A Scalable Framework for Secure Multicast

Sreedhar Mukkamalla Randy H. Katz
{msreedha,randy}@cs.berkeley.edu
CS Division, EECS Department, U.C.Berkeley

Abstract

The lack of security mechanisms for IP multicast has impeded the large scale commercial deployment of applications such as pay-per-view information dissemination services and real-time videoconferencing. In this report, we describe the design and implementation of a scalable mechanism for secure multicast. It relies on a trusted, centralized security manager to authenticate participants and distribute keys. To ensure that participants are unable to access session data sent before they join or after they leave the group, the security manager rekeys the participants of the session in response to group membership changes using a scheme based on [21]. To avoid an excess of rekeying traffic that would be caused by frequent membership changes, we introduce an *epoch*-based rekeying protocol, wherein rekeying takes place at most once per a fixed duration of time called an epoch. We use the SRM [18] reliable multicast protocol to disseminate rekeying messages. To minimize disruption of the session at participants caused by loss of rekey messages (and their consequent inability to decrypt session data), the usage of new keys is delayed by an additional epoch. This technique maximizes the probability of the reliable multicast mechanism delivering the rekey message to all participants. The key distribution protocol and the corresponding objects have been implemented in the MASH [4] toolkit as reusable modules. Performance studies reveal the bottleneck in our system to be the bandwidth consumed by rekeying traffic. Based on our observations, we propose an extension to our scheme that would go a long way towards achieving true global scalability for secure multicast groups.

1 Introduction

The widescale deployment of IP Multicast [9] has enabled the efficient distribution of data for applications such as large scale information dissemination services, real-time videoconferencing, distributed interactive simulation, multiplayer gaming, software updates etc. However, the serious deployment of these services suffers from the problem that today's Mbone has no provisions for authentication, access control, and privacy.

In this report, we present the implementation of a multicast key distribution protocol that scales to large groups. The protocol has a centralized trust model and relies on a *Security Manager* that is responsible for group key distribution. Rekeying of the group uses a scheme that is based on [21] wherein the size of the rekeying message when a single participant joins or leaves the group is $O(\log N)$, where N is the number of participants. However, to avoid an excess of rekeying traffic, rekeying takes place in our protocol at most once per a fixed interval of time, called an *epoch*, rather than on every join or leave. (The actual duration of an epoch is an application-specific parameter ; it is typically of the order of several seconds). The rekey messages are disseminated reliably to the participants. For certain applications, group membership changes are disseminated securely and reliably along with rekeying information. To minimize disruption of the session at participants caused by loss of rekey messages (and their consequent inability to decrypt session data), usage of new keys is delayed by an epoch. While this technique causes a startup delay for newly joining members and allows leaving participants to continue accessing session data for a short period, it has the effect of maximizing the probability of the reliable multicast mechanism delivering the keys to all participants.

The key distribution protocol and the corresponding objects have been implemented in the MASH [4] toolkit as reusable modules in the form of 2 objects - a *Security Manager* object and a *Secure Network* object. The source code for the entire implementation is a little over 2500 lines of C++ code. The SSL [11] protocol is used for secure unicast communication between participants of the session and the Security Manager. The SRM [18] protocol is used for dissemination of rekeying information and group membership information in a scalable manner.

Measurements performed on the system to identify scalability bottlenecks reveal this to be the bandwidth consumed by rekeying traffic for large, dynamic groups. Based on these measurements, we propose an extension to our scheme that adds a single level of hierarchy of trusted security managers to the system. This scheme captures the basic idea of Iolus [17], but doesn't suffer from its latency problem. The key idea is that using just a single level of hierarchy increases scalability but doesn't add as much latency as an Iolus hierarchy of arbitrary depth would.

The rest of the report is organized as follows: Section 2 outlines the basic issues in multicast security and briefly describes related work. Section 3 outlines the design of our key distribution protocol. In Section 4, we discuss the implementation of the protocol in the MASH toolkit and resolve some practical implementation issues. Section 5 presents analytical and empirical results of the system's performance. In Section 6 we discuss the implications of our measurements on the scalability of the system, and propose an extension to our scheme. Finally, we summarize our conclusions in Section 7.

2 Multicast Security Issues and Related Work

In this section, we first outline the basic issues in multicast sessions with large, dynamic groups. Then, we motivate the need for secure multicast with some real-world examples and illustrate the basic scalability problem. We outline a taxonomy of multicast applications with different session dynamics and security requirements. Finally, we describe related approaches to the secure multicast problem.

2.1 Introduction

The advent of IP multicast has engendered a number of new applications that make use of multipoint communications over a wide-area. However, the large-scale deployment of multicast applications for mainstream use has been slow due to number of factors. Foremost among these are the lack of standard, well-understood mechanisms to address the problems of:

- Reliable multicast for large groups
- Techniques to deal with heterogeneity of receiver populations
- Scalable security for securing multicast communications

Security concerns in multicast transmission are far more challenging than those of traditional unicast. First of all, multicast communications are much more susceptible to interception and insertion attacks than unicast communications. This is because the IP multicast group membership model allows any multicast capable host to send data to, and receive data from, any multicast group. (Of course, limited control of distribution topology is possible through the use of the Time-To-Live (TTL) field, but this is much too coarse grained and does not solve the problem at all when the legitimate communicating entities are across the globe from each other.) Secondly, the consequences of a successful attack are likely to affect a larger number of individuals. In addition, existing methods of advertising multicast sessions [15] (which multicast session information and group addresses to everyone) make the attacker's job easier. The scalability issues that crop up in securing multicast communications are discussed later in this section.

2.2 Privacy Issues with Dynamic Groups

The basic goal of any network security protocol is to allow a set of authorized entities to communicate securely over an insecure network where a malicious attacker can presumably read, modify or delete data transmitted over the network. This is usually achieved by setting up a security association between the communicating entities through the exchange of keying material. The keying material can subsequently be used for authentication, confidentiality and integrity through cryptographic techniques. While this mechanism is well understood in the context of securing unicast communications, it does not extend in a straightforward manner to the realm of group communications over a multicast channel. This is because of the dynamic nature of the group membership (members can join and leave the group at any time). The security protocol must ensure that a member of the group can only participate in the session when she is authorized to do so. In other words, the security protocol has to ensure that a joining member cannot access previously multicast data and a leaving member cannot continue to access data that is multicast after she leaves the group. This implies that the group security association (i.e., the group key) has to be changed when the group membership changes.

2.3 Motivating Examples

While the above requirement does not hold for all conceivable multicast applications, one can motivate its necessity through several real-world scenarios. Consider a pay-per-view information dissemination service that uses multicast to distribute data and charges its customers for the amount of time they spend accessing the data. Obviously, the service provider does not want its customers to "cheat" by accessing data for any more time than they paid for. [17] quotes another example of a video-conference between a group of prosecuting attorneys who are discussing the strategy for a criminal trial. At various times, they wish to interview certain other people (e.g., police officers, witnesses etc.). These other people need to participate in the secure multicast session, but only while they are being interviewed - they should not have access to any other previous or future communication.

2.4 The Scalability Issue

The following discussion explains the scalability issue in ensuring privacy of multicast sessions. Assume the scenario of a secure multicast session where a central key distribution center (KDC) is responsible for authenticating participants and for generating and distributing new keys to the group. A single group session key is used by all participants to encrypt data sent to the group. Consider the case of a newly joining member. As noted previously, the group key will have to be changed if the new member is to be unable to decrypt data previously sent to the group. A simple way to do this would be for the KDC to generate a new key, encrypt the new group key with the old group key and multicast the encrypted version of the new group key to the existing group. The new member gets his key from the KDC through a secure unicast channel. The case of a member leaving the group is a much harder problem. Rekeying of the group cannot use the old group key to encrypt the new group key since the leaving member is in possession of the old key. The trivial solution would be for the KDC to distribute the new key to each of the remaining members individually through secure unicast. Obviously, this is extremely inefficient and presents a severe scalability problem when large groups are involved.

2.5 A Taxonomy of Multicast Security Requirements

The first step toward a workable solution to the multicast security problem is to realize that there is a large diversity of security requirements in the realm of multicast applications. The authors of [6] recognize this fact and observe that there is a taxonomy of requirements based on applications characteristics such as group size and membership dynamics. They suggest two benchmark scenarios that are exemplified by the following applications and their characteristics:

- Pay-per-view Information Dissemination:
  - Single sender
  - Very large number of passive recipients
  - Extremely frequent changes in group membership
- Secure Videoconferencing:
  - Interactive groups (multiple senders)
  - Relatively small groups (up to a few tens or hundreds at most)

The above scenarios accurately characterize almost all multicast applications in terms of group dynamics. However, from the security viewpoint, there is one more crucial difference between the two classes of applications - in the former class, members are oblivious of the existence of each other while in the latter, they are not. This means that in the former case, it is alright to admit new members into the group without informing other members whereas in the latter case, it is imperative that changes in group membership are reliably disseminated to participants of the session. Consider the following example : Alice and Bob are carrying on a secure videoconference. Carol now joins the conference by authenticating herself to the key distribution center. To prevent Carol from being able to decrypt Alice's and Bob's earlier conversation (which Carol might have recorded), the KDC generates a new key which is distributed to Alice, Bob, and Carol. However, if Alice and Bob are not made aware that Carol is now in the session, they might continue exchanging information that Carol was not meant to be privy to, thus defeating the purpose of changing the group key. Hence, for such applications, participants must also maintain an accurate notion of group membership.

2.6 Related Work

A number of cryptographic schemes exist in the literature that deal with the problem of broadcasting a secret to an arbitrary subset of a universe of users. For example, there are schemes that use polynomial interpolation [13], secret-sharing [3], extensions to the Diffie-Hellman key exchange algorithm [5], or secure locks based on the Chinese Remainder Theorem [8]. However, these are of theoretical interest only since they all require computation that, at a minimum, is proportional to the group size. In addition, the computation per member is relatively expensive, rendering these schemes no better than the trivial solution described previously.

[17] proposes dividing the logical multicast group into a hierarchy of IP multicast sub-groups. A Group Security Controller (GSC) manages the top-level of the hierarchy and Group Security Intermediaries (GSIs) manage the keys for each subgroup, with each subgroup having its own key. Each GSI knows the keys of its own subgroup and that of the level above it so it can translate messages to/from higher levels. One problem with this hierarchical approach is the additional latency incurred due to the translation; this is undesirable for applications such as videoconferencing which have tight end-to-end delay requirements. Another problem is that there is no workable solution to the problem of how this hierarchy is formed and administered in the first place. It also doesn't handle the rekeying operation within a subgroup in a scalable fashion.

[7] proposes another scheme for efficient rekeying of large groups when a member leaves. This scheme uses only $O(\log N)$ keys in all and the rekeying operation is extremely efficient. But unfortunately,

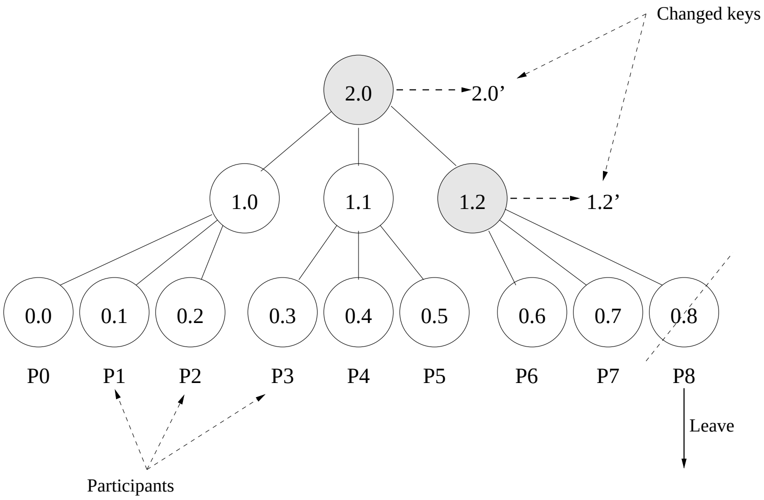

Figure 1: A hierarchy of keys maintained by the Security Manager. When participant 8 leaves the group, key 2.0 and key 1.2 have to be changed.

this scheme is susceptible to collusion attacks - members can decrypt data sent to the group (even after they were expelled) by exchanging their keys.

[21] and [20] use a key-tree hierarchy administered by a central server to make the rekeying of a large group tractable. Rekeying, when a single member leaves, is done by multicasting a single message of size $O(\log N)$ to the group. This idea forms the basis for our work, which to our knowledge, is the first implementation that studies the practical issues associated with large-scale multicast sessions. Foremost among these is the idea that rekeying the group should not disrupt the session for a majority of the participants. To this end, we came up with an epoch-based rekeying protocol which extends the basic rekeying scheme to allow for the joining or leaving of multiple members at once. Our scheme reaps two main benefits - firstly, it reduces the frequency of the rekeying operation (and hence the bandwidth consumed by rekeying traffic), and secondly, it minimizes the probability of disruption at receivers by delaying usage of new keys, thus providing a window of time for reliable delivery of rekey messages.

We identified the true scalability bottleneck of a system such as ours - the bandwidth consumed by rekeying traffic. Based on our observations, we propose an extension to our scheme that uses a single-level hierarchy of agents to directly tackle the scalability problem.

3 The Key Distribution Protcol

3.1 Overall Architecture

Our framework relies on a centralized, trusted Security Manager that is responsible for authenticating participants, and generating and distributing keys. In order to perform the rekeying operation in a scalable fashion, we adopted and extended the algorithm in [20] and [21] that uses a hierarchy of keys to make rekeying of the group an efficient operation.

First, we outline the basic scheme. Figure 1 shows a tree [1] of symmetric encryption keys maintained by the Security Manager. Each participant is represented as a leaf in the tree and is given a set

[1]This tree has nothing to do with the multicast routing tree - it is simply a data structure maintained by the Security Manager to make rekeying efficient.

of keys from the root to its position in the tree. For example, participant P2 is given three keys, key (2.0), key (1.0), and key (0.2). Since everyone has the root key, the group can communicate using it as the shared key for encryption of session data. When a participant leaves the group, all of her keys need to be changed. For example, when participant P8 leaves the group, key (2.0) needs to be changed to key (2.0′) and key (1.2) needs to be changed to key (1.2′). A hierarchy of keys makes re-establishing the new keys easier. Key (2.0′) can be encrypted by key (1.0) and multicast to participant P0, P1, and P2. Key (2.0′) can be encrypted by key (1.1) and multicast to participant P3, P4, and P5. Key (2.0′) can be encrypted by key (1.2′) and multicast to participant P6 and P7. Finally, key (1.2′) can be sent to participant P6 and P7 individually. To avoid sending a large number of multicast messages, all the above encrypted keys are put into one single rekey message which is multicast to the group. It is easy to see that with this scheme, when one participant joins or leaves, the size of the rekey message (assuming all the rekeying information is put into one message) grows as $O(\log N)$, where N is the number of participants in the group [2]. Note that rekeying messages need to be delivered reliably to participants if they are to be able to continue participating in the session.

In our current implementation, the tree is implemented as an actual tree of nodes using pointers (rather than as an array), with each node having a static number of children. The tree starts out empty (i.e., with a height of 0) and the height of the tree is increased on demand as the size of the group increases. Participants who leave the session leave "holes" in the leaves which are filled in by newly joining participants. The tree is not shrunk when participants leave. Thus, it is unlikely that the tree is full and balanced at any time since participants can join and leave randomly. However, there are no ordering constraints with regard to insertion of participants into the tree making it easier to maintain 'balanced-ness' [3].

3.2 Protocol Description

The protocol involves two types of entities, a *Security Manager* and *participants*. The Security Manager is responsible for authenticating participants, and generating and distributing keys. It periodically multicasts a heartbeat message to everyone in the group. The heartbeat contains the version number of the key currently being used to encrypt the session data. Every time the session key is changed, this version number is updated.

The Security Manager multicasts rekey messages whenever the session key is changed. For the purposes of rekeying, time at the Security Manager is divided into epochs. All changes to the group membership (i.e., *join*s or *leave*s) within a single epoch are aggregated and a single new rekey message is generated at the beginning of the next epoch and multicasted to the group. This epoch serves as a window of time within which any loss-recovery mechanism can take place to recover the lost rekey message at the participants. At the start of the following epoch, the heartbeat is updated to reflect the new version of the session key. The key idea here is that as long as participants get the rekey message sometime during the window of that one epoch, they will be in-sync and will not have to discard packets.

When a participant first joins the session, it sends a unicast *join* request to the Security Manager. The Security Manager authenticates the participant, checks an ACL, and if the participant is allowed to join the session, it is sent its set of keys (corresponding to the path from the participant's leaf to the root in the hierarchy of keys) at the start of the next epoch when the rekeying operation is done. The epoch following this one is when the new session key actually starts being used (as explained in the above paragraph). Thus, a new participant will experience a delay of up to two epochs when it joins the session.

[2]For more details on this scheme please refer to [21].

[3]A pathological case for this would be when a large number of participants join the group initially and most of them leave, resulting in a key-tree of sub-optimal height with only very few participants sparsely distributed among the leaves. One option in this situation would be to regenerate the tree from scratch

When a participant leaves the group, it sends a unicast *leave* message to the Security Manager. After receiving a *leave* request, the Security Manager will generate a rekey message at the beginning of the next epoch. In the epoch following this one, the new key will be used. Thus it takes up to two epochs to remove a participant from a group.

Figure 2 illustrates how the protocol works. At the beginning of each epoch, the Security Manager sends a heartbeat containing the current session key version. In epoch 1, two participants join and one participant leaves the session. Then at the beginning of epoch 2, the Security Manager aggregates all the *join* and *leave* requests and generates the appropriate rekey message. The rekey message is then sent reliably to the group. Then at the beginning of epoch 3, the Security Manager updates the heartbeat's session key version to signify that the new key should start being used.

Figure 3 illustrates the keys that need to be changed in the beginning of epoch 2, when two participants join and one leaves. Let participant P2 and participant P5 be the newly joining members and participant P3 be the leaving member. Three keys in the hierarchy need to be changed, key (2.0), key (1.0), and key (1.1). The rekey message will contain the following:

$$[2.0']'_{1.0}[2.0']_{1.1'}[2.0']_{1.2}$$

$$[1.0']_{1.0}[1.1']_{0.4}[1.1']_{0.5}$$

(where the notation $[2.0']_{1.0'}$ means that key $(2.0')$ is encrypted by key $(1.0')$).

4 Implementation

4.1 Platform and Programming Model

The key distribution protocol has been implemented in the MASH [4] toolkit. The MASH platform is a scripting-based programming environment for networked multimedia applications. It provides composable basic building blocks, such as network objects, codecs, widgets, and an event-driven programming model. A key part of the software architecture is the *Split Object Model* - the implementation of an object is split into low overhead control functionality implemented in a scripting language (OTcl/Tk) while performance critical data handling is implemented in a compiled language (C++). Compiled objects provide core, composable mechanisms that are "glued" together through the scripting language to flexibly implement applications.

The key distribution protocol in this report consists of 2 objects - a *Security Manager* object and a *Secure Network* object (which implements the participant). Figure 4 and Figure 5 show how these objects fit into the overall architecture. The objects are implemented in C++ and compiled into the MASH shell. They export OTcl and C++ interfaces which are listed in the Appendix.

4.2 Joining a Session - the Initial Bootstrap

As explained in the previous section, participants join the session by authenticating themselves to the Security Manager through a secure unicast channel. In our implementation, the SSL protocol [11] is used for this purpose. SSL is a widely used security protocol that provides communications privacy over the Internet. It uses public-key certificates for authentication and allows client/server applications to communicate in a way that is designed to prevent eavesdropping, tampering, or message forgery. After an SSL connection is established between the Participant and the Security Manager, the Security Manager checks an ACL to decide whether or not to admit the Participant. If the Participant is admitted to the session, it is given its set of keys at the start of the next epoch when rekeying is done. Subsequently, the SSL connection is closed.

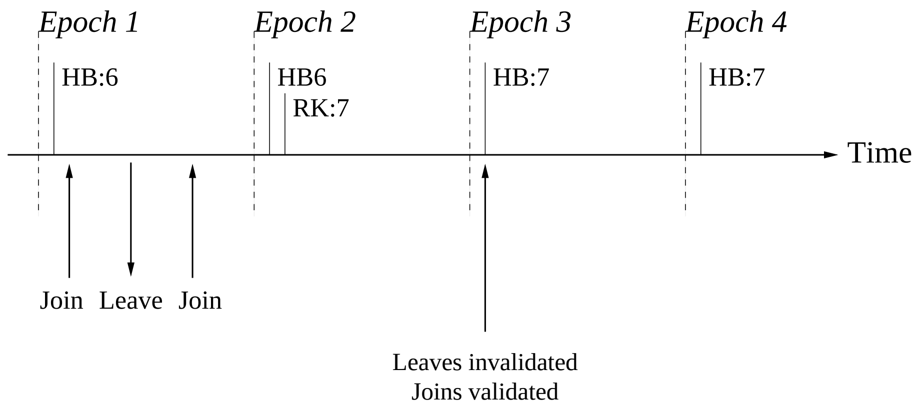

Figure 2: Two people joining and one person leaving in Epoch 1. HB:6 is a heartbeat message indicating that the group key version is 6. RK:7 is a rekey message containing version 7 of the group key.

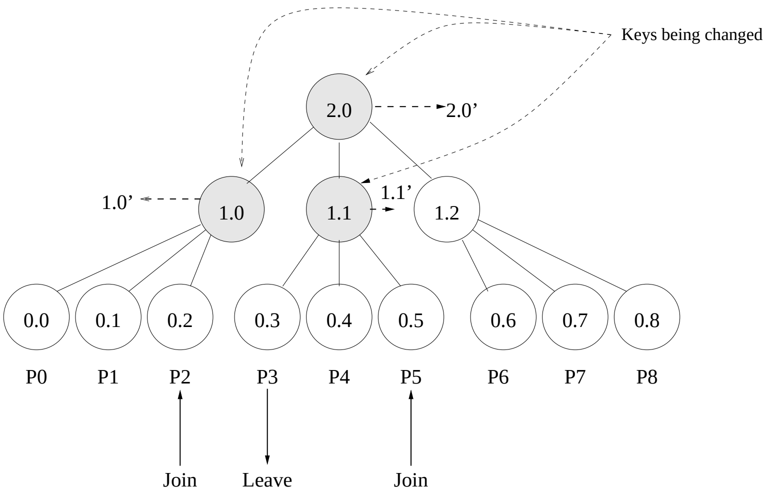

Figure 3: Keys that need to be changed after two participants join and one participant leaves the group.

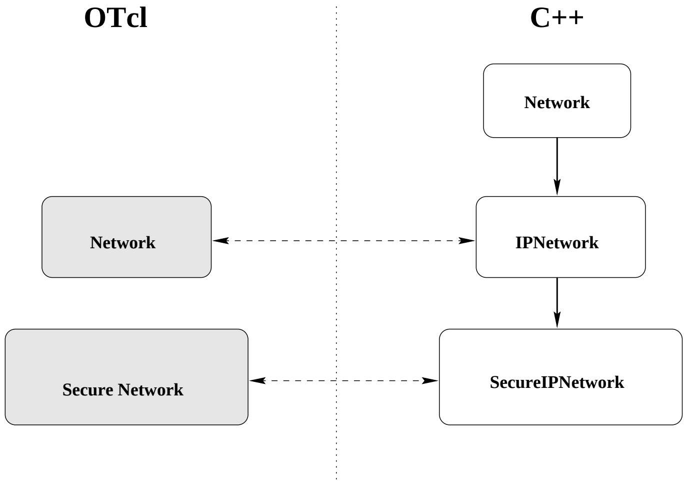

Figure 4: Class hierarchy of the Network Objects in the MASH toolkit

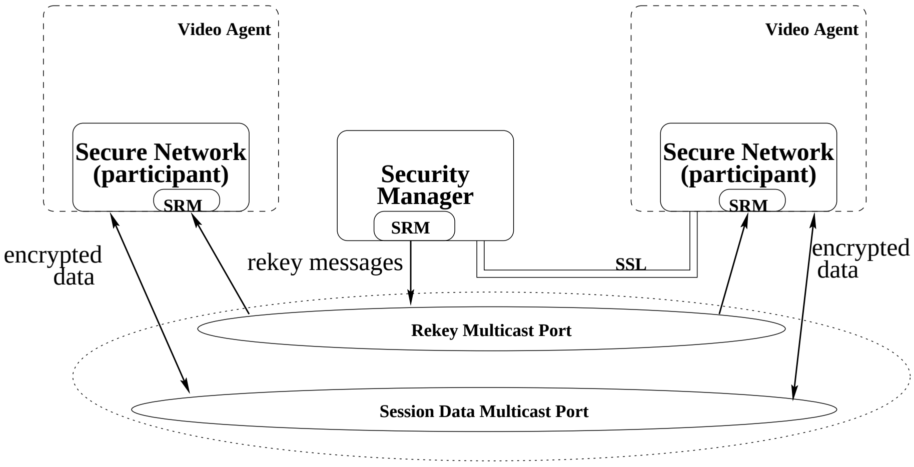

Figure 5: The objects involved in the key distribution protocol. The video agents are for illustration purposes only. They could be replaced by any other objects that need to use secure multicast.

4.3 Dissemination of Rekeying Information

As the group membership changes, the Security Manager periodically changes the group key (and the appropriate auxillary keys), as described in the previous section. The message containing the rekeying information has to be disseminated *reliably* to all remaining participants since a participant cannot decrypt data from the session if its notion of the group key is not up to date. For this purpose, we use the SRM [18] reliable multicast protocol. Rekey messages are digitally signed and timestamped by the Security Manager to prevent forgery and replay attacks. Rekeying information is sent out on a separate port on the same multicast group as the session. (This is very similar to the RTP/RTCP [19] approach of sending data and control packets to different ports of the same multicast group). This is illustrated in Figure 5.

For certain applications such as secure videoconferencing, it is meaningless to change the group key as members join and leave the session unless existing members are made aware of the changes in group membership in a reliable fashion. Hence, group membership information has to be disseminated securely and reliably as well. This is done by having the Security Manager include information about changes in the group membership along with the rekey messages. All newly joining participants receive a list of current participants from the Security Manager over the SSL connection when they initially join. They subsequently maintain an up-to-date list of members by applying the deltas to the group membership which they receive in rekey messages. Note, however, that group membership information dissemination in this fashion is not very scalable to large, dynamic groups simply because of the volume of information that needs to be distributed. Fortunately, the requirement of an accurate notion of current group membership only holds for those applications that tend to have small or moderately sized groups, such as secure videoconferences. For applications with extremely large group memberships, such as pay-per-view multicast services, receivers do not need to know (and perhaps, *should not* know) the identities of the other receivers of the service.

4.4 Avoiding Startup Latency

Using our key distribution protocol (described in Section 3), a newly joining participant will have to wait for a period of between 1 and 2 whole epochs before she can participate in the session. The reason behind this approach was to avoid disruption of the session caused by loss of rekeying traffic at current members. However, this approach has the undesirable effect of introducing a substantial startup latency for newly joining members of the session. For certain applications, (specifically, those applications where individual participants do not care about other participants), there is no harm in admitting new participants to the group immediately.

For these applications, our framework takes the following approach. For the duration between the time when a newly joining participant authenticates itself to the Security Manager and the time when the new key that it gets starts being used, it receives data for the group on a temporary multicast group. This is facilitated by the Security Manager serving as a *reflector* between the actual session multicast group and this temporary multicast group - the Security Manager receives a data packet on the actual session multicast group, decrypts it, re-encrypts it with a temporary key and multicasts it to the temporary multicast group. The temporary key is given to all participants joining in the current epoch and is changed every epoch.

4.5 Delay in Expulsion of Leaving Participants

As with the case of newly joining participants, in our key distribution protocol, there is a delay of between 1 and 2 epochs after a member leaves before the session key is actually changed. This is an unavoidable consequence of trying to delay usage of new keys until almost all participants have received the new keys. The implications of the fact that a leaving participant can continue to decrypt session data for a period of 1 to 2 epochs is dependent on the application and the epoch

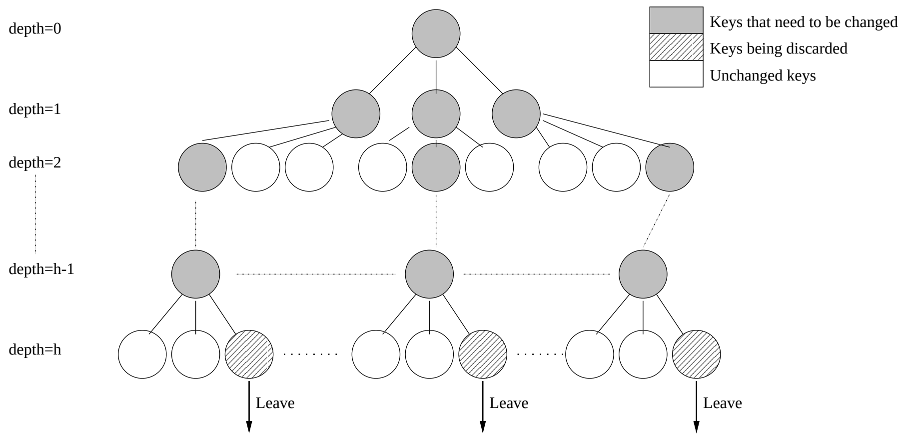

Figure 6: The keys that need to be changed when multiple participants leave the group in a single epoch.

duration. For videoconferencing applications, group membership information is disseminated along with rekeying information. Thus, a sender always knows *exactly* what set of receivers can decrypt the data she sends to the group and for this case, the delayed usage of new keys when a member leaves may not pose a problem. For pay-per-view information dissemination applications, a participant can continue to use the service even after she has "left" the group for a period of upto 2 epochs. Since epochs will be of the duration of a few seconds at most, we believe this is not a very serious problem.

5 Performance

5.1 Worst Case Rekey Message Size

While the basic scheme for rekeying of the participants of a session outlined in Section 3 results in a single rekey message of size $O(\log N)$, the rekey message is obviously larger if a single rekey message is used to handle multiple participants joining or leaving the group as is done in our scheme. First, we analyze the size of the rekey message in the worst case, as a function of the number of participants who join or leave. For more details regarding the rekey scheme when a single participant joins or leaves, refer [21].

For simplicity, we restrict our analysis to the case of multiple members *leaving* a session in the same epoch. (The size of the rekeying message is always greater for a leave than for a join, so we are deriving an upper bound for the size of the rekey message by only considering leaving members). For this discussion, assume that the tree is fully balanced.

Let the number of current participants be N. Let the key-tree degree be d. Assume the tree is fully balanced (i.e., N is an exact power of d). The height of the tree, h, is then equal to $\log_d(N)$. Let the number of particpants leaving in the current epoch be k. To simplify the discussion, assume k is an exact power of the key-tree degree d, i.e. $k = d^a$ for some integer a.

The worst-case scenario for the rekey message size occurs when the leaving participants are evenly distributed among the leaves of the key-tree. In this case, all keys of the tree starting from the root down to a depth a will have to be changed. In addition, from a depth $a + 1$ onwards down to a depth $h - 1$, each of the k leaving participants contributes a chain of keys to be changed. Since, when a member leaves, each of the keys that needs to be changed is encrypted with each of its d children (except for the changed keys at depth $h - 1$, each of which have to be encrypted with only

$d-1$ keys since one of their children corresponds to a leaving member), we get the size of the rekey message (in units of *rekey entries* where each rekey entry is a key encrypted by another key along with key ID information [4]), S to be:

$$S = d^0 \cdot d + d^1 \cdot d + \ldots + d^{a-1} \cdot d + ((h-2) - a + 1) \cdot d^a \cdot d + d^a \cdot (d-1)$$
$$= \frac{d}{d-1} \cdot (k-1) + d \cdot k \cdot (\log_d \frac{N}{k} - 1) + k \cdot (d-1)$$

Figure 6 illustrates the situation when 3 members are leaving a group having key-tree degree 3. If the height of the tree is 4 (i.e., there are 81 participants in the group), then up to a depth of 1 all keys have to be changed. From depth 2 to depth h-1, there are 3 chains of keys that have to be changed (one per participant).

For the case when the number of leaving members is not an exact power of the key-tree degree d, we can derive the size of the rekey message to be[5]:

$$S = \frac{d}{d-1} \cdot (d \cdot d^{\lfloor \log_d k \rfloor} - 1) + d \cdot k \cdot (\log_d N - \lfloor \log_d k \rfloor - 2) + k \cdot (d-1)$$

This serves as a reference against which to compare the actual sizes of rekey messages when a group of leaving participants is not distributed evenly among the leaves, but picked at random.

5.2 Rekeying Measurements

We have conducted a number of experiments to evaluate the performance of the rekeying scheme at the Security Manager. The experiments were carried out on a lightly loaded Pentium 200 Mhz PC running FreeBSD 2.2.7. The Security Manager was modified to simulate the joining of a large number of participants followed by the leaving of a number of them. The experiments were conducted for key trees with various degrees with a group size of 2000 participants and varying numbers of participants leaving the group.

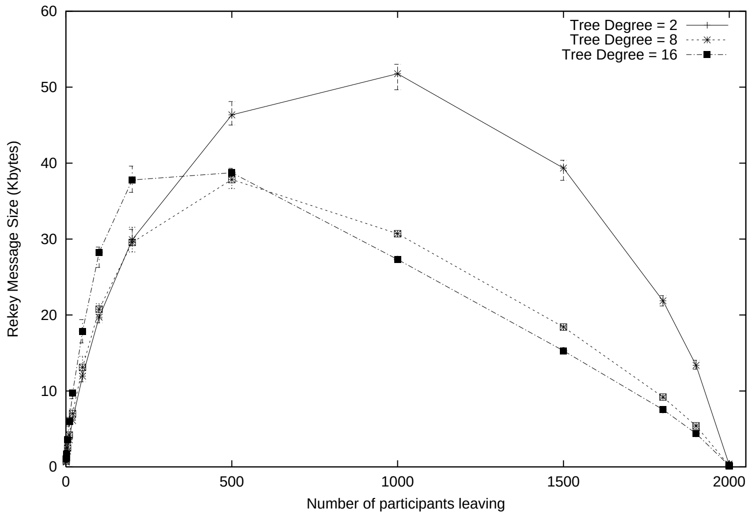

Figure 7: Size of rekey messages when a number of participants leave the group in an epoch (The initial group size is 2000).

[4]In our implementation, which uses 56-bit DES keys, each such rekey entry is 24 bytes long

[5]This formula is valid only when k is less than or equal to N/d

Figure 7 shows the size of the rekey message for a group of size 2000 when varying numbers of participants leave the group. Figure 8 shows the same data with the range on the x-axis constrained to a maximum of 10% of the group leaving. For each value of the number of leaving participants, a random subset of participants from the group was chosen and removed from a group initially populated with 2000 members. For each point in the graphs, this measurement was averaged over 20 iterations.

Figure 9 shows the rekey message size for a key tree of degree 2 with an initial group size of 2000 (with randomly chosen leaving participants) along with the worst-case rekey message size, which occurs when all leaving participants are uniformly distributed among the leaves of the key tree. Figure 10 shows the same data with the x-axis constrained to a maximum of 10% of the group leaving.

From the graphs, we can make the following conclusions. The increase in the size of the rekey message as a function of the number of leaving participants is almost linear when the fraction of the entire group that the leaving participants constitute is less than about 5% of the group size (for less than 100 participants leaving from a group of 2000). The reason for this is intuitively obvious - for relatively small groups of randomly chosen leaving users from a large session, there is little overlap in the key tree with regard to keys that have to be changed. (A very small amount of overlap exists at the top levels, but otherwise each leaving participant contributes a separate chain of keys from the root to its leaf that have to be changed). As the number of leaving participants grows to a larger fraction of the group size, the overlap of keys that need to be changed grows and the rekey message size grows much slower. When most of the members in the group are leaving, entire subtrees can be "pruned" away leading to smaller and smaller rekey messages.

Figure 11 shows the processing time at the Security Manger for the generation of each rekey message. This includes the time taken to traverse the key tree, generate new keys, update the key tree, create the rekey message and sign it digitally using RSA-512 signatures. The processing time is roughy linear in the number of leaving participants when the leaving number is relatively small.

While these graphs explain the complexity of the rekeying operation, what is important is its behaviour in practical circumstances - when a small fraction of the group is leaving and joining the session every epoch, as can be expected in a large-scale information dissemination service over the Mbone, for example. A back of the envelope calculation for a session of size 2000 participants, 5% of whose membership changes (randomly) per epoch (each epoch being 10 seconds, for example) leads to rekey messages of size 20 Kbytes, which translates to a stream of rekeying messages that consumes a bandwidth of 2 Kbytes/sec on average that has to be transmitted *reliably* to all participants. The processing time at the Security Manager for generating a rekey message that evicts 5% (100 participants out of 2000) of the group is less than 100 ms in all cases.

5.3 Authentication Measurements

While the size of the rekey messages might be a valid scalability concern for large, dynamic groups, another potential bottleneck in a framework such as ours, lies in the ability of the Security Manager to handle *join* requests from a large number of participants. This is because of the computational expense involved in generating and verifying digital signatures. In our implementation, we use the SSL protocol for mutual authentication of the Security Manager and Participant. Each SSL connection setup involves public-key operations at each of the Participant and the Security Manager, which are computationally quite expensive when signature schemes such as RSA are used. In our implemenation, we used X.509 certificates with RSA public keys (both 512-bit and 1024-bit modulus). To measure the ability of the Security Manager to handle *join* requests, we measured the time taken by it to handle a single *join* request. The measured time includes the processing time of the SSL connection setup, followed by the sending of a *join request* by the participant for which the Security Manager sends a *join response*. The participant and the Security Manager are both on the same local area network (for which end-to-end delays are less than a millisecond). Table 1

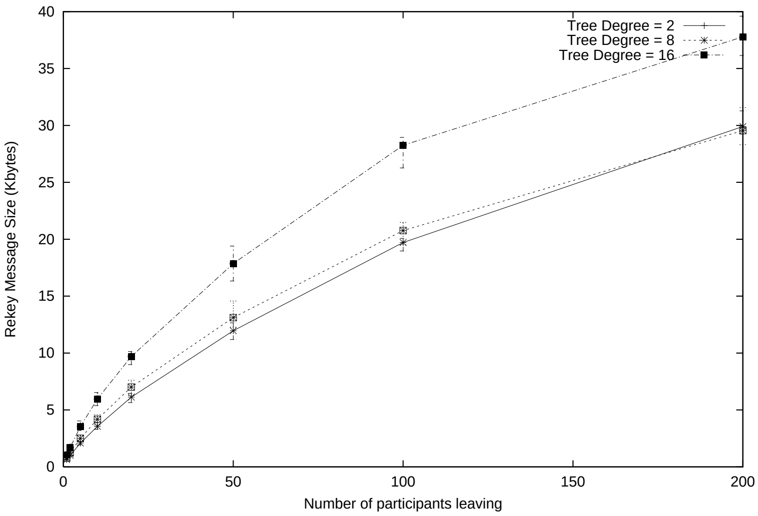

Figure 8: Size of rekey messages when a number of participants leave the group in an epoch (The initial group size is 2000).

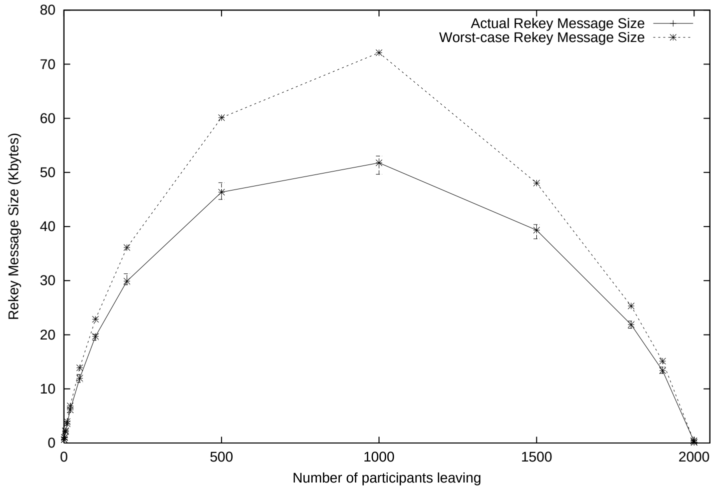

Figure 9: Average and worst-case rekey message size when a number of participants leave the group in an epoch (Initial group size = 2000 and Tree Degree = 2).

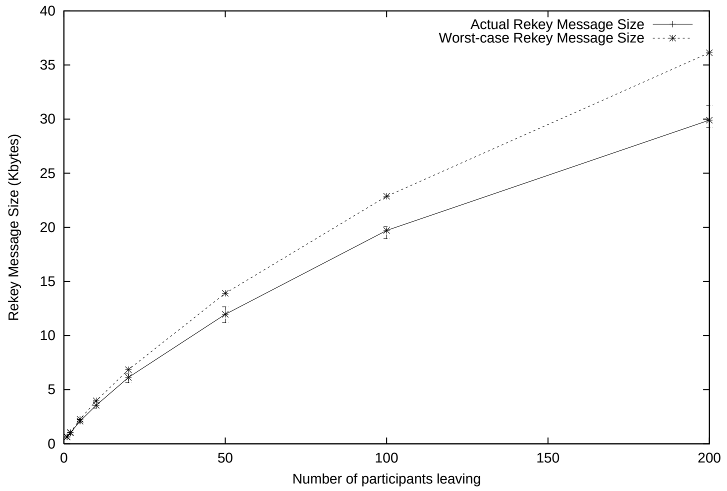

Figure 10: Average and worst-case rekey message size when a number of participants leave the group in an epoch (Initial group size = 2000 and Tree Degree = 2).

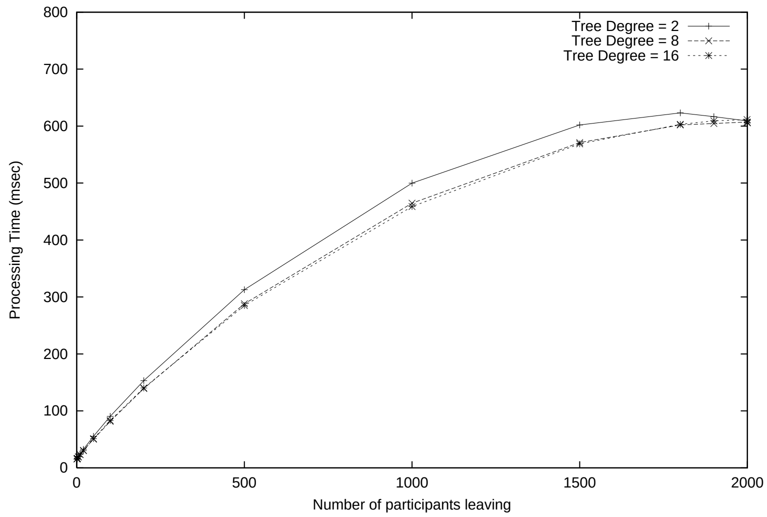

Figure 11: Processing time at the Security Manager for generating rekey messages.(The initial group size is 2000).

Signature Scheme	Processing time for a *join* request (msec)
RSA-512	13.891
RSA-1024	32.870

Table 1: Security Manager processing time per *join* request

shows the processing time for *join* requests at the Security Manager with RSA-512 and RSA-1024 certificates being used.

6 Discussion and Future Work

6.1 Scalability of the System

The measurements in the previous section lead to the following conclusions about the system:

- Processing time for the generation of the rekey message at the Security Manager is not likely to be the bottleneck for a large-scale multicast session that uses our framework. The time taken by the Security Manager to generate a rekey message for evicting 100 participants from a group of 2000 is less than 100 ms. (The processing involved for an equal number of participants joining the group would be even less.) In our system, with rekeying being done at most once per epoch (where an epoch would be of the order of a few seconds at least), this does not pose a serious performance problem.
- Processing at the Security Manager for admitting a newly joining participant into the group uses the strong authentication mechanisms that the SSL protocol provides. The processing times for processing join requests reported in the previous section lead us to conclude that a maximum of around 71 users can be authenticated per second when certificates using RSA-512 are used. When RSA-1024 is used, only about 30 users can be authenticated per second. For large sessions with extremely dynamic group membership, this could potentially be a limiting factor. However, this is not really a bottleneck due to two reasons. Firstly, the rapid acceleration in CPU performance nowadays will remove the processing bottleneck. Secondly, the authentication of participants can easily be parallelized to improve performance.
- The bandwidth consumed by the rekeying traffic for highly dynamic groups is a major performance concern. The calculations in the previous section illustrate that receivers need to continuously keep receiving streams of rekeying information of several Kbytes of data per second on average, *reliably*. (The actual bandwidth consumed by the rekeying traffic is, of course, a function of the actual group size, the fraction of change in group membership per epoch and the epoch duration - we arrived at a figure of 2 Kbytes/sec for a group of 2000, 5% of whose membership changed per epoch, with each epoch being 10 seconds long.)

The above observations lead to the conclusion that a secure multicast system built around a framework such as ours is likely to meet its bottlenecks in the bandwidth consumed by rekeying traffic. Multicast sessions for which scalability of this order of magnitude tends to be a concern are those that come under the first of the 'benchmark' scenarios outlined in Section 2. These are applications that are typically non-interactive (i.e., they have a single sender or very few senders, at the most) and have extremely large receiver populations with highly dynamic group membership behaviour. While the results of our performance studies lead us to the conclusion that achieving scalable security for sessions on the order of tens of thousands (or perhaps even millions) of receivers cannot be done using a framework such as ours, we believe that introducing a single level of hierarchy into the system can vastly improve scalability.

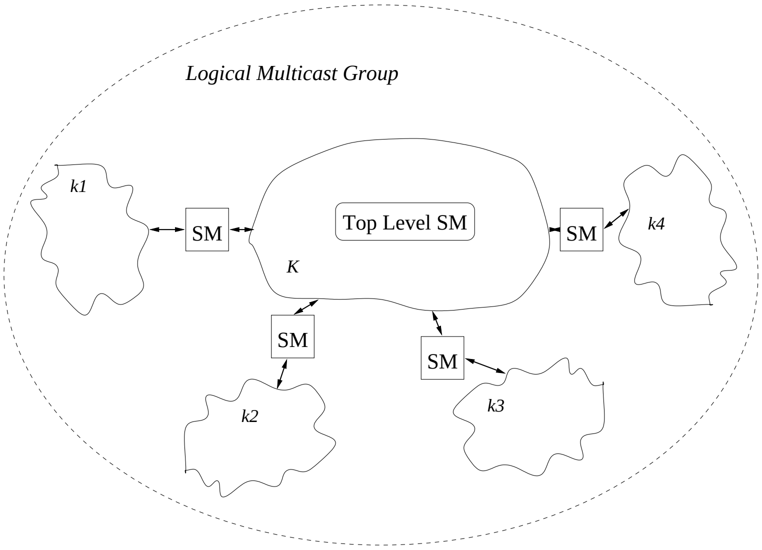

Figure 12: A secure multicast scheme using key-graphs with one level of hierarchy. Each of the inner clouds constitutes a separate secure IP multicast group.

This is illustrated in Figure 12. A number of Security Managers (all serving the same logical multicast group) form the 'participants' of a higher level secure multicast group that is administered by a Top Level Security Manager, using the key tree scheme of our framework. Each Security Manager is responsible for a local sub-group of participants, indicated by the cloud next to each Security Manager in the figure. Each such local sub-group is a separate IP multicast group and uses its own encryption key. Participants joining or leaving any such multicast group cause rekeying messages to be sent by the Security Manager only to the other participants in the same local multicast group. However, there is the notion of a single secure *logical multicast group* that spans all the local multicast groups - this is facilitated by the various Security Managers serving as 'forwarders' which forward data between their local subgroup and the higher level secure multicast group whose members are the other Security Managers. Each Security Manager decrypts any data that it receives on the higher level secure multicast group and re-encrypts it using the session key of its own local multicast group and multicasts it to the local group and vice versa. This approach is very similar in spirit to the idea in Iolus [17], with one major difference. In Iolus, the Group Security Intermediaries (the Iolus equivalent of our Security Managers) form a physical hierarchy and hence every data packet has to undergo repeated encryptions and decryptions to achieve the notion of a logical, secure, spanning multicast group, leading to unnecessary delays in delivery of the packet. In our proposed scheme, the Security Managers are all members of a single higher level multicast group and hence the decryption/re-encryption operation has to take place only once. The authors of [17] observe that the penalty for this operation can be mitigated by applying the old trick of a level of indirection - using a new randomly generated *message key* to encrypt each message and only encrypting the message key using the subgroup session key [6].

Another advantage of our proposed scheme over Iolus is that the physical hierarchy of Group Security Intermediaries in Iolus renders it less fault-tolerant - a subgroup depends on all its ancestor Group Security Intermediaries to be alive in order for it to be part of the logical group. In our scheme, for a subgroup to be part of the logical multicast group, only the local Security Manager and the Top Level Security Manager have to be up ; any other Security Manager that is down only causes the participants in its local subgroup to be cut off from the rest of the logical group.

[6] Performance measurements conducted by the Iolus authors reveal that each Group Security Intermediary contributes about 1 millisecond of additional latency for packets of size 1.5 Kbytes. (Larger packets incur even more latency).

The above scheme addresses both the scalability concerns of our implementation directly - firstly, each Security Manager is only responsible for managing the participants in its local subgroup, so the load of authenticating participants for the entire session is distributed among the Security Managers. Secondly, group membership changes within a subgroup only affect participants in that subgroup, greatly bringing down the amount of rekeying traffic that any given participant sees [7]. This, we believe, is crucial for sessions such as large-scale information dissemination services.

6.2 Fault Tolerance

Like in any other system that depends on a centralized server for proper functioning, our system has a single point of failure - the Security Manager. The progress of the secure multicast session depends crucially on the Security Manager being up and alive at all times. If the Security Manager crashes during a session, the existing participants can continue the session but new participants will not be admitted. The current participants will stop receiving heartbeat messages from the Security Manager and their keys will time out (the API of the current implementation exposes this to the application). At this point, it is up to the application built on top of our framework to try to join the session again.

If a participant joins the session and then crashes, ideally, it should be removed from the session (just as if it had sent a *leave* request to the Security Manager.) This can be achieved using soft state with timeouts at the Security Manager to keep track of participants who are 'alive'. This soft state can periodically be refreshed by 'I am alive' messages from the participants of the session with the frequency of these messages being inversely proportional to the number of participants in the session, thus keeping the overall bandwidth consumed by these messages constant.(This is the approach that Mbone sessions using the RTP/RTCP [19] protocols adopt.) However, in our system, the Security Manager has an exact notion of how many people are in the group at any given time. This can be made use of to let participants know exactly how many participants there are in the session (the group size can be multicast in the heartbeat messages sent by the Security Manager). Given this, it may not be necessary for participants to multicast the feedback messages - they can simply unicast them to the Security Manager. (Note that these feedback messages will have to be securely sent encrypted and authenticated using the shared key between the Security Manager and the participant, which the latter receives when she joins the group.) Our current implementation, however, does not incorporate this mechanism - participants who 'die' remain in the session.

6.3 Using Reliable Multicast

As mentioned previously, we currently use the SRM 2.0 reliable multicast toolkit [18]. The only reliability guarantee that a NAK-based transport protocol like SRM can give is that of eventual consistency - the sender can never be sure that *every* receiver has received all its data. In fact, the heterogeneity of the network connectivity of the receiver population may result in some receivers losing a lot of data and never 'catching up' with the session's progress. This is a distinct possibility in our system when large sessions with highly dynamic groups are in progress - the rekeying traffic alone (not to mention the actual session data) could swamp poorly connected receivers and cause exactly this situation. In our system, if a heartbeat message (also sent using reliable multicast) from the Security Manager is not received within a certain interval, the participant declares itself 'out-of-sync' and notifies the application. It is then upto the application to choose whether or not to attempt to rejoin the session. (Consider the case when a participant is isolated from the Security Manager for a period of time due to a network partition. When the partition heals, the participant has to recover *all* previous rekey messages in order for it to get its keys up to date and participate in the session. This might not be desirable if the total amount of lost rekeying traffic it has to

[7] In order that rekeying traffic of each sub-group be localized and not congest the backbone links, it is important that the division of the logical multicast group into different domains (handled by different Security Managers) map closely onto the actual network topology.

recover is very large - it might just be more efficient to contact the Security Manager directly and get its current set of keys through unicast. In our current implementation, this is done by explicitly rejoining the session. A future optimization to avoid the SSL connection setup again could be to use the shared key between the participant and the Security Manager (which the participant receives when she initially joins the group) for this secure unicast communication.

6.4 Security of the System

For all unicast communication between participants and the Security Manager, our implementation uses the SSL protocol with either RSA-512 or RSA-1024 certificates. The symmetric keys used for the key tree (and hence the rekey messages) and for encryption of session data are 56-bit DES keys. (In the future, this could be extended to other ciphers such as triple-DES, RC4 or Blowfish [8].) Apart from unicast communication between the Security Manager and the participants, the only other messages are the heartbeat messages and rekey messages sent by the Security Manager. These are digitally signed and encrypted using the current session key. They also include timestamps to guard against replay attacks.

7 Conclusions

In this report, we have presented a scalable framework for building secure multicast applications. We rely on a trusted centralized Security Manager for authentication of participants and distribution of keys. To ensure that participants are unable to access session data sent before they join and after they leave the group, the Security Manager rekeys the participants of the session in response to changes in the group membership. To avoid an excess of rekeying traffic and possible disruption of the session at receivers, rekeying takes place at most once per a fixed interval of time, rather than on every membership change. Rekey messages (and, for some applications, group membership information) are disseminated using the SRM reliable multicast protocol. The framework has been implemented as a pair of reusable modules in the MASH toolkit.

Based on measurements conducted on our implementation, we concluded that the bottleneck of the system was likely to be the bandwidth consumed by rekeying traffic. We proposed the addition of one level of hierarchy to our system to tackle the scalability problem. Our proposed scheme captures the essential idea of the Iolus [17] scheme while doing away with some of its drawbacks.

As the awareness and popularity of IP multicast increases in the Internet, we expect to witness a growth in its use for commercial applications and services. Scalable security for these services will be crucial if they are to be economically viable. We believe that our framework is an exploratory step in this direction.

References

[1] E. Amir, S. McCanne, and R. Katz. An active service framework and its application to real-time multimedia transcoding. In *Proceedings of ACM SIGCOMM'98*, September 1998.

[2] T. Ballardie. *Scalable Multicast Key Distribution, RFC 1949*, May 1996.

[3] S. Berkovits. How to broadcast a secret. In *Advances in Cryptology: Proceedings of CRYPTO '91.* Lecture Notes in Computer Science 576, Springer-Verlag, Berlin, 1991.

[8] Usage of other symmetric ciphers doesn't change the system's functionality in any way. Note, however, that the size of the rekey messages will be greater if ciphers with larger keys are used.

[4] E. Brewer, S. McCanne, R. Katz, L. Rowe, E. Amir, Y. Chawathe, K. Mayer-Patel, A. Coopersmith, S. Raman, A. Schuett, D. Simpson, A. Swan, T.-L. Tung, D. Wu, and B. Smith. Toward a common infrastructure for multimedia networking middleware. In *7th International Workshop on Network and Operating Sytems Support for Digital Audio and Video*, May 1997.

[5] M. Burmester and Y. Desmedt. A secure and efficient conference key distribution system. In *Advances in Cryptology: Proceedings of CRYPTO '94*. Lecture Notes in Computer Science 839, Springer-Verlag, Berlin, 1994.

[6] R. Canetti and B. Pinkas. *A taxonomy of multicast security issues*, May 1998. draft-canetti-secure-multicast-taxonomy-00.txt.

[7] I. Chang, R. Engel, D. Kandlur, D. Pendarakis, and D. Saha. Key management for secure internet multicast using boolean function minimization techniques. In *Proceedings of INFOCOM '99*, August 1989.

[8] G. H. Chiou and W. T. Chen. Secure broadcasting using the secure lock. In *IEEE Transactions on Software Engineering*, August 1989.

[9] S. E. Deering. *Multicast Routing in a Datagram Internetwork*. PhD thesis, Stanford University, December 1991.

[10] S. Floyd, V. Jacobson, C.-G. Liu, S. McCanne, and L. Zhang. A reliable multicast framework for light-weight sessions and application level framing. In *Proceedings of ACM SIGCOMM '95*, August 1995.

[11] A. O. Freier, P. Karlton, and P. C. Kocher. *The SSL Protocol Version 3.0*, November 1996. draft-freier-ssl-version3-02.txt, Internet Draft.

[12] R. Gennaro and P. Rohatgi. How to sign digital streams. In *Advances in Cryptology - CRYPTO '97*. Springer-Verlag LNCS 1294, pp. 180-197, 1997.

[13] L. Gong and N. Shacham. Multicast security and its extension to a mobile environment. *ACM-Baltzer Journal of Wireless Networks*, 1995.

[14] V. Jacobson and S. McCanne. Visual audio tool. Software Online, ftp://ftp.ee.lbl.gov/conferencing/vat.

[15] M. P. Maher and C. Perkins. *Session Announcement Protocol: Version 2*, May 1998. draft-ietf-mmusic-sap-v2-00.txt, Internet Draft.

[16] S. McCanne and V. Jacobson. vic: A flexible framework for packet video. In *ACM Multimedia*, November 1995.

[17] S. Mittra. Iolus: A framework for scalable secure multicasting. In *Proceedings of ACM SIGCOMM '97*, 1997.

[18] S. Raman and Y. Chawathe. libsrm: A framework for reliable multicast transport. Software Online, http://www-mash.CS.Berkeley.EDU/mash/software/srm2.0/.

[19] H. Schulzrinne, S. Casner, R. Frederick, and V. Jacobson. *RTP: A Transport Protocol for Real-Time Applications, RFC 1889*, January 1996.

[20] D. M. Wallner, E. J. Harder, and R. C. Agee. *Key management for multicast: Issues and Architectures*, July 1997. draft-wallner-key-arch-00.txt, Internet Draft.

[21] C. K. Wong, M. Gouda, and S. S. Lam. Secure group communications using key graphs. In *Proceedings of ACM SIGCOMM '98*, September 1998.

[22] C. K. Wong and S. Lam. Digital signatures for flows and multicasts. In *Proceedings of IEEE ICNP '98*, October 1998.

[23] T. Wong, T. Henderson, S. Raman, A. Costello, and R. Katz. Policy-based tunable reliable multicast for periodic information dissemination. In *Workshop on Satellite-based Information Services, WOSBIS'98*, 1998.

[24] E. A. Young. Ssleay 0.8.0. Software Online, ftp://ftp.psy.uq.oz.au/pub/Crypto/SSL/.

A Object APIs

A.1 Secure Network API

The Secure Network object exports the following OTcl member functions:

- `secureJoin $SMaddr $SMport $multicastgroup $multicastport $ttl $keymulticastgroup $keymulticastport $certfile $privkey $CAfile $CApath`
  Used to join a session by contacting the Security Manager at the addr/port specified by `$addr/port`. `$multicastgroup/$multicastport` and `$keymulticastgroup/$keymulticastport` are the address/port pairs used for session data and rekeying information respectively. The arguments `$certfile, $privkeyfile, $CAfile and $CApath` specify the files used for certificates and private key of the participant.
- `leave`
  Used to leave the current session.
- `setHandler $handlerobject`
  Specifies the name of an OTcl object which is invoked on certain events (can be used to implement a UI object, for instance). This object must handle the methods:
  - `add-participant $name`
  - `delete-participant $name`
  - `security-manager-name $name`
  - `message $msg`

In C++, it exports the following functions:

- `send()`
  Used to send data to the group.
- `recv()`
  Used to received data from the group.

A.2 Security Manager API

The Security Manager exports an OTcl API with the following functions:

- `startSession $localport $multicastgroup $multicastport $keymulticastgroup $keymulticastport $epochduration $certfile $privkeyfile $CAfile $CApath $ACLfile $ttl $handlerobject $tempmulticastgroup $groupinfo`
  Starts the Security Manager listening on the local unicast port `$localport`. `$multicastgroup/ $multicastport` and `$keymulticastgroup/$keymulticastport` are the address/port pairs used for session data and rekeying information respectively. The argument `$epochduration` specifies the duration of an epoch. The arguments `$certfile, $privkeyfile, $CAfile and $CApath` specify the files used for certificates and private key of the Security Manager.

`$ACLfile` specifies a file that stores a list of authorized participants. `$ttl` specifies the TTL value for all multicast packets. `$handlerobject` specifies the name of an OTcl object which is invoked whenever there is a change in group membership (can be used to implement a UI object, for instance). This object must handle the 2 methods:

  - `add-participant $name`
  - `delete-participant $name`

  `$tempmulticastgroup` specifies the temporary multicast group that is used by the Security Manager to distribute data to newly joining participants. (If this is not desired, this argument is set to the value 0). `$groupinfo` is a flag whose value should be set to 1 if group membership info is to be disseminated.

- `endSession`
  Ends the current session.

- `evictMember $name`
  Evicts the member named `$name` from the group in the next epoch.