Decentralized Key Management for Large Dynamic Multicast Groups using Distributed Balanced Trees

MS Thesis for the Degree of

Submitted in Partial Fulfillment
of the Requirements for the
Degree of

Master of Science (Computer Science)

at the

National University of Computer & Emerging Sciences

by

Junaid Haroon (MSCS018)

November 2004

Approved:

____

Head
(Department of Computer Science)

___________20 ____

Approved by Committee Members:

Advisor

Name & designation with University

Co –Advisor (optional)

Name & designation with University

Other Members:

Name(s) & designation with University

Vita

Mr. Junaid Haroon was born in Murree, Pakistan on August 16, 1979. He received a Bachelor of Science in Computer Science from National University of Computer and Emerging Sciences, Lahore in 2001. From 2001 – 2004, he taught at the same university. The research in this dissertation was carried out from 2002 to 2004.

His other interests include playing table tennis, and reading fiction.

Abstract

Many schemes of key management have been introduced for large dynamic multicast groups e.g. pay per view networks. The most successful schemes are based on tree based key hierarchies including LKH (Logical Key Hierarchy) and OFT (One-way function trees). All tree based schemes scale logarithmically to the number of members of the group when the tree is balanced and the performance can deteriorate towards linear when the tree gets out of shape. Also tree based schemes are inherently centralized and larger groups can impose high load on the single server causing the performance to deteriorate.

This thesis proposes a tree based scheme which solves both of these problems. The thesis presents an approach to dynamically keep the tree balanced by using intelligent member insertion and using AVL style rotations at removal of members. The scheme supports a total distributed nature with any number of subgroup controllers handling their members but all being part of a larger tree. Therefore getting the advantages of a single tree hierarchy and the load balancing that is possible with a distributed architecture.

Table of Contents

Chapter 1
Introduction

A major problem in widespread use of secure multicast communication is key management. Key management is the mechanism by which all members of a multicast group come to know of a secret that is used in the encryption of data sent on the group. Encryption is the basis of all secure communication. A number of different solutions have been proposed for efficient key management with varying results. The ones based on hierarchical key trees have been the most successful in terms of performance and scalability for large groups.

This thesis is an attempt to improve the performance of hierarchical binary trees based key distribution mechanism by distributing the work among a number of subgroup controller that dynamically share their load and balance the members such that every controller has as many members as it can safely accommodate and the whole tree is kept as much balanced as possible. The requirements on balancing and load distribution are laid down and the algorithms to ensure them are proposed as part of the work.

The remainder of this thesis is organized as follows. In this chapter the background is established by discussing network security and network layer multicast and a discussion of the issues that arise when trying to implement security in multicast communication. Key management solutions for group communications are categorized and key graph based schemes discussed in detail. In chapter 2 the proposed protocol and its details are presented with a suggested approach to reach the desired goals. Chapter 3 compares the modified scheme with the original and gives an analysis. It also gives some empirical results regarding the overhead of rotations. Chapter 4 contains conclusion and directions for future work.

1.1 Background

Before the widespread use of computers and electronic means of storing data and conducting business, theft was the only threat and physical security was the only measure against it. With the advent of computers, paper based documents were slowly replaced by electronic data. The security of computer files became an issue as they cannot be secured by placing them in separate physical lockers. With time shared systems especially those that can be accessed over a public network, the problem became more complex and security got more attention. The field of computer security comprises of a study of all tools and mechanisms to ensure security of computer based systems.

Traditional security measures are insufficient as network and communication facilities have enabled computers to communicate over large inter networks that pass through public networks like Internet. New threats are possible in this scenario that attack data during transmission. The field of detecting, preventing, and correcting security problems involving transmission of data is known as network security. There are other security issues in a networked system that do not involve the transmission of data, like local access to an unauthorized user or an unwanted program like a worm or virus destroying data on the computer. These are serious issues present on networked systems but are outside the domain of network security as they do not involve the transmission of data. The spread of a worm or virus is however in the domain of network security if it got injected into the system via the network.

1.2 Network Security

To understand network security it is necessary to discuss the possible threats or attacks on data in transit and the security services needed to prevent them and finally the mechanisms to implement those services. All three are necessary to capture the overall picture of network security.

1.2.1 Security Attacks

Under the normal flow of data as depicted in Figure 1 the data sent from the source reaches the destination and no third party peeks at the data. Two types of violations are possible with this normal flow of data; passive violations and active violations. Passive violations are those when the destination receives the data sent from the source unmodified but the data has also been received somewhere else as well. Passive attacks are difficult to detect as, neither the source nor the destination is seeing any unusual behavior. Active violations are those when the destination receives data forged or modified by a third party. Such attacks are difficult to prevent, however it is possible to detect them. Once they are detected human intervention is required to take corrective action.

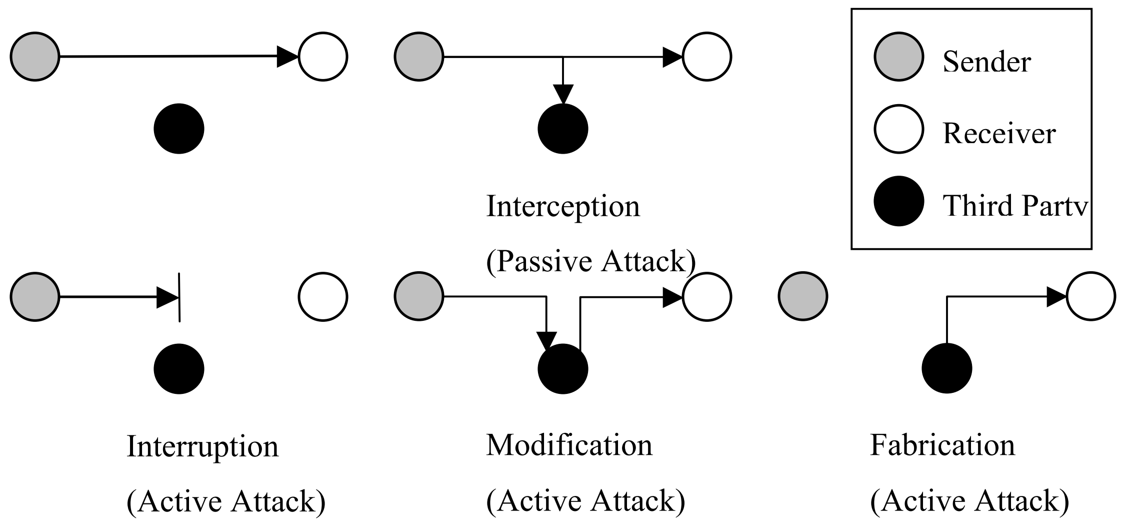

Figure 1 – Different types of attacks in a two party communication

Interception is a passive threat and occurs when the data is stolen in transit from the source to the destination and unauthorized third parties make access to it. This can result in release of data contents to unintended audience for unlawful purposes. Interception is very difficult to detect especially over public networks, but it can be prevented. A rather involved variation of the use of intercepted data is traffic analysis. Traffic analysis makes an educated guess regarding the communication going on between two parties by analyzing the amount, frequency, and timeline of the traffic flowing between them. Traffic analysis is difficult to prevent and in certain cases can provide sufficiently meaningful data to the third party.

Interruption is an active attack that results in the destination not receiving data sent for it. Denial of service attacks making a computer unavailable or unusable for service to intended clients is of this category. The two most dangerous active attacks are *modification* and *fabrication*. Modification is an attack where data is modified in transit. The sender believes its data is delivered to the destination and the destination believes the data it received is from the source but actually the third party is modifying it on the fly and gaining access which was intended for the source. Fabrication is a communication between the third party and the destination, but the third party prepares messages in such a form that the destination believes they are coming from the source, where the actual source is unaware of any communication going on at all. The unauthorized party pretends to be someone authorized, which is also known as masquerading or impersonating. Often fabrication is done after interception and the intercepted data is replayed to gain access into the system.

1.2.2 Security Services

Services that are needed to ensure safe transmission of data are called security services. They would be implemented by some underlying mechanism as discussed in the next section. Certain security services that provide protection against security attacks on data in transit are discussed below.

Confidentiality ensures that data transmitted from the source reaches the destination and no third party can peek into it. It is a protection of transmitted data from passive attacks like interception. *Authentication* ensures that the two parties involved in a communication are the two they claim to be, and keeps it ensured during the whole session. This ensures that fabrication attacks are not possible. *Integrity* ensures that the contents of a message are not changed in transit from the sender to the receiver. Modification is an attack against integrity. *Non-repudiation* service ensures that neither the sender nor the receiver can deny the transmission of a message that has been successfully transferred and acknowledged. This service is very important for businesses that are dependant on networked and distributed systems. An example can be the denial of a placed order which can cause serious issues and without this service there is no concrete proof as to who is correct. *Access Control* requires that every entity accessing a system must be authenticated to ensure he is who he claims to be. *Availability* ensures that the system will remain available to those for whom it is intended for. Denial of service and Interruption are attacks against availability.

1.2.3 Security Mechanism

There is no single security mechanism to implement the above mentioned services. There are various efforts differing in particular details but a general model can be sketched that is common to all these mechanisms. The network security model as shown in Figure 2 shows that the data undergoes some security transformations before being sent on the public network where the security threats may exist [1]. These transformations use encryption which forms the basis of all security mechanisms. Encryption transforms the data using a secret value such that the transformed data is meaningless in itself unless it is decrypted and the secret value is needed to decrypt it. This secret value used in encryption is called a key.

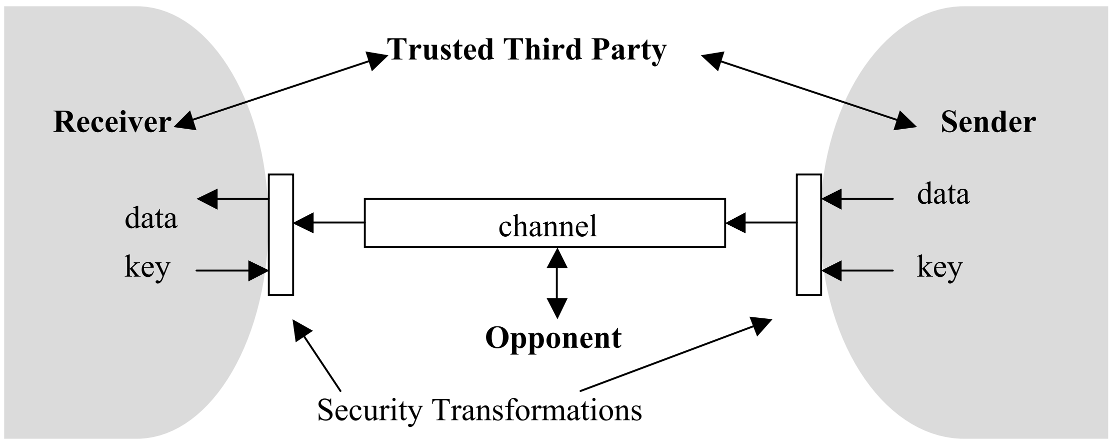

Figure 2 – A model for two party secure communication

There are two types of encryption; symmetric encryption and asymmetric encryption. Symmetric encryption uses a single key for encryption and decryption while asymmetric encryption uses one key for encryption and the other for decryption. Symmetric encryption is also known as private key cryptography while asymmetric encryption is also known as public key cryptography. With public key cryptography the party keeps one key with it called the private key and the other key is disclosed to everyone called the public key.

Public key cryptography can be used for source authentication, message integrity, and confidentiality services. For source authentication and message integrity the source can generate a one way hash of the data and encrypt it with its private key. Such an encrypted hash proves that the data was really sent by the named sender who was the only one in possession of the private key. The receiver will decrypt the hash using the sender's public key and compare it with the hash it itself calculated. If the two are same the source is authenticated and the message integrity is ensured. To achieve confidentiality the source can encrypt the message and its hash with the public key of the receiver so that only the receiver is able to decrypt the data as the corresponding private key is only known by the receiver.

Symmetric encryption is much faster than asymmetric encryption. Therefore asymmetric encryption is used to agree upon a shared secret to be used for the remaining session using symmetric encryption. The Diffie Hellman protocol is used to arrive at a shared secret using a public number and a private number generated by the protocol. If the public number generated by the protocol is transferred using asymmetric cryptography then the Diffie Hellman protocol is a very secure way to arrive at a shared secret.

The security of the above scheme assumes that the public key of the other party can be trusted. If the public key is transferred via the network, it faces the same security threats as the data that will be sent encrypted using public cryptography. For this reason trusted third parties are required whose public keys are well known through mediums other than the network. These trusted third parties register parties who want to do secure communication in their database and get to know their public keys and their authenticity via mediums other than the network. These trusted third parties then issue digital certificates that contain the information of a party and its public key digitally signed by the third party. As the public key of the trusted third party is well known the public key of the other party can be verified by decrypting the hash contained in the certificate and comparing it with the hash calculated over the message. This builds a chain of trust.

1.2.4 Network Layer Security

Transmitted data from the source to the destination passes through the five layers of the TCP/IP model as shown in Figure 3. Each of these layers is a possible target for the implementation of network security and certain implementations exist for each of them.

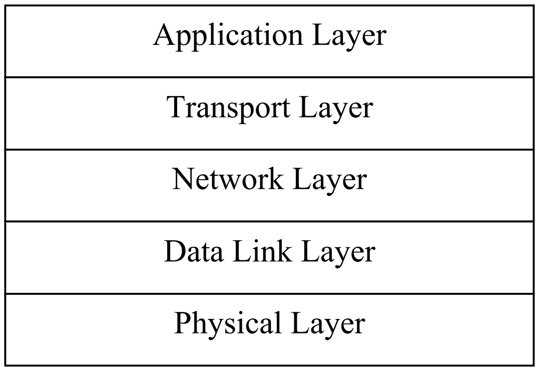

Figure 3 – Five layers of the TCP/IP Protocol stack

Physical layer security means securing the communication links. Nothing more can be done as the signals passing by have no other significance at this layer. This is the mechanism used in private networks physically protected from unauthorized access. Network security in a public network makes no sense on the physical layer since the network is by definition a public network and only a few links can be protected at most.

Security at the data link layer means having dedicated special links over which the safe transmission of data can be conducted. Security mechanism in Automated Teller Machines is an example of security at this layer. Security is fast and transparent at this layer but requiring special links makes it infeasible for the Internet or as a general network security mechanism.

Due to the absence of security implementations at a lower layer or due to specific application requirements a number of applications have implemented security inside the applications and are in widespread use. Email security using PGP and S/MIME is one example which is in heavy use for secure mailing.

Some applications have requirements that cannot be met at a lower layer. They require fine grained security with knowledge of the content of data in the message. One example is secure electronic transactions (SET) where a single message from the customer contains data for two recipients, the merchant and the bank, and each cannot see the other's data while the two are packed in an integral unit as well. Such applications will always need an implementation at the application layer. However when such special requirements are not there security at the application layer is a duplication of effort for each application needing security and the management of security related information for a large number of applications becomes more and more difficult for the security personnel.

Security at the transport layer is a reasonable alternate to data link layer and application layer security since the transport layer is shared by many application. Also the transport layer can ensure end to end security. However if other transport protocols need security then there is a duplicate effort for them which again results in complicated and may be inconsistent implementations and increased management overheads.

An example is Secure Sockets Layer (SSL) which was invented by Netscape and has now become an Internet draft standard known as Transport Layer Security (TLS). TLS implements security at the transport layer of TCP/IP protocol stack.

Every packet going out of the source or coming into the destination passes through IP, as depicted in Figure 4. Such an arrangement motivates implementing security at the network layer. Internet Protocol Security (IPSec) is an Internet draft standard [2] defined by Internet Engineering Task Force (IETF) to provide a detailed mechanism for network security at the network layer. It provides encryption [3] and authentication [4] services over public and private networks. IPSec is a flexible and modular standard allowing the use of various encryption and key distribution mechanism in a standardized way.

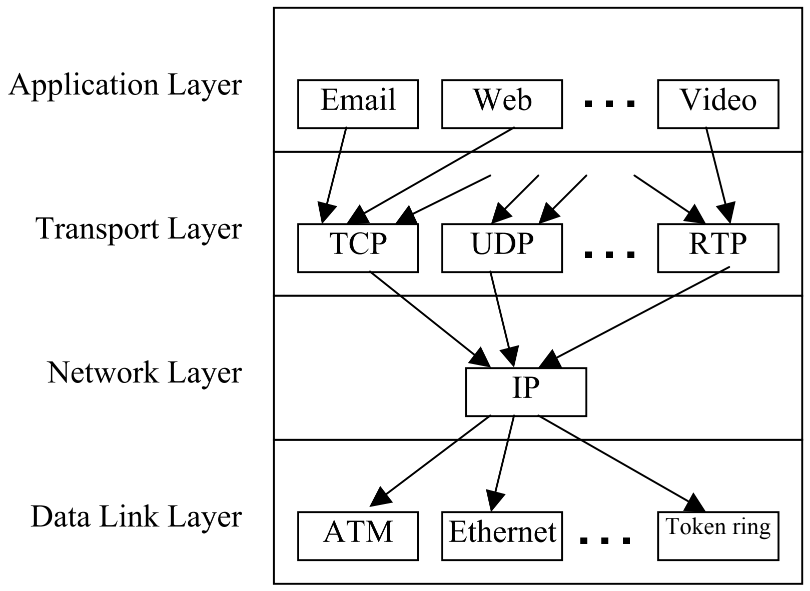

Figure 4 – The critical position of IP in the TCP/IP Protocol Stack for implementation of network security

Security at this layer will be transparent to applications and transport protocols, therefore businesses can take advantage of security services without the expense of modifying applications. Security at network layer is flexible enough to allow end to end security as well as security between intermediate nodes and between an end host and an intermediate node, since the network layer is the only layer knowing both the end hosts and every hop along the path. Security policy management becomes a system wide task rather than an application specific task which ensures consistent security policies across the whole system. Chances of error become less and therefore intrusion and security threats are suppressed.

1.3 Network Layer Multicast

There are different modes of packet delivery in a packet switched network. *Unicast* means the delivery of packet to one particular destination. *Broadcast* means the delivery of a packet to all hosts on a particular inter network or a segment of it. *Multicast* means the delivery of a packet to a set of hosts all of which have shown interest in receiving the packet. *Anycast* is a new concept introduced in IPv6[1] that means the delivery of a packet to one of a set of hosts, probably the nearest one.

[1] IPv6 is the new version of TCP/IP protocol suite with a very large address space.

Many applications need to deliver the same data to multiple destinations at the same time. Examples of such applications are multiparty video and audio conferencing, shared whiteboards, distributed interactive simulations, networked news, Usenet news, email distribution lists, networked games, and many others. Most of these applications use *replicated unicast* for multi-destination delivery. Others use *application level relays* (e.g. current video conferencing applications) that take packets from the source and forward them to the destinations or other relays. Certain network games and server discovery protocols use broadcast to reach multiple destinations. And lastly some new emerging applications have started using *network layer multicast* to reach their destinations.

Broadcast is the simplest solution for multi-destination delivery but it is limited in scope to smaller networks and less bandwidth intensive applications. If used on larger networks, routers must be configured to forward broadcast traffic in such a way that no routing loops are formed. Routing loops can result in bandwidth storms which can bring the whole network down to a stall. Also every node on the network is incurring the cost of processing packets that it might not be interested in.

At the other extreme is the possibility of replicated unicast but in this case there are equally prohibitive issues. The source now needs to know the network address of each destination. Managing this list whether statically or dynamically is a big overhead. Also the source bandwidth requirements increase proportionally with the number of clients and this number is always limited by an upper bound. Thus replicated unicast is not scalable to large number of clients.

Application level relays are special servers on the network that receive the data and take the responsibility to relay it to a set of destination nodes. Application level relays requires a static configuration of relays which cannot change with the dynamics of the group. Relays are involuntary receivers, since they can’t leave the group. There will be a duplication of effort and functionality in every application using such relays since they are application specific.

The main idea behind network layer multicast is to build a replication engine in the network layer so that application developers are relieved from the effort of managing host groups and multi-destination delivery and therefore the effort is not duplicated as well. This service enables the sender to send packets to a *group address* identifying a set of interest recipients [5]. The sender no longer needs to know the address of each interested receiver. The interested recipients inform their nearest router about this interest in a specific multicast group. The multicast capable routers then construct each group's *delivery tree* which connects the source network to those networks containing interested receivers. The delivery tree is also called *distribution tree* or *multicast tree*. These delivery trees allows the source to reach all receivers without fear of loops (which can lead to multicast storms), excess packet duplication, or additional management overhead.

Special ranges have been dedicated for multicast group addresses in both IPv4 and IPv6 [6]. In case of the 32-bit IPv4 address, any address that starts with 1110 is considered to be a multicast group address. While in case of the 128-bit IPv6 address any address starting with 11111111 is a multicast group address. Host group addresses may be permanently assigned by the Internet Assigned Numbers Authority (IANA) or transiently assigned for some duration of time.

Hosts can join or leave a multicast group by informing their nearest router using Internet Group Management Protocol (IGMP) [7]. Multicast capable routers use this information to construct each group's *delivery tree* which connects the source network to networks containing interested receivers. The delivery tree allows the network to route multicast packets to all recipients in a group. Several protocols have been suggested for use by multicast routers to construct the delivery tree. Among these are Distance Vector Multicast Routing Protocol (DVMRP), Multicast Open Shortest Path First (MOSPF), Protocol Independent Multicast Dense Mode and Sparse Mode (PIM-DM and PIM-SM), Core Based Tree (CBT), Ordered Core Based Tree (OCBT), HIP protocol for Hierarchical Multicast Routing and Border Gateway Management Protocol (BGMP) [8].

In the multicast architecture there is no authentication or authorization for group membership and for sending data to a group. Security threats become a problem in this scenario especially denial of service attacks are easier then ever. Due to this insecure nature of multicast it has not been widely deployed across the Internet [9]. However for research purposes and limited applications MBone [10] and Wide6Bone are used. MBone is the multicast backbone of the Internet connecting many multicast capable routers using IP tunnels. Wide6Bone is an IPv6 Multicast backbone formed on the same principles.

Before widespread use of multicast is seen, means must be established to provide authentication, authorization, confidentiality, data integrity, non-repudiation and availability for multicast traffic.

1.4 Secure Multicast

To make multicast secure, all transmitted data must be encrypted and only the group members should be in possession of the key that can be used to decrypt. Since membership in a multicast group is receiver initiated and the sender has no way to track the members of a group, security becomes a more important issue since no special access or mechanism is required to read the traffic of a multicast group. The only way to protect data is to encrypt it with a key that only group members know.

The unicast security model cannot be applied to multicast communication in a simple manner. Due to the multiparty nature of this communication, many issues arise that were not there in the unicast case [11, 12, 13, 14, 15]. The major difference from the unicast model is that a single key cannot be used for the whole session. Whenever a new member joins the group or an existing member leaves the group the key must be changed and all the members should be informed of the new key. Backward secrecy requires that the key should be changed when a member joins the group otherwise the new member will be able to understand previous group traffic. Forward secrecy requires that the key should be changed when a member leaves the group otherwise the leaving member will still be able to understand group traffic. The message containing the new key is called a key update. The mechanism by which the new key is generated and distributed is called *key management*. We explore the varieties of key management and survey the suggested protocols in the next section.

1.5 Key Management for Secure Multicast

There are two basic varieties of key management, key distribution and key agreement. With key distribution a central authority securely distributes the key to all members of the group. In key agreement all members contribute to agree upon a key. Key agreement is used in most distributed solutions that do not depend on a single controller to function. Such schemes are required for peer groups where no body can entirely trust any single authority like a forum of several competing companies. However for large groups such a scheme is infeasible as the computation and communication overhead will be very large. In such groups some sort of key distribution mechanism must be used.

From the perspective of centralization of the solution, the proposed protocols fall in three categories. Distributed solutions, centralized solutions, and decentralized ones [16].

1.5.1 Distributed Solutions

An n-party extension of the Diffie Hellman protocol can be used to agree upon a shared secret among n members of a group. However the resulting solution is only viable for reasonably sized and fairly static multicast groups. This is because key agreement takes reasonable time and communication among the members. The only advantage is that there is no dependence on any group controller and thus no single point of failure. Key agreement has been used in Cliques, Octopus Protocol, Conference Key Agreement, Distributed Hierarchical Binary Tree, Distributed One-way Function Tree, DH Binary Tree, and Distributed Flat Table protocols [16].

1.5.2 Centralized Solutions

For large dynamic groups, the key must be distributed by some authority rather than agreed among group members. This authority is called the group controller and is often the same as the one authenticating and authorizing the members. On every member join or leave operation a new key must be distributed to ensure forward and backward secrecy. Forward secrecy means that a revoked group member cannot understand further group communication and backward secrecy means that a new group member cannot understand previous group communication. It has been shown that updating keys on small regular intervals is more scalable than updating the key on every join and release. These are called batch updates and disassociate key updates from the dynamics of the group [17].

Key updates cannot be simply multicast as any eavesdropper on the multicast group will be able to get the key and henceforth all encryption is meaningless. If however the key updates are unicast to every recipient or a single packet is multicast containing the group key encrypted by each of the remaining member's public key, performance degrades proportionally to the size of the group. This scheme has been used in Group Key Management Protocol [18, 19], Dunigan and Cao's Group Key Management, Scalable extension of Group Key Management Protocol [20], and Secure Lock.

1.5.2.1 Hierarchical tree based schemes

An efficient method for key updates is by using key trees [21]. Key trees consist of a tree of keys with group members as nodes. Every group member knows all keys on the path from its node to the root. The key at the root is used as the group key. When a member is deleted all keys from that node to the root must change. The new values of these keys are multicast on the group encrypted with its children keys in the tree except for the key of the node that is deleted. This way a logarithmic sized message can be used for key update making a much scalable and efficient multicast group possible.

A logical key hierarchy (LKH) [21] is a hierarchical binary tree scheme where each node in the tree is a Key Encryption Key (KEK). Each leaf is attached to one member of the group. The key at the root is used as the Traffic Encryption Key (TEK) or the group secret. Group members receive and maintain the KEK in the leaf associated to them and every KEK in their path to the root node. For a balanced tree the keys held by each group member are logarithmic to the total size of the group.

When a member joins a group it is associated with a new leaf of the tree. All keys in the path from the new leaf to the root are compromised as the new member can use them to understand previous traffic (backward secrecy). Therefore all of these keys have to be changes. The keys are multicast in the group encrypted with both the children KEKs. This way a message of size 2 log n is needed to rekey the whole group. When a member leaves the group, all keys from its path to the root are compromised as the old member can understand future traffic with them (forward secrecy). This time the new keys are changed and sent in a multicast group encrypted with the children's KEKs except for the leaving member's KEK. This way every group member except the leaving member can recover all necessary keys. This is different from a joining member only in that the old member's new parent key is not sent encrypted with its key. An example of the scheme is given in the next section.

An improvement suggested in the hierarchical binary tree scheme was to use one way functions in the generation of keys [22, 23, 24]. In this scheme the keys are generated rather than attributed to nodes in the tree. Lets denote the KEK of left and right nodes as Kr and Kl and the one way function as g. The KEK of the parent node is given by f(g(Kr), g(Kl)) where f is a mixing function like XOR. Every member knows its KEK and the blinded KEKs of all sibling nodes on its path to the root. Blinded KEK means a KEK which has been passed through the one way function. Using this information and the formula given above, group members can generate all required keys.

When a member joins the group all keys on its path to the root are compromised. The blinded version of the new keys must be told to the sibling nodes at every level in the tree. The improved scheme uses a message size of log n instead of 2 log n. An example of the scheme is given in the next section.

Many improvements have been suggested in the key graph based protocols more importantly those that relate to efficiency [25, 26]. In particular it was shown that re-keying regularly instead of re-keying at every join and leave and dealing with these membership changes in a batch improves performance and removes its dependency on the dynamics of the group [17]. A better version is also proposed that uses Boolean function minimization techniques to find the smallest rekey message size possible [27]. Another improvement related to performance partitions the tree according to the temporal pattern of member joins and leaves [28]. Some improvements relate to distributing the hierarchical tree and handling it without a centralized authority [29, 30]. Some other researchers have considered the problem of reliability in re-keying [31, 32, 33, 34]. The centralized and distributed versions of flat table reduce server space requirements to log n as well [35].

1.5.2.2 Logical Key Hierarchy

In the hierarchical binary tree shown in Figure 5, K is the group secret or TEK known by all members of the group. K14 is a KEK known by nodes 1 through 4, K12 is known by 1 and 2 only while K1 is only known to member 1. The M1 through M8 are members of the group.

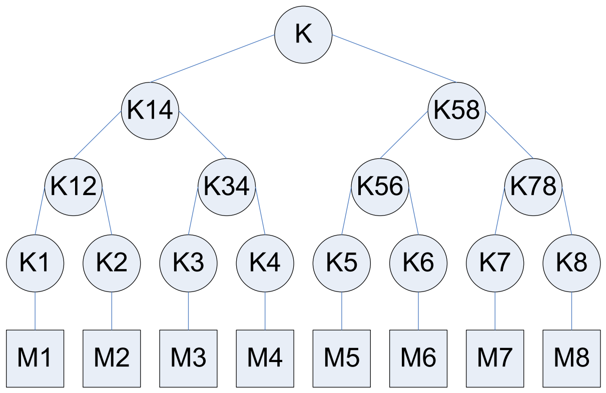

Figure 5 – A Hierarchical Binary Tree comprising of 8 members represented by rectangles and keys used by them represented by circles

If M6 has just joined the group and is given its leaf key K6 via secure unicast, then the rekey multicast contains the new K56 encrypted with K5 and K6, the new K58 encrypted with the new K56 and K78, the new group secret encrypted with K14 and the new K58. This way every member including the newcomer knows the group secret but the newcomer is unable to guess the previously used group key. M1-M4 receive it because they knew K14 and a copy of the new key is encrypted with K14. M5 and M6 know it because they knew K5 and K6 and therefore both were able to decrypt K56 and decrypt K58 using it and finally the new key as it was also encrypted with K58. M7 already knew K7 while M8 has just been told of K8 by unicast, therefore both of them are able to decrypt K78 and therefore are able to decrypt K58 and using it the group key.

If M6 leaves the group the rekey multicast packet will contain all keys as were in the joining case except that the new K56 will not be sent encrypted with K6 so M6 is unable to guess K56, and therefore K58 and therefore the group key. This way M6 is left out of the new tree and has no way to guess the new key used in the group.

1.5.2.3 One Way Function Trees

One way function trees use one way functions to arrive at keys in the tree. The following formula is used to calculate the keys in the tree, if g is the one way function and f is a mixing function like the XOR function.

$K_{parent} = f(\ g(K_{right}),\ g(K_{left})\ )$

The key when it is passed through the one way function is called a blinded key. Every host knows its key and blinded keys of all siblings on the path to the root. For example in Figure 5, M6 would know K6, g(K5), g(K78), g(K14). Using this information it can derive all keys on the path to the root and therefore the group key as follows.

K56 = f(g(K5), g(K6))

K58 = f(g(K56), g(K78))

K = f(g(K14), g(K58))

If M6 joins the group K6, K56, K58, and K must change. The blinded version of K6 i.e. g(K6) must be told to M5 so it is sent encrypted with K5. g(K56) must be told to M7 and M8 so it is sent encrypted with K78. g(K58) must be told to M1 through M4 so it is sent encrypted with K14. Observe that every key has to be encrypted only once as there is one sibling. It had to be encrypted twice in case of the simple logical key hierarchy as it was encrypted for both children rather than the single sibling.

1.5.3 Decentralized Solutions

Certain protocols have decentralized group management by introducing subgroup managers. This does not eliminate the single point of failure in the multicast group but it does improve performance by distributing the load in terms of time and space requirements of the single group controller. Scalable Multicast key distribution, Iolus, Dual encryption protocol, MARKS, Cipher sequences, Kronos, Intra domain group key management and Hydra are examples of decentralized solutions [16].

Iolus splits the large group into small groups with a group security agent (GSA) controlling each group. The Intra domain group key management protocol has a domain key distributor (DKD) and multiple area key distributors (AKD) and each area key distributor handles a subgroup. Similarly the other schemes partition the group and handle them over to static predefined subgroup controllers. The membership of subgroup controllers is not dynamic and subgroup controllers cannot have further subgroup controllers. Also the members are not transferred among controllers; they are statically bound to the controller they contacted. This static nature of the controllers requires that servers be pre-designated as controllers. There may be times when only one or few controllers may be bearing the whole load of the group as there is no mechanism to redistribute them.

Chapter 2
Proposed Scheme

This chapter highlights the problems in existing key management schemes for large dynamic groups and proposes an improved variant solving these deficiencies.

2.1 Problems in Existing Schemes

The most efficient key management schemes are based on hierarchical key trees. However all such schemes depend on a single controller to store keys for the whole group and generate key updates. Since a single controller is bearing all load of the group, performance deteriorates as group size increases. This is one weakness in hierarchical key tree based schemes. The second weakness addressed in this thesis is that the message size is logarithmic only when the key tree is balanced but a balanced tree is not guaranteed in any of these schemes. Therefore performance also deteriorates as tree gets out of balance which is frequently possible with large dynamic groups.

The following sections describe an improved scheme that ensures the balancing of the distributed tree, therefore a guarantee of logarithmic performance, and a distributed architecture that spreads the load of storing keys and generating key updates among a number of controllers.

2.1.1 Scope of Work

The work is limited to a decentralized key management scheme for large dynamic multicast groups. It does not cover authentication or authorization issues. It also does not cover any specific cryptographic protocols. Any such protocol can use this scheme to distribute its keys to the group members. The issue of fault tolerance in the distributed scheme is also out of the scope of this work.

2.2 Outline of the Scheme

There is a main controller at the root of the tree managing the whole group. Subgroup controllers can dynamically join and leave the tree. Subgroup controllers inform the root controller of their capacity when joining. The load is distributed in proportion to the registered capacities of these subgroup controllers.

An example key graph at an intermediate stage is shown in Figure 6. The rectangular boxes show the part of the tree controlled by a single controller. For example C18 controls two members and two subgroup controllers and maintains seven keys for the purpose (K18, K14, C58, C12, K34, K3, and K4). Controllers can be at any level in the tree and can have more subgroup controllers managing a part of their sub-tree.

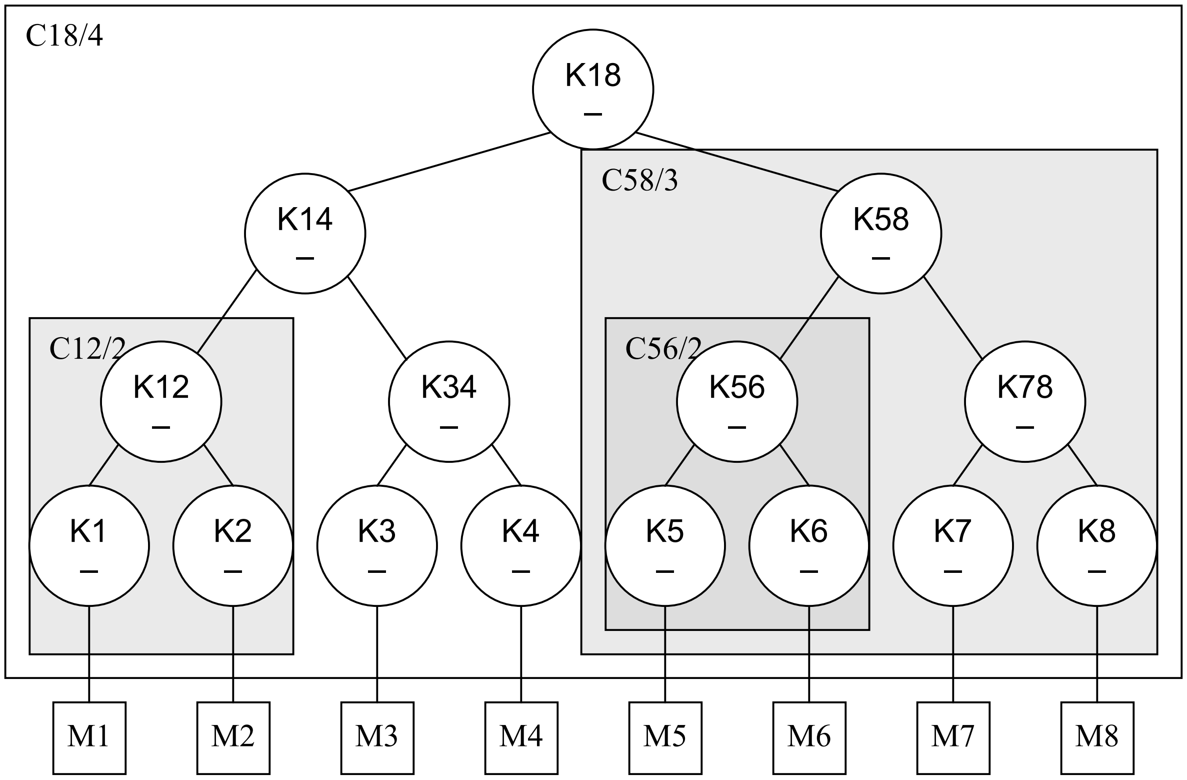

Figure 6 – A group controlled by 4 controllers

Controllers do not know the details of members and keys maintained by a child controller. Figure 7 shows how the root controller views the tree and the information maintained by it. It knows that the right side of K18 in the key tree is controlled by a controller with key C58 and its height is 3 but it knows no more about it. The '–' in the nodes show the balance. In this example all nodes are balanced.

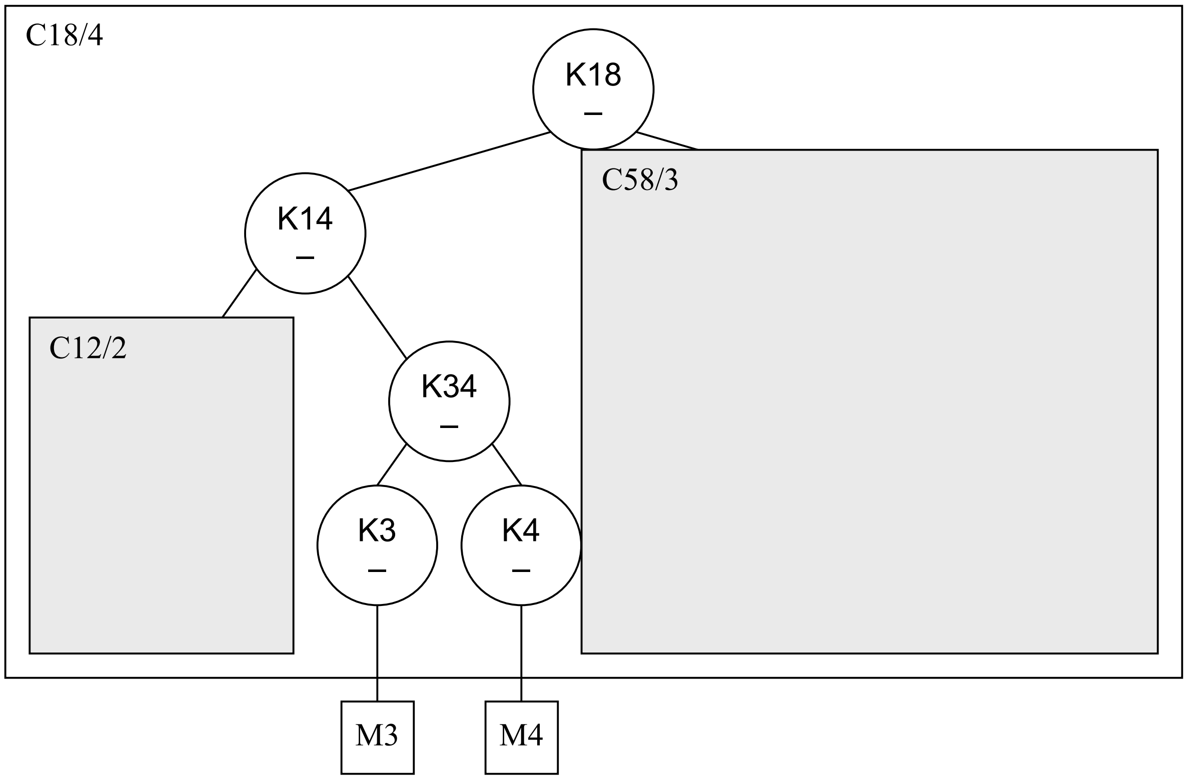

Figure 7 – A multi controller group as seen by the root controller

Controllers are not a node in the key tree and therefore controllers join and leave do not affect the height of the tree. They just manage and store a part of the actual key tree. They are however represented as a node in the tree of their parent controller with a different assigned key than actually used in the key hierarchy. This node is actually a box that hides the detail of the actual tree behind it.

The interesting thing is that the tree at each controller does not look balanced. Still the whole distributed tree remains balanced and the key updates performance is maximized.

When a new member or subgroup controller wants to join the group, the root controller assigns it to a parent controller or to itself. The designated controller then adds the member or the controller at a leaf node in its key graph. This may result in passing the member or subgroup controller to a different controller and thus changing the parent controller of the new member or controller.

The requirement for balancing is the same as used in AVL trees. The number of nodes in each sub-tree should not differ by more than one. The balancing at join works by the main controller forwarding the request to a controller selected in proportion to registered capacities after authentication. The controller tries to adjust the new member in the tree without increasing the height. However, if the new member cannot be added without increasing a level, the request is forwarded to the parent controller. The parent controller tries to adjust the member somewhere. If however the request is received back from the parent controller it must be satisfied even if it requires an increase in height.

The balancing at leave is more complicated. The immediate controller tries to balance its tree while maintaining the height of the tree. If however this is not possible, it must inform its parent that its height has decreased which itself may have to go through a rotation to satisfy the balancing criteria. This rotation may result in a change of duties among the controllers with a child controller now becoming the parent and the parent controller becoming a child.

This work suggests an improvement to make the key graph based key distribution algorithms decentralized with dynamic load balancing. It also maintains the key graph balanced which is distributed among a number of subgroup controllers. The balancing uses rotation that is similar to rotations as used in AVL trees.

The next section describes the messages sent among members and controllers while the remaining sections use these messages to describe the various algorithms involved in formation and maintenance of the distributed balanced tree.

2.3 Messages between Controllers and Members

The communication between the controllers and group members comprises of a few very simple messages. The possible messages accepted by root controller, controllers and group members are detailed in the following two sections. Here C stands for a normal controller; RC stands for the root controller while M stands for a group member.

2.3.1 CONTROLLERJOIN <CONTROLLER>, <CAPACITY>

This message is sent by a controller to the root controller when it wishes to join the group as a controller. The message contains the capacity in number of members the controller is willing to control.

2.3.2 CONTROLLERFORWARD <CONTROLLER>, <CAPACITY>

The message is sent by the root controller to a member controller which should act as parent controller for a new controller which just sent a CONTROLLERJOIN message to the root controller.

2.3.3 CONTROLLERLEAVE <CONTROLLER>, <MEMBERTREE>

This message is sent by a controller to its parent controller when it wishes to leave its responsibility as a controller in the group. This message contains the member tree controlled by the controller so that the parent controller can take over that part of the tree and inform members and controllers therein about the change of their parent controller.

2.3.4 CONTROLLERACCEPT <CONTROLLER>

This message is sent by the parent controller to a new controller which showed the intent to join the group as a controller. The message is sent from the controller assigned as parent by the root controller.

2.3.5 CONTROLLERREJECT

This message is sent by the root controller to a new controller which showed the intent to join the group as a controller if the new controller cannot be authorized to join the group as a controller. It is also sent to the new controller if the parent controller found no work for this controller or all of its members leave the group.

2.3.6 HIEGHTCHANGED <CONTROLLER>, <INCREASED>

This message is sent by a controller to its parent controller when its height increases or decreases. It contains a boolean parameter indicating whether it’s a height increase or a height decrease.

2.3.7 MEMBERJOIN <MEMBER>

This message is sent by a new member to the root controller when it wishes to join the group.

2.3.8 MEMBERFORWARD <MEMBER>, <MUSTJOIN>

The message is forwarded by the root controller to a selected controller after a member shows intent to join the group. The message may further be forwarded among controllers. The message contains a boolean parameter which dictates if the message shouldn’t be forwarded to a parent controller even if it results in an increase in height. The parameter is set to false in the initial message forwarded from the root controller. The algorithm section details its use.

2.3.9 MEMBERLEAVE <MEMBER>

This message is sent by an existing member to its assigned controller when the member wants to leave the group.

2.3.10 MEMBERACCEPT <CONTROLLER>

This message is sent by the controller who accepted a newly joined member to the member. This message is also sent by a parent controller to the members controlled by its leaving child controller.

2.3.11 MEMBERREJECT

This message is sent by the root controller to a new member who cannot be successfully authenticated.

2.3.12 ROTATION <NODE>, <NODE>

This message is multicast by a controller to the group when a rotation occurs in the part of the group controlled by it. The node demoted in level and the one promoted in level are part of the message. The message also contains the two keys required by members underneath encrypted by appropriate KEKs.

2.4 Algorithms for Controller and Member Join and Leave

The algorithms for the joining and leaving of controllers are simple. However the joining and leaving operations for members must cater for the balancing of the distributed tree. The requirement for balancing is the same as used in AVL tree. The height of each sub-tree should not differ by more than one. However the balancing is simpler than AVL trees in our case as the new node can be inserted in either of the sub-trees. The proposed tree is a balanced tree but not a search tree.

2.4.1 Controller Joins

Whenever a new controller wants to join and handle a part of the tree it sends a CONTROLLERJOIN message to the root controller containing its approximate capacity, i.e. the number of members it can easily accommodate.

The root controller verifies the authenticity of this controller. In case the new controller cannot successfully authenticate with the root controller, a CONTROLLERREJECT message is sent to it by the root controller.

In case of successful authentication, the root controller chooses a controller to act as parent for this controller. This parent controller is chosen in proportion to registered capacities of subgroup controllers. An improvement in the current scheme can choose the most overloaded controller and then give few of its members to the new controller.

The root controller sends a CONTROLLERFORWARD message containing the new controller and its capacity to this new controller designated as parent controller of the new controller.

The designated parent controller finds a member node in its tree node. This member node should not be the root of the designated parent controller's controller sub-tree. This may happen when there is just one member controller by this controller. If such a node is not available, the new controller is sent a CONTROLLERREJECT message.

The designated controller sends a CONTROLLERACCEPT to the newly joined controller. The information of the leaf member node select above is also added to the CONTROLLERACCEPT message. The newly joined controller takes over the control of this member. It communicates with this parent controller about group activities until this controller decides to leave the group or the root controller decides to abolish the group.

The parent controller can either do a simple search on a random path in its tree to find a leaf node for the new controller (logarithmic time) or it can do an exhaustive search of the whole tree managed by it (linear time) to find a leaf node. This is an option with the controller and can by dynamically decided. For example the controller can switch to exhaustive search if it is highly loaded, so that the new controller can share its load. The detailed algorithm in this document does not perform an exhaustive search. It tries on a random path. The request is forwarded to a child controller if one occurs in the path. If the terminating node is not the root node of a controller it is made the root of the new controller, otherwise the new controller's request is denied.

A controller cannot exist if there is no member for it to control. This is why an initial member is given to it in the acceptance message. Similarly when the last member leaves, and the controller informs its parent in a HEIGHTCHANGED message that its height is zero now, it is sent a CONTROLLERREJECT message meaning that you are no longer needed at this point in the tree. The controller may join the tree again by sending the root controller a CONTROLLERJOIN message and the root placing it in another part of the tree.

2.4.2 Controller Leaves

When the controller wants to leave the group management task it sends a CONTROLLERLEAVE message to its parent controller. It then transfers all information of its members to the parent controller. This information contains the sub-tree structure, height information of its child controllers, balance information and keys in the whole tree. All members and child controllers of the leaving controller are informed of this change by the new controller with a CONTROLLERACCEPT message.

2.4.3 Member Joins

A member requests to join the group by sending a MEMBERJOIN message to the root controller. An improvement can be made by changing this requirement to sending the MEMBERJOIN message to "any" controller. The only issue in that case is the discovery of a controller other than root controller. The remaining algorithm will run identically but this will further decrease the load of the root controller.

The new member receives a MEMBERREJECT message from the root controller if it could not authenticate properly. If it did authenticate the root controller sends a MEMBERFORWARD message with the MUSTJOIN parameter set to false to a parent controller chosen in proportion to registered capacities. This designated parent controller may not be the actual controller accepting this member as the algorithm allows passing the new member between controllers.

The root controller itself is a possible candidate in the controller selection. So it may happen that the new member is just assigned to the root controller itself. This is necessary as there will be no other controller when the group starts up and joining of controllers is not mandatory for working of the group.

The chosen controller starts from the root node of the sub-tree it's managing. If the height of one sub-tree is less than the other, the controller tries to insert the new member on that side. If the height of both sub-trees is equal one of them is chosen randomly.

If a subgroup controller is encountered the MEMBERFORWARD message is sent to it. MUSTJOIN is set to the same value as received by the parent controller except when it the subgroup controller is in a smaller sub-tree of the parent controller i.e. when one or more decisions based on height have been taken while traversing from root node of the controller till this point in the tree. In this case when the subgroup controller lies on the smaller side of the tree, MUSTJOIN is always set to true.

If the new member cannot be inserted without height increase and MUSTJOIN was false the MEMBERFORWARD message is sent to the parent controller with MUSTJOIN again set to false.

The parent controller repeats the same algorithm except that it does not send the member back to the same controller it received the MEMBERFORWARD message from to avoid indefinite loop. It however will forward it back if it is to send it with MUSTJOIN set to true because of some height based decisions in its tree. In this case there is no possibility of the child controller sending it back and therefore no chance of getting into an indefinite loop.

If the parent controller is the root controller and it receives a MEMBERFORWARD message from a child controller it changes MUSTJOIN to true. This is because the new member must be inserted and the decision cannot be deferred further up as this is the top of the tree.

If the new member cannot be inserted without height increase and MUSTJOIN was true either because of moving in a smaller sub-tree or because of restarting from root controller, the new member is inserted causing an increase in height. The parent controller is informed of an actual height increase by a HEIGHTCHANGED message so that it can update its record of sub-tree heights. If this change in height of the subgroup controller causes a change in height of the parent controller, the parent will itself need to generate a HEIGHTCHANGED message for its parent further up. However if the subgroup controller whose height increased was in a smaller sub-tree of the parent the parent will just update its record and not inform the parent further up because the overall height has not changed.

The crux of the algorithm is to try inserting the member without height increase at a controller, at its parent controller, up to the root. If all fails, the process again starts down from the root with height increase allowed. The MUSTJOIN parameter dictates whether height increase is allowed or not.

The controller that finally accepts this member sends a MEMBERACCEPT message to the new member. The member can also receive a MEMBERACCEPT message during the session when there is a new parent controller because of the existing parent controller leaving the group.

2.4.4 Member Leaves

When a member wishes to leave the group it sends a MEMBERLEAVE message to its parent controller. The parent controller removes the member from its sub tree. If this removal disturbs the balancing property it performs rotations to adjust it. Only single rotations are required since there is no ordering requirement among the members and any member can be placed anywhere.

If a rotation decreases the height of the sub tree managed by a controller it must be informed to the parent controller by a HEIGHTCHANGED message. This change in height of the subgroup controller can cause parent controller's balance to disrupt. The parent controller will then undergo rotations. If inform its parent further up with another HEIGHTCHANGED message if its overall height also changes. The issues of rotation are discussed in detail the next section. The root controller need not generate any HEIGHTCHANGED message since it is at the top of the tree and need not inform any other controller about changes in its height.

2.5 Rotation

A rotation is required when one sub tree has a height two more than the other sub tree. In the proposed scheme this can only occur when a member leaves and not when a member joins. This is because a new member is added in the less high tree if the heights are different. So it is not possible to create a height difference of two at member join. When rotation is required in an AVL tree with similar balancing requirements, there are four possible cases. However there is just one case in our scheme as no ordering of elements is required. The left and right sub trees are just two sub trees not in any particular order.

To describe this only case we take R as root of tree, H as root of higher sub tree and S as root of shorter sub tree, HH as higher sub tree of H and SH as shorter sub tree of H as shown in Figure 8. Height of H and S differs by 2 while height of HH and SH differs by 0 or 1. We transform the tree and make H the new root with HH as one child with its sub trees attached and R as the other. S and SH become the new children of R with their sub trees attached. The transformed tree after rotation can be one less in height than the original.

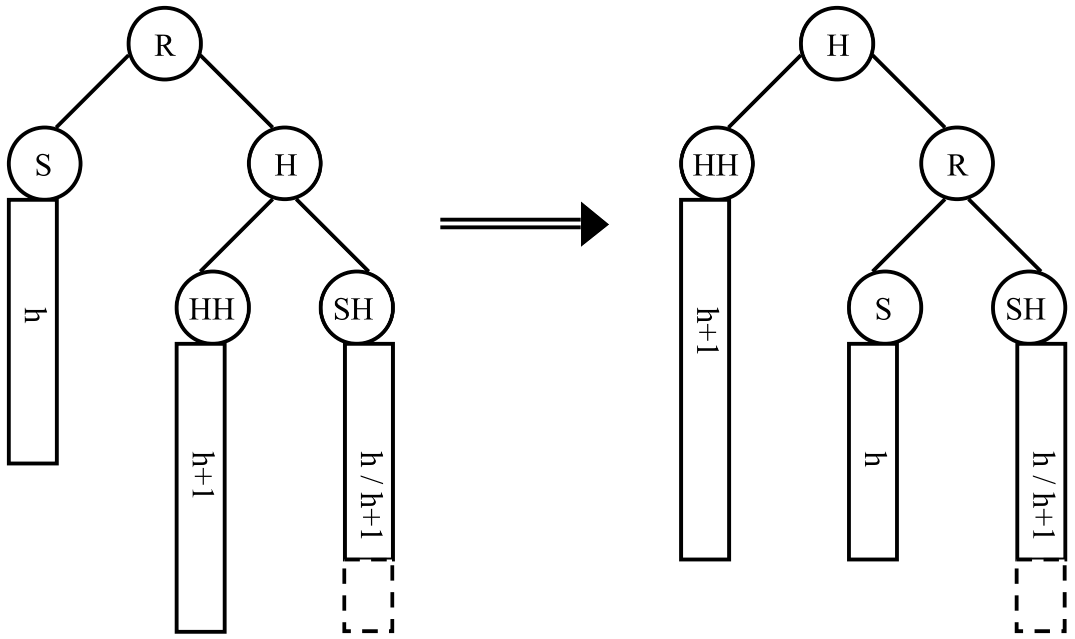

Figure 8 – Steps performed when a rotation is done in the Key Tree

2.6 Pseudocode for Algorithms

The following sections contain the pseudocode for algorithms implemented at the controllers.

```
// the root controller will initialize and then wait for messages from controllers
// and members
RootControllerMain()
Begin
    Initialize();
End
// when a subgroup controller starts it needs to contact the root controller for
// joining the group and for informing it of its capacity, after which it will wait for
// an acceptance message from it
ControllerMain( RootController, Capacity as Integer )
Begin
    Initialize();
    Send( RootController, CONTROLLERJOIN );
End
```

```
// the root controller receives a message from a prospective controller
ControllerJoin( Controller, Capacity as Integer)
Begin
    // if the controller cannot authenticate its entry will be rejected
    If( Authenticate( Controller ))
    Begin
        // the request of new controller is forwarded to a proportionately
        // selected existing controller and the new controller is added to the
        // list of controllers
        Var Parent as Integer;
        Parent = ProportionateChoose( ControllerList );
        Send( Parent, CONTROLLERFORWARD, Controller, Capacity );
        AddToList( Controller, ControllerList );
    End
    Else
        Send( Controller, CONTROLLERREJECT );
End

// a controller is given to the receiving controller to make it a child
ControllerForward( Controller, Capacity as Integer)
Begin
    Var Accepted as Boolean;
    ControllerNodeJoin( GetRoot(), Controller );
End
// try to join the controller at this node if possible
ControllerNodeJoin( N as Node, NewController as Integer )
Begin
    Var LeftChild, RightChild as Node;
    LeftChild := GetLeftChild( N );
    RightChild := GetRightChild( N );
    NodeJoin := 0;
```

```
        // there is a child controller at this node so forward the request
        If( IsControllerNode( N ))
            Send( GetController( N ), CONTROLLERFORWARD, NewController );
        // there is a member node available
        Else If( IsMemberNode( N ))
        Begin
            // if it is the root of this sub-tree reject the request
            If( N = GetRoot() )
                Send( Controller, CONTROLLERREJECT )
            Else
            // turn this node into a controller node and send the member therein
            // to the new controller via the CONTROLLERACCEPT message
            Begin
                SetAsControllerNode( N );
                Send( Controller, CONTROLLERACCEPT, GetMember(N) );
            End
        End
        // if this is a normal node move towards smaller side if one is smaller
        Else If( RandomlyChoose( LeftChild, RightChild ))
            NodeJoin( LeftChild, NewController );
        Else
            NodeJoin( RightChild, NewController );
    End
    // when a controller receives an acceptance message it receives the initial root
    // member so make it into the root node
    ControllerAccept( Member as Integer )
    Begin
        RootNode := MakeNode( Member )
    End
```

```
// if a controller's entry is rejected it just terminates
ControllerReject()
Begin
    Terminate()
End

// when a controller receives a leave message from a subgroup controller it
// receives the sub-tree it was managing
ControllerLeave( Controller as Integer, Members as Tree )
Begin
    Var Parent, ControllerNode as Node;
    // find where in the tree was this subgroup controller attached
    ControllerNode := FindControllerNode( Controller );
    Parent := FindParent( ControllerNode );
    // remove the controller node and attach the whole member tree there
    If( Parent->left = ControllerNode )
        Parent->left = GetRoot( Members );
    Else
        Parent->right = GetRoot( Members );
    // notify the members of change in parent controller
    For each M of Members
        Send( M, MEMBERACCEPT )
End
// the root controller receives a message from a new member
MemberJoin( Member as Integer )
Begin
    // if the member cannot authenticate its entry will be rejected
    If( Authenticate( Member ))
    Begin
        // the request of new member is forwarded to a proportionately
        // selected existing controller
```

```
                Var Parent as Integer;
                Parent = ProportionateChoose( ControllerList );
                Send( Parent, MEMBERFORWARD, Member );
        End
        Else
                Send( Member, MEMBERREJECT );
End


// when a controller receives a request of a new member to join the group it finds
// a place for it and if its insertion causes change in height the parent is informed
MemberForward( Member as Integer, Mustjoin as Boolean )
Begin
        Var NewNode as Node;
        Var Result as Integer;
        Result := NodeJoin( RootNode, Member, Mustjoin );
        If( Result = 1 )
                Send( ParentController, HEIGHTCHANGED, 1 );
        Else if( Result = -1 )
                Send( ParentController, MEMBERFORWARD, Member );
End


// tries to insert the new member by splitting this node if possible
NodeJoin( N as Node, Member as Integer, Mustjoin as Boolean )
Begin
        Var LeftChild, RightChild as Node;
        LeftChild := GetLeftChild( N );
        RightChild := GetRightChild( N );
        NodeJoin := 0;
        // if a subgroup controller is encountered the request is forwarded
        If( IsControllerNode( N ))
            Send( GetController( N ), MEMBERFORWARD, Member,
Mustjoin );
        // if member node is encountered
        Else If( IsMemberNode( N ))
        Begin
            // split this node if height increase allowed
            If( Mustjoin )
            Begin
                SetRightChild( GetMember( N ));
                SetLeftChild( Member );
                SetAsNonMemberNode( N );
                NodeJoin := 1;
            End
            Else
                Nodejoin := -1;
        End
        // try to move towards the smaller sub-tree
        Else If( GetHeight( LeftChild ) < GetHeight( RightChild ))
            NodeJoin( LeftChild, NewNode, TRUE );
        Else If( GetHeight( LeftChild ) > GetHeight( RightChild ))
            NodeJoin( RightChild, NewNode, TRUE );
        Else If( RandomlyChoose( RightChild, LeftChild ))
            NodeJoin := NodeJoin( RightChild, NewNode, Mustjoin );
        Else
            NodeJoin := NodeJoin( LeftChild, NewNode, TRUE );
    End
```

// when a controller receives a member leave message it finds its node and removes from the tree

```
MemberLeave( Member as Integer )
Begin
    Var N, N2 as Node;
    N := FindMemberNode( Member );
    N2 := FindParent( FindParent( N ));
    N := FindSiblingNode( N );
    MakeParent( N2, N );
    // adjusts balance upwards
    While( N )
    Begin
        // if node was in balance its balance changes and process stops
        If( Balance( N ) = 0 )
        Begin
            AdjustBalance( N );
            Break;
        End
        // if larger sub tree decreases the balance is 0 other rotation occurs
        Else If( BalanceTowardsDeletion( N ))
            SetBalance( N, 0 );
        Else
            Rotate( N );
        N = GetParent( N );
    End
End
```

2.7 Key Updates

When sending a key update, the root controller considers all subgroup controllers as normal members. The sub controllers therefore have one key attributed to them by the root controller. The key update generated by the root has updated keys encrypted by their parent KEKs.

Subgroup controllers use another key, generated by them, as the KEK of the node they designate in the entire distributed tree. Subgroup controllers generate key updates for their part of the tree.

This way the load of generating key updates and maintaining member information for generating them is distributed among all the controllers. The process is detailed for both LKH and OFT schemes.

Key updates are done at regular intervals to avoid excessive communication among controllers regarding members joining and leaving. The only communication that occurs is when a member cannot be inserted or removed without a change in height.

2.7.1 Logical Key Hierarchy

A particular controller knows all keys on its path to the root (since it is a normal member of the parent controller) and all keys underneath it (since it is a controller for them). Like a normal member it has a particular key assigned to it by the parent controller. Also like a node in the key tree it has a particular key which is the KEK in the entire key tree at its place.

The controllers are special in that they can request their parent to change the keys on their path to the root for no reason. They will do this whenever there is any member join or leave in the last key update period. Referring to the original hierarchical key distribution algorithms, a member join or leave requires all keys on its path to the root be changed. The immediate parent will update all keys from leaving or joining member to its root node, but it should have a way to request the parent that keys need to be changed further up. For the parent there was no change in the tree as the child controller neither left nor joined but something happened in the child controller’s tree that the parent did not know.

This means when a new member joins its sub tree, the controller is able to provide it with all the keys it needs. The new member needs all keys on its path to the root and the controller has all of these keys.

When the root controller does a key update, it considers the controller as a normal member node. The controller after receiving a key update, generates it own key update message which has updated keys of its sub tree. This key update does not repeat anything from the parent's update.

The root controller stamps the key updates with a sequence number and the sub controllers use the same sequence number in their key updates. Combining both key updates, members get every key they need except when the parent KEK of the controller has changed. This is because the changed key is sent encrypted by the private key of the controller which is not known to the members below it. This has to be re-encrypted in the key update of the controller by the KEK it represents and only then the members beneath it can understand the changed key.

Note that in this whole scheme no extra messaging or overhead is introduced. The total size of key updates is the same as the single large key update in case of no controllers. The member combining the key updates of same sequence number gets the same information as in the case of no controllers without any extra overhead.

We take the example of the tree in Figure 6. The root controller stores K18, K14, K58, K12, K34, K3, and K4 and these keys will be in the key updates of the root controller. The value of K58 and K12 as stored by the root controller is not the actual value as used in the whole tree. The controller with managing K12 triggers its own key update message after receiving the root controller's message containing keys K12, K1, and K2 and similarly the controller managing K58 will generate its own message. After seeing this message the controller managing K56 generates its own message containing K56, K5, and K6. All the update messages have the same sequence number.

2.7.2 One way Function Trees

New members are assigned random keys in this scheme and parent keys are generated from them. Therefore our modified scheme should allow that child sub-trees generate their keys and key updates first followed by parent trees until the final update from the root. This order is opposite to the order in the LKH scheme.

The controllers like the LKH scheme can inform the root to change the keys on their path to the root. However they need to specify that a particular key is assigned to them. This would be the key generated by the sub-tree of the controller. In the LKH case the key of the controller as known by the parent controller was different from that as known by the members beneath it. The controller had to re-encrypt whenever the parent controller used it to encrypt some information. However since the keys are generated rather than attributed in OFT, the controller must inform its parent about the generated key that must be used.

The change scheme is that the root controller generates an empty key update with a sequence number at a periodic interval. This is the start of a key update. All controllers generate a similar empty update after receiving this with the same sequence number.

Those controllers that have no further sub controller generate a proper key update using the same sequence number. Such controllers can also combine this message with their empty message to save time. This key update can be empty if there is no change in the tree.

Those controllers that have further sub controllers beneath them generate a key update after receiving key updates from every subgroup controller.

This whole process takes a little longer than the original scheme but the group communication is not disturbed and continues using the old keys until the second update from the root controller is received after which group traffic must use the new key. The total size of messages transmitted is also the same. The only overhead is C empty key updates from the C controllers. The advantage is that after the second message from root everybody knows that the new key must be used however in LKH scheme key update took half the time but there was no concise point after which new key must be used. However no packets are lost in either of the schemes.

Considering the tree in Figure 6 on page 22, the root controller sends an empty update message followed by empty updates by K12 and K58 and then from K56. After this K12 generates a proper update containing K12, K1, and K2, and K56 generates containing K56, K5, and K6. After K56 sent its message K58 can generate its message containing K58, K56, K78, K7, and K8. After receiving both K58 and K12 messages the root generates its second update containing K18, K14, K58, K12, K34, K3, and K4 and from this point onwards the new value of K18 is used in the group. This scheme is slightly better than the previous in terms of synchronization as all members exactly know when the new key will be used and slightly worse as there were empty messages to be sent.

2.7.3 Optimizations in Key Updates

All suggested improvements in key updates including key update minimization using Boolean functions can be done locally by the controller implementation at that node. Similarly other optimization like replacing leaving members with joining members and only run the algorithm on the remaining surplus of leaving or joining members. This would improve the efficiency and can also be done locally at the controller.

Chapter 3 Analysis and Simulation Results

The performance of the given scheme is analyzed in terms of time and space requirements and a comparison is given below with the centralized schemes. Simulation is done for those parameters which can not be analytically compared.

3.1 Analytic Comparison with simple LKH and OFT Schemes

We compare a number of parameters of the proposed logical key hierarchy scheme and one way function trees with the original centralized logical key hierarchy scheme and one way function trees. Since our scheme is intrinsically dependant on regular key updates instead of re-keying at every member join and leave, we compare only with centralized schemes having key updates. Key update has proved to provide a scalable solution for large dynamic groups. Processing at key update depends on the number of joins and leaves that have occurred since last key update. To make an analytic comparison possible we consider that a single join has occurred since last key update. More joins and leaves within a key update interval will have similar impact on all schemes.

We show here that for a balanced tree the proposed schemes does not cause any overhead for the members, slight overhead in multicast message size, however the total storage and processing at controller is the same as before but is now distributed among a large number of controllers. With this slight overhead the scheme is able to ensure a balanced tree and thus ensure logarithmic performance.

	LKH	OFT	Proposed LKH	Proposed OFT
Multicast message size for key update	(2d-1)K	(d+1)K	(2d-1+c)K	(d+1+c)K
Unicast message size at join	(d+1)K	(d+1)K	(d+1)K	(d+1)K
Size of inter controller unicast messages at join	–	–	none to 2Ic	none to 2Ic
Storage at controller	(2n-1)K	(2n-1)K	(2n-1+C)K distributed	(2n-1+C)K distributed
Storage at group member	(d+1)K	(d+1)K	(d+1)K	(d+1)K
Join processing at controller	dH + (d+1)E	dH + (d+1)E	dH + (d+1)E	dH + (d+1)E
Join processing at member	(d+1)D	(d+1)D + d(H+X)	(d+1)D	(d+1)D + d(H+X)
Leave processing at controller	–	–	– when no rotations	– when no rotations
Key update processing at controller	(2d-1)E	(d+1)E + d(X+H)	(2d-1+C)E distributed	(d+1+C)E + CD + d(X+H) distributed
Key update processing at member	(d+1)D	(d+1)D	(d+1)D	(d+1)D

n	Number of members in the group
I	Number of bits in member id
a	Degree of the tree
d	Height of the tree (for a balanced tree $d = \log_a n$)
K	size of a key in bits
H	height of the tree under a particular sub controller
C	Total number of controllers
c	Number of controllers in the path to the root

Table 1 – Comparison of Proposed scheme with existing schemes

3.2 Overhead for Distribution and its advantage

The other two noticeable parameters changed are the storage at controllers and processing at controllers which is also slightly increased but again in the same small proportion. However the two quantities are now the total storage of all controllers and the total processing requirement of all controllers and with more controllers, this storage and processing cost is distributed which is the main aim of a distributed algorithm.

To ensure maximum balancing of work load among controllers, controllers can optionally send updates to the root controllers about their current load and two new messages can be introduced that increase and decrease a controller's level in the tree thus increasing or decreasing its work load. However the current proposal does not include this and the current method of work distribution is solely based on the root controller distributing new members in proportion to registered capacities.

3.3 Overhead for Balancing and its advantage

We observe that the only parameter affected is the multicast message size and a new parameter of inter controller messages is introduced. However since the number of controllers is a reasonably small number as compared to the number of members in the group, the overhead is very small. However this overhead can only be compared when both schemes with a balanced tree are considered i.e. the same value of tree depth (d) for both schemes. As and when the original scheme's tree is deformed the value of d increases and there is no guarantee or cap on the value of d. However in the proposed scheme balancing is ensured and the value of d will remain logarithmic to the size of the group thus ensuring better performance in cases where the original scheme's tree deformed and slight overhead when the original scheme's tree remained almost balanced.

This analysis does not consider the case of rotations which are a major part of this proposal as they keep the tree balanced. The case of rotations is discussed in the following section.

3.4 Extra overhead for Balancing in case of Rotation

In the above comparison it is considered that a single join has occurred since last key update. If a leave has occurred since the last key update, there is no difference, except when it causes a rotation. Rotation at member leave waits for the key update. One rotation will result in one fixed size multicast message from which the effected members and subgroup controllers can figure out the change in the keys on their path to the root. This message contains three keys as explained in the sections on messages for the proposed protocol.

We expect that rotations are member leave are infrequent and in most cases of rotation there will be only one rotation in the tree. We show this with a simulation of a large number of members dynamically joining and leaving and observe the number of rotations. We will enumerate members that caused none, one, two, and three rotations against the size of the group.

3.4.1 Simulation Technique

Controllers are not considered as proper nodes in the key tree. A controller just controls a part of the tree and if the controller is not there that part is controlled by the parent controller. Therefore adding or removing controller do not affect the shape of the tree and thus do not affect how many rotations take place while balancing the tree.

Therefore we build the structure of the group, i.e. the member nodes and KEK nodes and add and remove member node like they would in a real group and calculate the number of rotations. This means there is no real simulation of key management rather a simulation of group structure and the effect of member joins and member leaves on the structure. This is enough because all we need from simulation is the number of rotations in the group structure as all other parameters can be analytically compared.

We will take random formation of the group and random joins and leaves of members in the system to establish the results. We will arrive at the average increase in size of multicast messages using the proposed scheme.

3.4.2 Rotations when all members are removed from a tree

Our first experiment is for the number of members that caused rotation, when all members were removed from a given sized group. The table entries list the number of members that caused no rotation, one rotation, two rotations, or three rotations for the mentioned group sizes.

Group size	No Rotation	One Rotations	Two Rotations	Three Rotations
100	79	21	0	0
1000	780	208	11	1
10000	7778	2092	129	1
100000	77410	21103	1470	17
1000000	775070	210251	14496	181 + 2 times four rotations

Table 2 – Rotations when all members are removed from a tree

Our actual overhead is the total number of rotations, since each rotation causes a multicast message to the group. The above figures show an average of 24% rotations for different group sizes.

The above experiment is less realistic as not all members leave the group immediately in reality. The next experiment shows that the above figures are valid for dynamic groups as well.

3.4.3 Rotations when members leave from a dynamic group

This experiment takes a starting group of the given size and randomly performs additions and removal of members until the given number of removals has been done. For example for the first data a group size of 100 was taken and random insertions and removal was done until 100 members were removed. Note that in this case the ending group won't be empty, rather it would be of similar size as the starting group. This is the behavior that is expected from a real dynamic group. The following table shows the same statistics as were shown for the last experiment.

Group size	No Rotation	One Rotations	Two Rotations	Three Rotations
100	79	20	1	0
1000	773	221	6	0
10000	7692	2213	93	2
100000	77441	20535	2022	2
1000000	759008	230774	10142	76

Table 3 – Rotations when members are removed from a dynamic tree

The above tables show that the figure of 24% rotations is valid for a dynamic group as well. We expect this number to drop significantly if joining members take the place of leaving members on key updates and the algorithms are only run for surplus members.

3.4.4 Advantage of Balancing with combined overhead

Although an actual calculation of overhead is very difficult in case of key updates, we take the case of a single member removal to give a quantitative comparison. Remember that in case of joins, no rotations are involved. Also in case of suggested scheme of joining members replacing leaving members, no rotations are involved. Rotations are only involved when there are only leaving members or more leaving member than joining a particular controller and even in these cases only 24% of the time as shown above.

In case of single member leaving the group and proposed LKH scheme, the size of key update is (2d-1+c)K instead of (2d-1)K of standard LKH as the extra factor of c is introduced which is the number of intervening controllers. This extra factor was introduced because of distribution and the more we increase distribution of load i.e. more subgroup controllers this factor will increase. However this factor should be much less than the depth of the tree (d) for a normal group. For example a large group with depth of around 10 may have 2-3 subgroup controllers on a particular path to the root.

If leave causes a single rotation this figure changes to (2d-1+c+2)K where two of the three keys involved in the rotation are also updated (the other one was already included in the path to the root and is included in the key update) and everyone using them informed of their rotation. Two extra keys are sent in case of a single rotation and four extra keys sent in case of two rotations and so on. On average extra keys are sent 24% of the time so the average becomes (2d-1+c+0.24*2)K or (2d-1+c+0.48)K. Considering that a large group would be 10-15 levels deep (d=10) with 2-3 controller on the path (c=2), this addition only worsens the average by 2.3%. Given that the scheme ensures that depth of the tree will remain logarithmic to the size of the group this overhead is acceptable as if the tree gets out of balance the overhead is much more than that. Compare that the overhead of our scheme is 0.48K per update and the overhead of a single level increase in height of the tree of 2K per update. This overhead of 2.3% is only the overhead of balancing and not of distribution. The overhead of distribution was the factor of c keys per update and the overhead of balancing is the factor of 0.48 keys per update. Similar calculation shows that there is a small overhead of balancing for OFT scheme.

3.5 Overheads and Advantages Summarized

Multicast message size increased from (2d-1)K to (2d-1+c+0.48)K for LKH and from (d+1)K to (d+1+c+0.48)K for OFT with the advantage that the tree is guaranteed to be balanced or in other words the depth of the tree (d) is guaranteed to be logarithmic to the size of the group (n).

Storage at the single controller in original scheme was (2n-1)K while it is (2n-1+C)K distributed over all the controllers in the proposed scheme. Similarly key update processing is (2d-1)E in LKD and (2d-1+C)E in proposed LKH distributed among all controllers. Similarly for OFT the processing at the single controller was (d+1)E + d(X+H) while in proposed OFT it is (d+1+C)E + CD + d(X+H) distributed among all controllers. The load is currently distributed in approximation however perfect load balancing is possible with a little extra communication among controllers as suggested in future directions in the next section.

Chapter 4 Conclusion

Secure multicast groups require key distribution mechanisms such that all current members of the group can receive the key and thereafter comprehend group traffic. As a new key need to be generated regularly, a simple distribution scheme is a large overhead. Therefore new methods were introduced among which those based on key graphs have become popular in research areas because of the logarithmic time performance.

This work improves upon the tree based algorithms to make them decentralized by introducing subgroup controllers that can dynamically join and leave the group and take responsibility for a number of group member according to their capacity. Additionally the improvements ensure that the tree remains balanced so that logarithmic time performance is ensured. Thus the proposed scheme forms a distributed balanced key tree among the group controllers.

Comparing the proposed scheme with a balanced tree of the original scheme a small overhead is discovered however the performance is better than the original scheme for a deformed tree. The overhead of balancing is less than the overhead of a single level increase in height of the tree. The remaining overhead comes due to subgroup controllers that make the load distribution possible.

4.1 Future directions

Further optimizations of the proposed scheme can be done as hinted at certain places in the thesis. Boolean function optimization can be applied locally on each controller. Initial controller selection can be improved; initial member assignments to the controller can be improved. On key updates joining members can replace leaving members and thus not require rotations. Further experimental comparison and an optimization of the messages between controllers and members may also be possible. Controller demotion and promotion i.e. a change in their level can be made possible by a couple of new messages among controllers and with the two messages and a message to inform the root controller about a controller's current load can make perfect load balancing among controllers possible.

References

[1] W. Stallings, *Cryptography and Network Security – Principles and Practice,* Prentice Hall, 1998.

[2] S. Kent and R. Atkinson, "Security Architecture for IP," IETF RFC 2401, November 1998, www.rfc-editor.org/rfc/rfc2401.txt.

[3] S. Kent and R. Atkinson, "IP Encapsulating Security Payload (ESP)," IETF RFC 2406, November 1998, www.rfc-editor.org/rfc/rfc2406.txt.

[4] S. Kent and R. Atkinson, "IP Authentication Header," IETF RFC 2402, November 1998, www.rfc-editor.org/rfc/rfc2402.txt.

[5] T. Jinmei, "Implementation and deployment of IPv6 Multicasting," *Proc. 10th Ann. Internet Soc. Conf. Japan* (INET 2000), July 2000, http://www.isoc.org/isoc/conferences/inet/00/cdproceedings/1c/1c_2.htm

[6] R. Burk, M. Bligh, and T. Lee, *TCP/IP Blueprints*, Techmedia, 1998.

[7] B. Cain, S. Deering, I. Kouvelas, B. Fenner, and A. Thyagarajan, "Internet Group Management Protocol Version 3," IETF RFC 3376, October 2002, www.rfc-editor.org/rfc/rfc3376.txt.

[8] M. Capan, "Survey of Different Multicast Routing Protocols," *MIPRO International Convention '98*, June 1998, p. 277.

[9] T. Maufer, *Deploying IP Multicast in the Enterprise,* Prentice Hall, 1998.

[10] K. Savetz, N. Randall, and Y. Lepage, *MBONE: Multicasting Tomorrow's Internet*, Hungry Minds Inc, 1996.

[11] T. Hardjono and G. Tsudik, "IP Multicast Security: Issues and Directions," *Annales de Telecom*, July-August 2000, pp 324-340.

[12] C. Diot, B. N. Levine, B. Lyles, H. Kassem, D. Balensiefen, “Deployment Issues for the IP Multicast Service and Architecture,” *IEEE Network magazine special issue on Multicasting*, January/February 2000, pp. 78 – 88.

[13] D. Wallner et al., “Key Management for Multicast: Issues and Architectures,” IETF RFC 2627, June 1999, www.rfc-editor.org/rfc/rfc2627.txt.

[14] R. Canetti, “A taxonomy of multicast security issues,” Internet draft, work in progress, April 1999.

[15] T. Hardjono, “Secure IP Multicast: Problem areas, Framework, and Building Blocks,” Internet draft, work in progress, March 2001

[16] S. Rafaeli and D. Hutchison, “A Survey of Key Management for Secure Group Communication,” *ACM Computing Surveys*, vol 35, no. 3, Sep. 2003, pp 309 – 329.

[17] X. Li, Y. Yang, M. Gouda, and S. Lam, “Batch Rekeying for Secure Group Communications,” *Proceedings of the tenth international conference on World Wide Web Hong Kong*, May 2001, pp 525 – 534.

[18] H. Harney, “Group Key Management Protocol (GKMP) Specification,” IETF RFC 2093, July 1997, www.rfc-editor.org/rfc/rfc2093.txt.

[19] H. Harney, “Group Key Management Protocol (GKMP) Architecture,” IETF RFC 2094, July 1997, www.rfc-editor.org/rfc/rfc2094.txt.

[20] A. Ballardie, “Scalable Multicast Key Distribution,” RFC 1949, May 1996, www.rfc-editor.org/rfc/rfc1949.txt.

[21] C. Wong, M. Gouda, and S. Lam, “Secure Group Communications Using Key Graphs,” IEEE/ACM Transactions on Networking (TON), vol. 8, no. 1, Feb. 2000, pp. 16 – 30.

[22] A. Sherman and D. McGrew, “Key Establishment in Large Dynamic Groups Using One-Way Function Trees,” *IEEE Transactions on Software Engineering*, vol. 29, no. 5, May 2003, pp. 444 – 458.

[23] S. Rafaeli et al., *“An efficient one-way function tree implementation for group key management*,” 2001; http://www.comp.lancs.ac.uk/computing/users/rafaeli/papers/oft.zip.

[24] D. McGrew and A. Sherman, "Key Establishment for Large Dynamic Groups: One-way Function Trees and Amortized Initialization", Internet draft, work in progress, Feb. 1999.

[25] S. Rafaeli, L. Mathy, and D. Hutchison, “EHBT: An efficient protocol for group key management,” *Proc. third international workshop on Networked Group Communication London* (NGC'01), vol. 2233 of LNCS, Nov. 2001, pp. 159 – 171.

[26] A. Perrig, D. Song, and J. Tygar, “ELK, a New Protocol for Efficient Large-Group Key Distribution,” *Proc. IEEE Symposium on Security and Privacy*, May 2001, p. 247.

[27] I. Chang, R. Engel, D. Kandlur, D. Pendarakis, and D. Saha, "Key Management for Secure Internet Multicast Using Boolean Function Minimization Techniques," *Proc. of IEEE Infocom'99*, Mar. 1999, vol 2, pp. 689 – 698.

[28] S. Zhu, S. Setia, and S. Jajodia, "Performance optimizations for group key management schemes," *Proc. 23rd International Conf. Distributed Computing Systems*, 2003, pp. 163 – 171.

[29] O. Rodeh, K. Birman, and D. Dolev, “Optimized group rekey for group communication systems” *Symposium on Network and Distributed System Security (NDSS ’00) San Diego, Calif.*, Feb. 2000, pp. 37 – 48.

[30] S. Rafaeli, *A Decentralised Architecture for Group Key Management*, PhD appraisal, Computing Dept., Lancaster Univ., Lancaster UK, Jan. 2000.

[31] Y. Yang, S. Li, B. Zhang, and S. Lam, “Reliable group rekeying: design and performance analysis,” *Proc. ACM SIGCOMM ‘01*, San Diego, CA, 2001, pp. 27 – 38.

[32] X. Zhang, S. Lam, and Y. Lee, “Group rekeying with limited unicast recovery,” Tech. Rep. TR-02-36, Dept. Computer Science, Univ. Texas, Austin, July 2002. (Revised, Feb. 2003).

[33] S. Setia, S. Zhu, and S. Jajodia, “A Scalable and Reliable Key Distribution Protocol for Multicast Group Rekeying,” Technical Report, George Mason Univ., Jan. 2002.

[34] C. Wong and S. Lam, “Keystone: A Group Key Management Service,” *Proc. International Conf. on Telecommunications*, May 2000.

[35] M. Waldvogel, G. Caronni, D. Sun, N. Weiler, and B. Plattner, “The VersaKey framework: Versatile group key management,” IEEE J. Selected Areas Communications (Special Issue on Middleware), vol. 17, no. 9, Aug. 1999, pp. 1614 – 1631.