going to analyze in what follows.

(MOT): Since each TGDH tree acquires $\lceil {}^{n-2^d}/_{2^d} \rceil$ members, the GSC storage cost according to the TGDH formulas is: $\lceil {}^{n-2^d}/_{2^d} \rceil$+log$\lceil {}^{n-2^d}/_{2^d} \rceil$+1 at the 1st step and $\lceil {}^{n-2^d}/_{2^d} \rceil$+log$\lceil {}^{n-2^d}/_{2^d} \rceil$+(d+1) at the 2nd.

Member Storage: A member needs to store the session key and: the private 2-party DH key between the member and its GSC when (O) is applied, the private and the subgroup key when (MO) is applied, the private and the h blinded keys for the TGDH tree when (MOT) is applied. Every member needs additionally store the d parts that serve in constructing the group key and are sent by its GSC.

Initial Communication:

(MO): For the GDH.2 we have: n rounds (n-1 unicast messages from the M_i member to the M_{i+1} member of length (i+1) K bits and one multicast message from member M_n to the rest n-1 members of length nK bits). The total message length in average is: $\sum_{i=1}^{n-1}(i+1)K + nK = (n^2+3n-2)K/2$

Initially, each of the 2^d participants establishes a DH key with its group of $\lceil {}^{n-2^d}/_{2^d} \rceil$ members. Using the GDH.2 formula we get: 2^{d-1} ($\lceil {}^{n}/_{2^d} \rceil^2$+3$\lceil {}^{n}/_{2^d} \rceil$-2) K for the total message length for the initial establishments of the group keys for all the 2^d group leaders.
Then, the 2^d GSCs execute the 2^d – Cube protocol via DH exchanges with initial values the keys they have obtained from the operation described above. We have a total of d rounds, in each of which we have a total exchange of 2^{d-1} 2 messages (for every simple application of DH we have exchange of two messages, and for every round we have 2^{d-1} pairs and the group key is computed). So, we have a total message length of: 2^ddK bits.
At the 3d step, all d values are broadcast to the group as d separate messages. Here, we broadcast the first value to the whole subgroup, eliminating the contribution that corresponds to the subgroup key. This way, only members of the subgroup that know the subgroup key of course, can add the subgroup key contribution to the value they received and derive the appropriate part of the group key. In this case, all de parts are broadcast to the subgroup. So the total number of messages in this case is: 2^dd K.
For (MO), we get for the total initial communication: 2^{d-1} ($\lceil {}^{n}/_{2^d} \rceil^2$+3$\lceil {}^{n}/_{2^d} \rceil$-2) K +$2^d$d K + 2^dd K= 2^{d-1} ($\lceil {}^{n}/_{2^d} \rceil^2$+3$\lceil {}^{n}/_{2^d} \rceil$-2) K+$2^{d+1}$d K

(O): The 2nd step is equivalent. At the 1st step we have $(n-2^d)$ unicast DH messages from the members of the group to their GSCs. The GSCs can broadcast their DH message to the members of their group, thus we have 2^d broadcasts. So we have n messages in total for the 1st step. In the 3d step, we have $(n-2^d)$ unicast and $(d-1)2^d$ broadcast messages for the d values that have to be communicated to the members respectively (the d-1 values are common to the whole subgroup) from the GSCs to their subgroup members. For (O), initial communication is: $(n+2^d d+(n-2^d)+(d-1)2^d)K= (2n+(d-1)2^{d+1})\ K$ bits.

(MOT): Initially at the 1st step, the sponsors of each subgroup broadcast $2\lceil n/2^d \rceil$keys for each subgroup. The 2nd step is the same as in the previous case so we have 2^d d K communication exchanges. For the 3d step the course of thought is exactly the same as for the (MO) protocol. So the initial communication is: $2^d\, 2\lceil n/2^d \rceil K + 2^d d\, K + 2^d\, d\, K = 2nK+2^{d+1}\, d\, K$

GSC and Member Initial Computation:

(O): Almost all calculations of members are pure exponentiations. Every GSC at the 1st step must do $2\lceil (n-2^d)/2^d \rceil$ exponentiations and must generate$\lceil n/2^d \rceil$ random numbers using the pseudo-random number generator C_{rr}. Furthermore, the GSC after exchanging secret keys with all its sub-group members must multiply them to obtain the secret key it is going to use at the 2nd step. Multiplication of elements that are K bits long each, has complexity of K^2 bits approximately. These are modulo multiplications, thus if we are multiplying $\lceil (n-2^d)/2^d \rceil$ elements, the overall complexity is $\lceil (n-2^d)/2^d \rceil\ K^2$. Every simple member does two exponentiations for this step. At the 2nd step the 2^d GSCs do 2d exponentiations since each of them participates in one DH key exchange per round. At the 3d step, every member does d exponentiations as we have already observed. Every GSC computes $\lceil (n-2^d)/2^d \rceil$ products of $(\lceil (n-2^d)/2^d \rceil)-1$ members' secret keys. Each product consists of $(\lceil (n-2^d)/2^d \rceil-1)$ factors (the factor that is missing is the secret key associated with the member to which the GSC intends to communicate the result of the particular multiplication). The complexity of these products seems to be $(1/2)\lceil (n-2^d)/2^d \rceil(\lceil (n-2^d)/2^d \rceil-1)$ if we compute each product from scratch. But there is a way to substantially reduce this complexity by working as follows: Assume we separate the secret keys of the members into k groups of approximately equal size x. Thus, $A=\lceil (n-2^d)/2^d \rceil=k\ x$. We will work separately for each such group of x keys. Thus we will find the complexity of the same problem but for a smaller size of group each time, assume P(x), and we will combine all groups together. Generally, the number of multiplications needed in a group of size x, to find all combinations of multiplications of (x-1) and of x elements is: $x(x+1)/2-2$ (This result has been also derived for the same operation in GDH.2 protocol). Assume that we find the complexity for the operation of deriving the appropriate products in each such group. In order to derive an element which is the result of multiplying A-1 factors we have to multiply now k factors. Not two factors are taken out of the same group (there exist k groups of size x). Every such group contains x elements: (x-1) elements are derived by multiplying (x-1) keys, and one is derived by multiplying x keys. For the (x-1) elements of the particular group, say group 1, we have already found the rest of the factors that should be picked from the rest (k-1) groups in order to carry out the final multiplication: we should pick those elements that result from multiplication of x bits. For these (k-1) elements we do (k-2) multiplications, and the product is going to apply to all x-1 elements of group 1. This is exactly the course of thought for all the rest k-1 groups. Thus for all (k-1) such combinations we have to do $(k-1)\ k/2-2$ multiplications. The complexity of combining all the elements together to produce the appropriate products, once we have calculated the complexity of producing the x elements of each group is: (k-1) k/2+k(x-1)-2. The overall complexity of the problem is: $P(A) = P(kx) = k(x(x+1)/2-2)+(k-1)\,k/2+k\,(x-1)-2 = A(x+1)/2 - (2A/x) + (A^2/2x^2) - (A/2x)+A-(A/x)-2$. We want to reduce this complexity as much as possible by defining the appropriate size of x. After calculations we get: $x^3+x=2A$, and we can say that the complexity after doing the appropriate calculations and after a little rounding is about: $P(A)= A(A^{1/3}+1.25)$. This is a substantial reduction in complexity. Now $P(\lceil {}^{n-2^d}\!/_{2^d} \rceil)=(\lceil {}^{n-2^d}\!/_{2^d} \rceil)^{4/3}+1.25 (\lceil {}^{n-2^d}\!/_{2^d} \rceil)$.

After the appropriate products have been derived, the GSC does the following calculation for each of its subgroup members: it raises the blinded key it gets from another GSC during the first round of the 2nd step, to the power of the product related to this member, and communicates the results to the member. It does the same for all its sub-group members, thus it does $\lceil {}^{n-2^d}\!/_{2^d} \rceil$ exponentiations. In total, the amount of computation each GSC does initially is: $\lceil {}^{n}\!/_{2^d} \rceil C_{rr}+(2\lceil {}^{n-2^d}\!/_{2^d} \rceil+2d+\lceil {}^{n-2^d}\!/_{2^d} \rceil)C_E+((\lceil {}^{n-2^d}\!/_{2^d} \rceil)^{4/3}+1.25(\lceil {}^{n-2^d}\!/_{2^d} \rceil))K^2$. The total initial computation on the part of each member is: (d+2) exponentiations.

(MO): The GSC (M_n) does n exponentiations when we apply the (MO) protocol. In the 1st step each one of the 2^d GSCs does $\lceil {}^{n-2^d}\!/_{2^d} \rceil$ exponentiations. Each member M_i does (i+1) exponentiations of the same complexity each. We can say that in average the exponentiations a member does are $(½)\lceil {}^{n-2^d}\!/_{2^d} \rceil$. And it does one more exponentiation when it gets the stream message in order to construct the session key K_n. In the 2nd step the 2^d GSCs do 2d exponentiations since each of them participates in one DH key exchange per round. In the 3d step every member does d exponentiations as we have noted in the observations. Therefore, a GSC does $\lceil {}^{n-2^d}\!/_{2^d} \rceil+2d$ exponentiations and generates $\lceil {}^{n}\!/_{2^d} \rceil$ random numbers (using the pseudo-random number generator C_{rr}) in total. A member does $(½)\lceil {}^{n-2^d}\!/_{2^d} \rceil+d$ exponentiations.

(MOT): At the 1st step, the sponsors of the same subgroup do $2\log\lceil {}^{n}\!/_{2^d} \rceil$ exponentiations according to our analysis for TGDH, and also generate $\lceil {}^{n}\!/_{2^d} \rceil$ random numbers (using the pseudo-random number generator C_{rr}). At the 2nd step, the 2^d GSCs do 2d exponentiations since each of them participates in one DH key exchange per round. At the 3d step, every member does d exponentiations as we have noted in the observations.

The papers on the original Octopus protocol do not refer to the cases that we have membership updates (additions/evictions) within the group. We will analyze the case of member addition/deletion and calculate the respective cost values for the (O), the (MO), and the (MOT) protocols.

The key idea here is to initially re-compute the subgroup key only for the subgroup that has accepted the new member to join. So, we run again the protocol executing the 1st step only, for the subgroup mentioned above. Then we run the 2nd and the 3d steps as we are going to describe in what follows. However we can observe here that at the 2nd step not all the calculations need to be done anew. The DH key exchange that does not involve participants whose key value has changed do not need to carry out again the same operation. We observe that at the first round we will modify the keys of two participants, at the second round we will modify the keys of four participants, at the third of eight participants, and at the dth round we will modify the keys of 2^d participants.

Important Security Constraint:

(O): In the case of member addition/eviction we do not acquire the GDH.2 or the TGDH properties to disguise the contributions of the rest of the members of the subgroup to the new/evicted member. If the protocol is used as such, then there is not backward secrecy for the new member. When the GSC sends the d parts of the group key to the new member it essentially sends to it the appropriate parts to construct the old group key. Also, there is no forward secrecy in the case of eviction because the evictee will be able to derive the new key simply by eliminating its own DH key (it may be possible under circumstances) from the previous group key. So, we do the following simple modification: one member preferably of the same subgroup in which the addition/eviction occurs, modifies its secret share and establishes a new DH key with its GSC. We want to have at least another exponent (other than the new member's for the addition case) modified, so that we can disguise the old value from any past and future subgroup members. Thus, the GSC in the case of member join does two DH key exchanges and in the case of eviction does one DH key exchange. Also, the GSC does four exponentiations at the 1st step for the join case and two for the eviction case. Two members do two exponentiations at the 1st step for the join case, and one member does one exponentiation at the same step for the eviction case.

(MO), (MOT): At the 1st step, GDH.2 protocol produces a sub-group key of the form $a^{ab\ldots z} = a^N$. At the 1st step, TGDH protocol produces a sub-group key of the form [redacted]$^{xy} = a^N$. The resulting key for both cases is equivalent to (O) in the case that subgroup contained one member only other than the GSC. A designated member or sponsor from each subgroup that is the first to derive the subgroup key and broadcast it to the rest of members, plays the role of the GSC and provides the subgroup key for the 2nd step. In the end, all 2^d participants broadcast to their subgroups d values similar to those broadcast at the 3rd step when (O) is applied. Each member raises one specific part out of the d parts it receives from its GSC, to the power of its subgroup key known from the 1st step. The subgroup key is known to all members that belong to the same subgroup and thus it suffices that all d values are broadcast to the members of the same subgroup. So the number of messages communicated in this case is 2^d d. For the addition/eviction events (d-1) of the d values need not be broadcast anew since they remain unchanged, provided that they are already stored in every member. The GSC in the group of which a change of membership occurred need not communicate anything to its members at the 3d step. All the rest of GSCs broadcast only the updated value to their members. Thus, in the case of member eviction/ addition the number of messages communicated at the 3d step is: 2^d-1.

GSC Addition Computation:

(MO): In GDH.2 M_n generates a new exponent N_n' and computes a new up-flow message using N_n' not N_n and sends it to the new member M_{n+1}. Now member M_{n+1} becomes GSC and computes the new key K_{n+1}. And it computes and broadcasts to the other group members the n sub-keys. So this member computes n+1 intermediate values, it does n+1 exponentiations. Since in our case the new number of nodes in this subgroup is $\lceil n/2^d \rceil$+1, we have $\lceil n/2^d \rceil$+1exponentiations at the 1st step of the algorithm. At the 2nd step of the algorithm the amount of exponentiations each one of the 2^d GSCs does is not the same. We have already explained that the results of some of the DH exchanges remain the same so we don't need to execute the DH operation for every participant in every round. Given that the basis in the d-dimensional vector space remains the same, we have a way to determine for every GSC in how many DH exponentiations it is going to participate. The participant that belongs to the subgroup in which an update of membership occurred participates in all the d rounds (as well as its mate from the first round of course). In average a GSC participates in $\lceil \frac{d+1}{2} \rceil$ rounds. In every such round, it does two exponentiations. Finally, at the 3d step all the GSCs have computed the group key, they just broadcast to the members of their sub-group the appropriate value indicated at the 3d step of the protocol. As we analyzed earlier, the GSC does no computations at this step. Thus, the GSC does in total: $(\lceil n/2^d \rceil+1+2\lceil \frac{d+1}{2} \rceil)C_E+C_{rr}$ computations.

(O): The GSC in the case of join generates two random numbers and does two DH key exchanges (four exponentiations). Then it multiplies the keys of all members together and executes a single exponentiation of this product (complexity of one exponentiation and $\lceil (n-2^d)/2^d \rceil K^2$ bits due to the multiplication). This sequence of operations can be also viewed as $\lceil (n-2^d)/2^d \rceil$ successive exponentiations with the DH key of each member every time. The 2nd step is handled exactly the same way as in (MO): each GSC does $2\lceil \frac{d+1}{2} \rceil$ exponentiations. At the 3d step only two GSCs do calculations as we have discussed in the observations. These two GSCs do exactly the amount of computations done by any GSC at the 3d step of the initial phase. Each of the two GSCs computes $\lceil (n-2^d)/2^d \rceil$ products of $(\lceil (n-2^d)/2^d \rceil)-1$ members' secret keys. Each product consists of $(\lceil (n-2^d)/2^d \rceil-1)$ factors, and the factor that is missing is the secret key of the member to which the GSC intends to communicate the result of the particular multiplication. After the appropriate products have been derived, the GSC does the following calculations for each of its subgroup members: it raises the blinded key that it gets during the first round of the 2nd step to the power of the product related to the particular member, and communicates the results to this member. It does the same for all its subgroup members, thus it does $\lceil (n-2^d)/2^d \rceil$ exponentiations. In total the amount of computation for each of the two GSCs is: $(\lceil (n-2^d)/2^d \rceil+4+2\lceil \frac{d+1}{2} \rceil + \lceil (n-2^d)/2^d \rceil)C_E + (\lceil (n-2^d)/2^d \rceil)^{4/3}+1.25\,(\lceil (n-2^d)/2^d \rceil)K^2 + 2C_{rr}$. The rest of GSCs do $(2\lceil (n-2^d)/2^d \rceil+2\lceil \frac{d+1}{2} \rceil)$ exponentiations. Assume that all GSCs store all the products from their previous multiplications, even the intermediate ones. In that case each of the two GSC needs to do as follows: It computes the product of the two updated DH keys. For $(\lceil (n-2^d)/2^d \rceil-2)$ members it suffices that their intermediate product of $(\lceil (n-2^d)/2^d \rceil-3)$ keys (the product lacks the contributions of the member itself and of the members that are associated with the updated keys) be multiplied with the product of the updated keys. Each of the rest two members is multiplied with one of the updated keys or the other respectively. In this case, each of the two GSCs need only carry out: $1+2+2(\lceil (n-2^d)/2^d \rceil-2) = 2\lceil (n-2^d)/2^d \rceil-1$ multiplications. In this case the total amount of calculations these two GSCs do is: $(2\lceil (n-2^d)/2^d \rceil+4+2\lceil \frac{d+1}{2} \rceil+\lceil (n-2^d)/2^d \rceil)\, C_E+2(\lceil (n-2^d)/2^d \rceil-1)\, K^2 +2C_{rr}$.

MOT): The sponsor does 2 log $\lceil (n+1)/2^d \rceil$ exponentiations at the 1st step (addition computation case in TGDH) and generates two random numbers. The 2nd step is handled exactly the same way as in (MO), each sponsor (GSC) does $2\lceil \frac{d+1}{2} \rceil$ exponentiations. The 3d step is similar to the (MO) case, during which the GSC does no further exponentiations. So, the total amount of calculations for the (MOT) protocol is: $(2\log\lceil (n+1)/2^d \rceil+2\lceil \frac{d+1}{2} \rceil)\, C_E +2C_{rr}$.

Member Addition Computation:

(MO): The member gets the broadcast stream and it only needs to do one exponentiation to get the key K_{n+1} in GDH.2 protocol. Thus, members that belong to the subgroup in which membership update occurs are required to do one exponentiation to get the new subgroup key during the 1st step. At the 3d step, as we already know each member of every sub-group must do d exponentiations or less to reconstruct the group key, in average two.

(O): At the 1st step two members of the same subgroup only, are required to do two exponentiations to create a two-party DH with the group GSC each. Then, at the 3d step, each member of every subgroup does d exponentiations or less to reconstruct the group key, in average two.

(MOT): At the 1st step the members do in average two exponentiations as we have seen in the TGDH formulas. At the 3d step, as we already know each member of every subgroup does d exponentiations or less to reconstruct the group key, in average two.

Add Communication:

(MO): For the GDH.2, the M_n member sends to the M_{n+1} member an up-flow message of (n+1) K length. Then M_{n+1} member broadcasts to all n members its n+1 intermediate values, of total length (n+1) K. So the total length of the add communication is 2(n+1) K.
At the 1st step of the protocol, one of the 2^d subgroups only performs the GDH.2. In this group we have $\lceil {n-2^d}/{2^d} \rceil$ members so the length of the add communication is: $2\lceil {n}/{2^d} \rceil$K.
At the 2nd step as we have already discussed we have d rounds, in the round i however we have 2^i DH key exchanges instead of 2^d. For every DH exchange we have 2 communication messages sent, one to each node and the common key is computed. So we have 2 $(1+2+\ldots 2^{d-1})$ communication messages. The total number of bits communicated is $2(2^d-1)$ K bits.
At the 3d step, the GSC needs only communicate (broadcast) one value to the members of its subgroup as we have already discussed. All members receive the value that has changed for them. However, all members of the subgroup in which the new member joined get no value. So, $(2^d-1)\lceil {n-2^d}/{2^d} \rceil$ members get a single message. The total number of bits transmitted during the 3d step is: (2^d-1) K. Consequently, the total number of bits communicated in the case of a member join for the (MO) protocol is: $2\lceil {n}/{2^d} \rceil K+2(2^d-1) K+(2^d-1) K$.

(O): At the 1st step two DH exchanges take place (four communication messages). The 2nd step is exactly the same way as in (MO). During the 3d step however the communication of messages to the members of two groups (the one in which the membership update occurs and the one whose participant communicates during the 1st round of the 2nd step with the participant of the updated member) is done in unicast. So, $2\lceil {n-2^d}/{2^d} \rceil$ members get one message and (2^d-2) groups get a single broadcast message. The total number of bits transmitted during the 3d step is: $((2^d-2) +2\lceil {n-2^d}/{2^d} \rceil)$K. The total number of bits communicated in the case of a member join for the (O) protocol is: $4K+2(2^d-1) K+(2^d-2)K+2\lceil {n-2^d}/{2^d} \rceil K$.

(MOT): At the 1st step the new member of the subgroup sends its blinded key to its sibling (sponsor). Then, the sponsor broadcasts the updated h blinded keys to all the members in the tree, and additionally communicates all the n blinded keys of the tree to the new member. So, since the subgroup acquires $\lceil {}^{n+1}/_{2^d} \rceil$ nodes, the total communication bits for this step are: (1+log$\lceil {}^{n+1}/_{2^d} \rceil$+$\lceil {}^{n+1}/_{2^d} \rceil$) K. The 2nd step is handled the same way as in (MO). The number of bits communicated is $2(2^d-1)$ K. The 3d step is again similar to the 3d step for (MO). So the total communication is: (1+log$\lceil {}^{n+1}/_{2^d} \rceil$+$\lceil {}^{n+1}/_{2^d} \rceil$) K + $2(2^d-1)$ K + (2^d-1) K.

GSC Deletion Computation:

(MO): For the GDH.2 protocol, if member M_p is evicted then M_n, that is the current GSC, computes a new set of n-2 sub-keys. What is missing is the term $\alpha^{N_1*...N_{p-1}*N_{p+1}...N_{n-1}*N_n'}$ $\alpha^{N_1*...N_{p-1}*N_{p+1}...N_{n-1}*N_n'}$ so that M_p cannot compute the new key. The procedure is exactly like the GSC Addition Computation case, with the only difference that now M_n must do n-1 exponentiations. It also generates one random number using the pseudo-random number generator C_{rr}.
During the 1st step, the GSC has to carry out $\lceil {}^{n}/_{2^d} \rceil$ exponentiations to compute the updated subgroup key according to the GDH.2 formulas. For the 2nd step as we have already described for the GSC addition computation, the participant that belongs to the subgroup of the new incomer participates in all the d rounds as well as its mate from the first round. In average a GSC participates in $\lceil \frac{d+1}{2} \rceil$ rounds, and does $2\lceil \frac{d+1}{2} \rceil$ exponentiations. For the 3d step, no further exponentiations are required from the GSCs. Thus, the total computation of a GSC for the case of member deletion is: ($\lceil {}^{n}/_{2^d} \rceil$+2$\lceil \frac{d+1}{2} \rceil$) C_E +C_{rr}.

(O): The GSC for the eviction case does one DH key exchange (two exponentiations) and generates one random number using the pseudo-random number generator C_{rr}. Then it multiplies the keys of all members together and exponents the resulting product (one exponentiation and $\lceil {}^{n-2^d}/_{2^d} \rceil K^2$ bits due to the multiplication). This sequence of operations can be also viewed as $\lceil {}^{n-2^d}/_{2^d} \rceil$ successive exponentiations with the DH key of each member every time. The 2nd step is handled exactly as it is described for the previous cases: each sponsor (GSC) does $2\lceil \frac{d+1}{2} \rceil$ exponentiations. At the 3d step only two GSCs do calculations as we have discussed in the observations. Each of the two GSCs computes $\lceil {}^{n-2^d}/_{2^d} \rceil$ products of ($\lceil {}^{n-2^d}/_{2^d} \rceil$)-1 members' secret keys. Each product consists of ($\lceil {}^{n-2^d}/_{2^d} \rceil$-1) factors, and the factor that is missing is the secret key of the member to which the GSC intends to communicate the result of the particular multiplication. After the appropriate products have been derived, the GSC does the following calculation for each of its subgroup members: it raises the blinded key that it gets during the first round of the 2nd step to the power of the product related to this member, and communicates the results to the member. It does the same for all its subgroup members, thus it does $\lceil {}^{n-2^d}/_{2^d} \rceil$ exponentiations. In all three steps the total complexity of computations (exponentiations and multiplications) for each of these two GSCs is: ($\lceil {}^{n-2^d}/_{2^d} \rceil$+2+2$\lceil \frac{d+1}{2} \rceil$+$\lceil {}^{n-2^d}/_{2^d} \rceil$) C_E+($\lceil {}^{n-2^d}/_{2^d} \rceil)^{4/3}$+1.25($\lceil {}^{n-2^d}/_{2^d} \rceil$)$K^2$. The rest of GSCs do (2$\lceil {}^{n-2^d}/_{2^d} \rceil$+2$\lceil \frac{d+1}{2} \rceil$) exponentiations. Assume that all GSCs store all the products from their previous multiplications, even the intermediate ones. In that case each of the two GSCs needs to do as follows: for ($\lceil {n-2^d}/{2^d} \rceil$-2) members it suffices that their intermediate product of ($\lceil {n-2^d}/{2^d} \rceil$-3) keys (the product lacks the contributions of the particular member itself, of the evicted one and of the member associated with the updated key) be multiplied with the updated key. In this case, each of the two GSCs need only carry out: 1+2($\lceil {n-2^d}/{2^d} \rceil$-3) = (2$\lceil {n-2^d}/{2^d} \rceil$-5) multiplications. The total amount of calculations these two GSCs do at all three steps is: $(2\lceil {n-2^d}/{2^d} \rceil+2+2\lceil \frac{d+1}{2} \rceil+\lceil {n-2^d}/{2^d} \rceil)C_E+(2\lceil {n-2^d}/{2^d} \rceil-5)K^2+C_{rr}$.

(MOT): At the 1st step, exactly as in the Add Sponsor Computation case, the sponsor (GSC) does 2h exponentiations, thus 2log$\lceil {n-1}/{2^d} \rceil$exponentiations since the size of each group is approximately: $\lceil {n-1}/{2^d} \rceil$. It also generates a random number using the pseudo-random number generator C_{rr}. The 2nd step is handled exactly the same way as in (MO): the number of exponentiations for each GSC is: $2\lceil \frac{d+1}{2} \rceil$. For the 3d step, no further exponentiations are required from the GSCs. Thus, the total computation of a GSC for the case of member deletion is: $(2\log\lceil {n-1}/{2^d} \rceil+2\lceil \frac{d+1}{2} \rceil)\, C_E+ C_{rr}$.

Member Deletion Computation

(MO): For GDH.2, it is exactly the same as for the member Addition Computation case. The member gets the broadcast stream and it only needs to do one exponentiation to get the key K_{n+1} in GDH.2. So, for the 1st step, every member needs to do one exponentiation. Then, at the 3d step the member does d exponentiations at maximum or two in average, and the group key is computed anew.

(O): At the 1st step only one member of the subgroup of the evicted member is required to do two exponentiations. Then, at the 3d step, every member does d exponentiations at maximum or two in average, and the group key is computed anew.

(MOT): At the 1st step the members of the subgroup of the evicted member compute the new subgroup key after having received the updated blinded keys that the sponsor broadcasts. In analogy to the case of member addition, every member in average carries out two exponentiations. At the 3d step, every member does d exponentiations at maximum, or two in average and the group key is computed anew.

Delete Communication:

(MO): In GDH.2 only member M_n broadcasts a message of length n-1 to the rest of the members. So the total overhead for the delete communication is: (n-1) K. At the 1st step, the total messages for the delete communication are: ($\lceil {n-2^d}/{2^d} \rceil$-1) K. Next, the 2nd and the 3d steps are exactly as in the Add Communication case for (MO). So the total number of bits that have to be communicated is: ($\lceil {n-2^d}/{2^d} \rceil$-1) K+2($2^d$-1) K+($2^d$+1) K.

(O): At the 1st step, we update the contribution of another member of the subgroup in which a membership eviction occurs. This member exchanges new DH key with its GSC (two communication messages). The 2nd step is handled the same way as in the Addition Communication case for (O). The total number of bits communicated is $2(2^d-1)$ K bits. At the 3d step however the communication of messages to the members of two groups (the one in which the membership eviction occurs and the one whose GSC communicates during the 1st round of the 2nd step with the GSC of the evicted member) is done in unicast. So, $2\lceil (n-2^d)/2^d \rceil$ members get one message and (2^d-2) groups get a single broadcast message. The total number of bits transmitted during the 3d step is: $(2^d-2)K+2\lceil (n-2^d)/2^d \rceil K$. So, the total number of bits to transmit in (MO) is: $2K+2(2^d-1)K+(2^d-2)K+2\lceil (n-2^d)/2^d \rceil K$ bits.

(MOT): At the 1st step, the sponsor broadcasts the updated h blinded keys to all the members of the tree. So, we have $\log\lceil (n-1)/2^d \rceil$ messages broadcast since the size of the subgroup is $\lceil (n-1)/2^d \rceil$. For the 2nd step we have the same exchange of messages exactly as in the 2nd step of the Addition Communication case. The total number of bits communicated is $2(2^d-1)$ K. The 3d step is again similar to the 3d step for (MO). So the total communication is: $\log\lceil \frac{n-1}{2^d} \rceil K+2(2^d-1)K+(2^d-1)K$.

Cost	2^d-Octopus (O)	Mod. 2^d- Octopus (GDH.2)-(MO)	Mod.2^d-Octopus (TGDH)-(MOT)
GSC Storage	$K(\lceil (n-2^d)/2^d \rceil+d)$ / $K(2\lceil (n-2^d)/2^d \rceil+d)$	$K(\lceil (n-2^d)/2^d \rceil+d)$	$(\lceil n/2^d \rceil+\log\lceil n/2^d \rceil+d)K$
Member Storage	$(2+d)K$	$(d+1)K$	$(\lceil n/2^d \rceil+\log\lceil n/2^d \rceil+d)K$
Initial GSC Comput.	$(3\lceil (n-2^d)/2^d \rceil+2d)C_E+(\lceil (n-2^d)/2^d \rceil)^{4/3}+1.25(\lceil (n-2^d)/2^d \rceil)K^2+\lceil n/2^d \rceil C_{rr}$	$(\lceil n/2^d \rceil+2d)C_E+\lceil n/2^d \rceil C_{rr}$	$(2\log\lceil n/2^d \rceil+2d)C_E+\lceil n/2^d \rceil C_{rr}$ at max.
Initial Members Comput.	$(d+2)C_E$	$((1/2)\lceil n/2^d \rceil+d)C_E$	$(2\log\lceil n/2^d \rceil+d)C_E$ at max
Initial Comm/tion	$(2n+(d-1)2^{d+1})K$	$(2^{d-1}(\lceil n/2^d \rceil^2+3\lceil n/2^d \rceil-2)+2^{d+1}d)K$	$(2^d 2\lceil n/2^d \rceil+2^{d+1}d)K$
Add GSC Computat.	$(3\lceil (n-2^d)/2^d \rceil+4+2\lceil \frac{d+1}{2} \rceil)C_E+2C_{rr}+(\lceil (n-2^d)/2^d \rceil)^{4/3}+1.25(\lceil (n-2^d)/2^d \rceil)K^2$ / $(3\lceil (n-2^d)/2^d \rceil+2\lceil \frac{d+1}{2} \rceil+4)C_E+2C_{rr}+2(\lceil (n-2^d)/2^d \rceil-1)K^2$, one $(2\lceil (n-2^d)/2^d \rceil+2\lceil \frac{d+1}{2} \rceil)C_E$ rest	$C_E(\lceil \frac{n+1}{2^d} \rceil+1+2\lceil \frac{d+1}{2} \rceil)+C_{rr}$, one $C_E(2\lceil \frac{d+1}{2} \rceil)$, rest	$C_E(2\log\lceil \frac{n+1}{2^d} \rceil+2\lceil \frac{d+1}{2} \rceil)+2C_{rr}$, one GSC $C_E(2\lceil \frac{d+1}{2} \rceil)$, rest
Add Members Comput.	$4C_E$, two $(2+d)C_E$ max. $2C_E$, the rest dC_E max.	$3C_E$, one subgroup $(1+d)C_E$ max. $2C_E$, rest dC_E max.	$4C_E$, one member $(h+d)C_E$ max $2C_E$, rest dC_E max

Add Comm/tion	$(4+2(2^d-1)+(2^d-2)+2\lceil (n-2^d)/2^d \rceil)K$	$(2\lceil (n+1)/2^d \rceil+2(2^d-1)+(2^d-1))\ K$	$(\log\lceil (n+1)/2^d \rceil+\lceil (n+1)/2^d \rceil+2(2^d-1)+(2^d-1))K.$
Delete GSC Comput.	$(3\lceil (n-2^d)/2^d \rceil+2+2\lceil \frac{d+1}{2} \rceil)C_E + C_{rr}+(\lceil (n-2^d)/2^d \rceil)^{4/3}+1.25(\lceil (n-2^d)/2^d \rceil)K^2$ / $(3\lceil (n-2^d)/2^d \rceil+2+2\lceil \frac{d+1}{2} \rceil)\ C_E + C_{rr}+ +(2\lceil (n-2^d)/2^d \rceil-5)\ K^2$, one $(2\lceil (n-2^d)/2^d \rceil+2\lceil \frac{d+1}{2} \rceil)$ rest	$C_E(\lceil \frac{n-1}{2^d} \rceil+2\lceil \frac{d+1}{2} \rceil)+C_{rr}$, one $C_E(2\lceil \frac{d+1}{2} \rceil)$, rest	$C_E(2\log\lceil \frac{n-1}{2^d} \rceil+2\lceil \frac{d+1}{2} \rceil)+C_{rr}$, one $C_E(2\lceil \frac{d+1}{2} \rceil)$, rest
Del. Members Comput.	$3C_E$, two $(1+d)C_E$ max. $2C_E$, the rest dC_E max.	$3C_E$, one subgroup or $(1+d)C_E$ max. $2C_E$, rest dC_E max.	$4C_E$, one member $(h+d)C_E$ max $2C_E$, rest dC_E max
Delete Comm/tion	$(2+2(2^d-1)+(2^d-2)+2\lceil (n-2^d)/2^d \rceil)\ K$	$((\lceil \frac{n-1}{2^d} \rceil-1)+2(2^d-1)+(2^d+1))\ K$	$(\log\lceil \frac{n-1}{2^d} \rceil+2(2^d-1)+(2^d-1))K$