The Cost of Adding Security Services to Group Communication Systems[*]

Cristina Nita-Rotaru
Advisor: Dr. Yair Amir

Submitted in partial fulfillment of the Ph.D. qualifiers
Department of Computer Science
Johns Hopkins University

Abstract

Numerous applications requiring information delivery from one sender to many receivers are based on a group communication model. Group communication systems are used in industry and military systems where reliability and high-availability are required. With the growth of the Internet, the number of applications that can take advantage of a group communication infrastructure increased (teleconferences, white-boards, video conferences, distributed interactive simulation, collaborative work). Over wide area networks the need for providing confidentiality, integrity, and authenticity of messages is essential.

In this paper we present Secure Spread, a secure version of the Spread Toolkit. Secure Spread is a group communication system that utilizes contributory group key management developed by the Cliques project and Blowfish symmetric encryption algorithm. Its modular design allows drop-in replacement of encryption and/or key agreement protocol. This work will not go to the details of a complete solution that handles every possible combination of network events. Rather it will focus on the performance evaluation in the general case. The results will give a good indication and insight of the overall cost of security in a group communication environment.

1 Introduction

Numerous applications requiring information delivery from one sender to many receivers are based on a group communication model. Group communication systems are used in industry and military systems where reliability and high-availability are required. With the growth of the Internet, the number of applications that can take advantage of group communication infrastructure increased (teleconferences, white-boards, video conferences, distributed interactive simulation, collaborative work). Over wide area networks the need of providing confidentiality, integrity, and authenticity of the messages is essential.

Providing security services for group communication is a challenging task. The interrelation between high-availability and security guarantees are not well understood, especially in the presence of general, possibly cascading, network events. Since the group is a dynamic entity, key generation is an asynchronous event, as oppose to session establishment at connect time in a two-party case. Multicast communication implies using a complete network in contrast to unicast where we have only one channel. It is more difficult to maintain the security in such an environment.

[*]This work was supported in part by a grant from the National Security Agency under the LUCITE program.

Key agreement is a critical part of providing security services for group communication. There exist several different ways to achieve key agreement in a group. One solution is to have a single entity that decides upon a key and then distributes it to the group. In this case the key generation entity maintains long-term keys with each member of the group in order to enable secure two-party communication used to distribute the key. A flavor of this solution uses a fixed, trusted third party. as the key generation entity. Since using a trusted third party does not support network partitions, this is not an acceptable solution for secure group communication. Instead, empowering a member of the group, chosen in a deterministic way, with the ability of generating keys seems more attractive. Another solution looks into providing a contributory key where each member of the group adds its own share such that the group key is a function of the individual contributions.

Each of the above solutions has its advantages and disadvantages. In a centralized key agreement environment, the trust of the whole system is put in the member that generates the key. Moreover, the key generation responsibility makes this member an attractive target for an attacker. A contributory key generation, in contrast, achieves a better randomness of the key, even if some of the participants lack a good random generator. In general, the computational effort required for a contributory key agreement protocol is much larger. However, in certain cases, when there is no one fixed trusted entity, a centralized key agreement protocol may be more expensive if the last key generator is leaving the group. A good key agreement protocol needs to provide strong security guarantees (key independence, authentication, key confirmation, perfect forward secrecy, resistance to known-key attacks) while being scalable.

We present a secure group communication system that utilizes a contributory key agreement protocol. We discuss the costs of adding security services to the Spread [1, 23] group communication toolkit. We focus on the encryption, key agreement, and key distribution costs. This work will not discuss the details of a complete solution that handles every possible combination of network events. Rather, it will focus on the performance evaluation in the general, more common case. The results will give a good indication and insight for the overall cost of security in a group communication environment.

The rest of the paper is organized as follows. We present some related work in Section 1.1. Section 2 describes the architecture of the system we developed. We discuss some of the issues raised when integrating key agreement with reliable group communication in Section 3. Section 4 presents performance evaluation. Finally, we conclude our work and discuss future work in Section 5.

1.1 Related Work

Designing and implementing a secure group communication is a complex problem and there are many works that can relate to it. The most obvious one is secure group communication itself. Another important field is key management that includes key distribution (or key transport) and key agreement.

Key distribution protocols rely on a centralized entity they focus on distributing the key and they are not very interested in addressing dynamics in group membership change. The main concern for such protocols is scalability. They perform better for very large groups (usually the case with one sender and many receivers) and are usually based on a tree hierarchy.

The problem of scalability was addressed by S. Mittra who proposed Iolus [15]. This work is one of the first that addresses the scalability problem by making use of a hierarchy of subgroups. Scalability is achieved by having each subgroup be relatively independent. Because there is no global shared key, leaves and joins affect only the subgroup.

An interesting hierarchical approach to improve scalability was taken by C.Wong at al [14]. Basically instead of using a hierarchy of group security agents as Iolus did they used a hierarchy of keys. They assumed a trusted server responsible for group access control and key management. They implemented the re-keying strategies and protocols in a group key server.

An actual implementation of group communication systems focusing on security issues is the secure distributed CORBA (Immune) system built on top of Secure-Ring [17] group communication work at UCSB [18]

The system developed at UCSB provides protection against Byzantine failures using cryptographic techniques to secure a low-level ring protocol (that forms the base of the Totem system) and replicating the protected CORBA objects sufficiently to detect and recover from up to a fixed number of compromised objects or computers.

Another implementation of a secure group communication is the Ensemble [11] system developed by the group at the Cornell University. The group has a long history in developing group communication systems: RAMPART, Isis, Horus, Ensemble, and recently Spinglass. RAMPART system [16] was concentrated on Byzantine robustness, while Ensemble addresses the problem of generating a shared group key and re-keying. It allows application-dependent trust models and optimizes certain aspects of group key generation and distribution protocols. A group member in Ensemble can be authenticated using a common group key or a member's long-term secret (e.g., using Kerberos Authentication Server [19] or PGP key).

Ensemble's security mechanism is based on the extensions of conventional cryptographic tools as PGP authentication engine [12]. Ensemble uses a centralized key generation approach, where a group leader, chosen as the lowest-numbered member of the group, is responsible of generating secure keys. Key distribution is performed using encryption (via PGP). This solution is expensive since the key must be encrypted individually for each member and does not provide perfect forward secrecy since a compromise of just one member's long-term secret (PGP private key) exposes all previous group session keys, and, hence, all prior group communication. Recent research had a result a decentralized and optimized approach for group re-keying. [13].

2 Secure Spread Architecture

The final goal of this work is to have a scalable secure group communication system, while maintaining the high-availability and group services such systems traditionally provide. We achieve this by integrating and adapting the Cliques protocol suite [3, 4, 5] with the Spread toolkit. [1]. This results in a secure group communication layer and an API, adding security to the traditional services.

Modularization is one of the most important issues that we consider when designing Secure Spread. This allows, in particular, fast replacement of the key agreement and encryption building blocks. The Cliques design was very appealing to us. It defines a protocol which allows members of the group to compute a key in the presence of join and leave events. The key is contributory, so that each member is adding a private share to the computation of the key. The algorithm is relatively efficient and guarantees key independence, key confirmation, perfect forward secrecy and resistance to known key attacks. Cliques is based on group extension of the Diffie-Hellman key exchange algorithm [2].

The Cliques API is a library that can be used to implement a key agreement protocol. Using the Cliques API, we implemented two such protocols: the first which we will refer to as distributed Cliques, is a contributory key agreement. The second, centralized Cliques, uses the last member to join the group as a key generation entity.

We achieve confidentiality by encrypting the messages using the Blowfish [7] encryption package. This package implements an efficient symmetric encryption algorithm that encrypts 64 bytes of plaintext, using a variable key (32 to 448 bits). It combines a Feistel network, key-dependent S-Boxes, and a non-invertible F function to create what is perhaps one of the most secure algorithms available. There are no known attacks against Blowfish. Blowfish is free and is also available as part of the Openssl library [9].

Next we present an overview of Spread, then we describe Cliques, and finally we detail our integrated system.

2.1 Spread

Spread is a group communication system for wide and local area networks. Spread provides all the services of traditional group communication systems, including unreliable and reliable delivery, FIFO, causal, and total ordering, and membership services with strong semantics.

Spread creates an overlay network that can impose any arbitrary network configuration including for example, point-to-multi-point, trees, rings, trees-with-subgroups and any combinations of them, to adapt the system to different networking environments. The Spread architecture allows multiple protocols to be used on links between sites and within a site.

Spread is very useful for applications that need the traditional group communication services such as causal and total ordering, and membership and delivery guarantees, but also need to run over wide area networks.

Spread is constructed as a long-running daemon and a library that links with the application. This architecture has many benefits, the most important for wide-area settings is the resultant ability to pay the minimum necessary price for different causes of group membership changes. Simple join and leave of processes translate into a single message. A daemon disconnection or connection does not pay the heavy cost involved in changing wide area routes. Only network partitions between different local area components of the network requires the heavy cost of full-fledged membership change. Luckily, there is a strong inverse relationship between the frequency of these events and their cost in a practical system. The process and daemon memberships correspond to the more common model of lightweight and heavyweight groups.

Spread scales well with the number of groups used by the application without imposing any overhead on network routers. Group naming and addressing is no longer a shared resource (the IP address for multicast) but rather a large space of strings which is unique per collaboration session.

Spread can support large number of different collaboration sessions, each of which spans the Internet but has only a small number of participants. The reason is that Spread utilizes unicast messages on the wide area network, routing them between Spread nodes on the overlay network.

The Spread system provides two different semantics: Extended Virtual Synchrony [20, 21] and View Synchrony [22] which will be discussed later. The EVS semantics is provided by the Spread library, while the VS semantics is provided by the Flush layer on top of it. The Spread toolkit is available publicly and is used by several organizations for both research and practical projects. The toolkit supports cross-platform applications and has been ported to several Unix platforms as well as Windows and Java environments.

2.2 Cliques

Cliques [3, 4, 5] is a cryptographic protocol suite which provides authenticated contributory group key management. Cliques also guarantees key independence, perfect forward secrecy and resistance to known key attacks. Cliques generates a contributory key, using a one-way function of random shares from every member of the group. This approach fits the nature of peer-group communication and also facilitates obtaining a better key, as the randomness of key generation does not rely on the pseudo-random generator of just one part. Rather it is enough that only one participant will have a good pseudo-random generator.

Cliques provides key independence. Any new key is independent of all of the previous keys, so that an attacker can not find out any information about previous or future session keys, even if somehow he obtained some session keys.

A key management algorithm with a strong resistance to active attacks should provide perfect forward secrecy and be resistant to known-key attacks. A system that does not achieve perfect forward secrecy allows an attacker that acquires the long-term key, to be able to find out past session keys, so he can decrypt previous traffic. If long-term secret keys are compromised, future session may be compromised too, as the attacker has the opportunity to impersonate. Perfect forward secrecy protects old communication even if long-term keys are compromised. Note that in the group communication environment, the attacker can be a member of the group. Perfect forward secrecy prevents members of the group to decrypt communication exchanged before they become members of the group. Perfect forward secrecy is one of the strongest services guaranteed by Cliques. Cliques is also resistant to known-key attacks. A known-key attack assumes that an attacker that knows one or more past malformed session keys can compute the long-term keys.

The Cliques API [6] is an implementation of the Cliques protocol suite. Its main purpose is to implement the cryptographic primitives, assuming the existence of an underlying communication system that provides the group communication. Cliques confers a special role for a group controller, the last member to join a group. This role floats as the group membership changes. A controller is charged with initiating key regeneration following membership changes.

The following operations will trigger key regeneration:

- join: a single new member is added to the group.
- merge: one or more members are added to the group. The merge operation optimizes for massive joins to the group.
- leave: one member voluntarily leaves the group.
- partition: one or more members leave the group due to a network event. Cliques handles leaves and partitions in a similar manner.

Cliques allows a key refresh operation which may be initiated only by the controller.

In addition, the Cliques API also implements the primitives for the centralized key agreement protocol. This protocol confers a special role to a group controller. The controller generates a new key every time the group view changes and distributes the key to the group. Failures or partitions of a group controller can be solved by selecting a new controller for the remaining group.

2.3 Integration

In this section we describe our integrated system. We start by motivating our decision to use the VS semantics and we discuss the interaction between Cliques and Spread. We then present the system architecture which allows drop-in replacement of key agreement and encryption algorithm. Finally, we detailed the state machine implementing distributed Cliques.

There are two semantics generally accepted in the group communication literature: Extended Virtual Synchrony (EVS) model [20, 21] and View Synchrony (VS) model [22]. These two models have much in common: they both guarantee that group members see the same set of messages between two sequential group membership events and that the order of messages requested by the application is preserved. However, there is a difference between them: EVS guarantees that messages are delivered to all recipients in the same membership as the message was originally sent on the network. VS, in contrast, guarantees the stricter property that messages are delivered to all recipients in the same membership as the sending application thought it was a member of at the time it sent the message. Providing this property requires a round of application acknowledgement messages before installing a new membership. This need for application level acknowledgements requires that the groups be closed, only allowing members of the group to send messages to it. This triggers some benefits of EVS over VS: it has better performance, it allows open groups, where non-members of a group can send messages to the group, and provides a more general and stronger service. VS semantics can be implemented on top of EVS semantics, but EVS can not be implemented on top of VS.

It is important to point out that the knowledge that a message is received in the membership the application believed it was sent in, which is provided by VS semantics makes implementing a secure group system much easier: every message is encrypted with the same key that the receiver believes is current when the message is delivered to it. To implement a group key agreement protocol on top of EVS would require the security layer to implement something very like VS semantics to correctly maintain which messages were sent using which key.

Thus, at this point we do not see any significant benefit to building security requiring only EVS semantics as a library layer. Therefore, we build the security library on top of the Flush layer that provides VS semantics.

Understanding the interaction between the group communication system and the key agreement library, and clearly delimiting what one provides for the other, is one of the important parts of this work. Spread handles all the low level network communication, it maintains a view of the group and updates this view as the group membership changes. Spread handles all the network events (failure, network partition, merges) as well as member voluntary events (join, leave). The Cliques API defines a collection a functions helpful in implementing the Cliques key agreement protocol, without handling any communication. The Cliques API maintains a current view of the group which is based on information provided to it by the group communication service. The core of the integration is implementing the Cliques protocol, the glue that ties the Cliques library and Spread, using the information about group changes provided by Spread and making the correct Cliques calls at the correct moment. This forms the Secure Spread library.

Two observations can be made. First, we need a mapping from Spread events to Cliques events. Second, the Secure Spread Layer has to implement the logic of the Cliques protocols [3, 4, 5]. This logic is not encapsulated in the Cliques library as presented by its API. The only events meaningful for Cliques are join, leave, merge, partition, and key refresh. A complete mapping of Spread events to Cliques events is presented in Table 2.3.

To illustrate this, suppose that a network partition splits a group of four members into two network components, each with two members. Secure Spread will translate this event into a Cliques leave event that includes both partitioned members. The Cliques protocol will be invoked by Secure Spread (in each of the components) and will finally translate into a Secure Spread membership notification to the application.

Spread Events	**Cliques Events**
Join	Join
Leave	Leave
Disconnect	Leave
Partition	Leave
Merge	Merge
Partition + Merge	Leave then Merge
Group Change Request	N/A
N/A	Key Refresh

Table 1. Mapping of Spread Events to Cliques Events

One of the main goals in the design of Secure Spread is having a flexible system that will allow an easy drop-in algorithm change. Most of the practical cryptosystems are not provably secure. The security comes most of the times from preventing a brute-force attack by choosing parameters that will require a huge computational effort to break, while still being efficient during regular operation. In this context, the trust in encryption algorithms may vary over time. Of course, better algorithms are continuously introduced. Therefore, a flexible and modular Secure Spread will minimize the time and effort required in order to take advantage of the latest advances in crypto algorithms, or replacing algorithms known to be broken. Such a modular architecture is presented in Figure 1.

The Secure Spread layer consists of three modules: Secure Spread module, the Cliques Protocols module and the Blowfish encryption module. Secure Spread relies on the Flush layer to provide the communication infrastructure and uses the Cliques library as a tool in implementing the Cliques protocols. Secure Spread library is the engine of this layer, implementing a state machine that handles all of the communication events. This layer also exports the secure group communication API that any application can use.

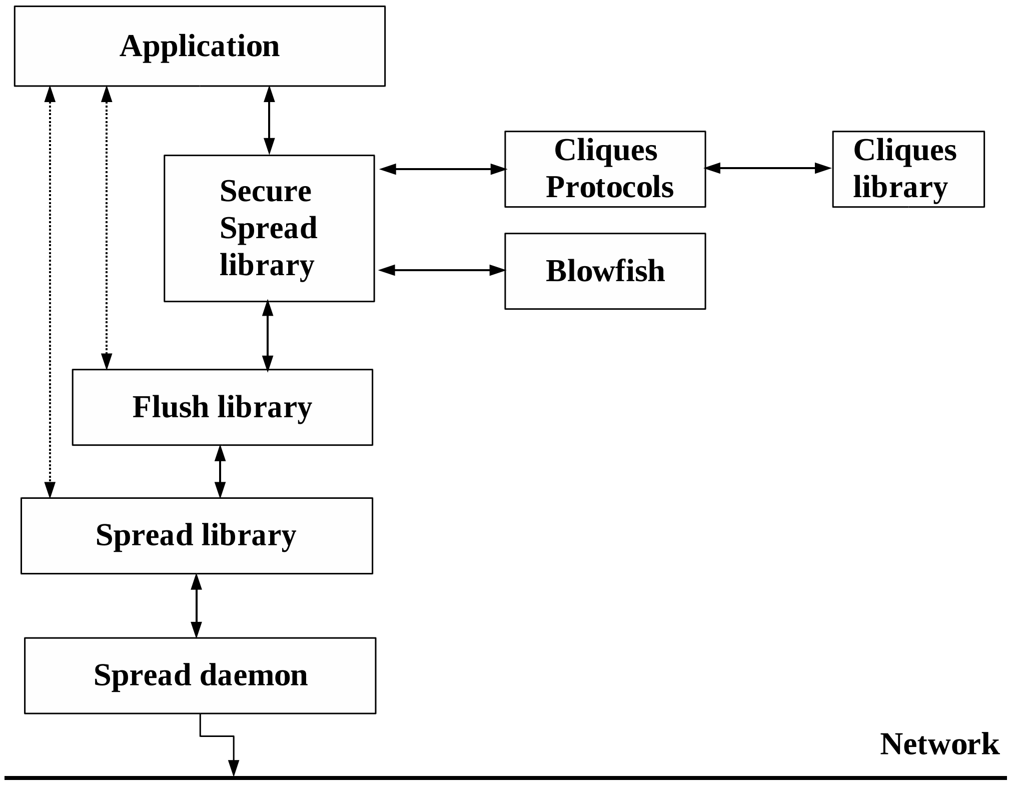

Figure 1. Secure Spread Architecture

The core functionality of Secure Spread is encapsulated within an event-handling loop. Network events are generated by the VS group communication layer (Flush). Secure Spread handles each of these events and, depending on their context, passes it on to the proper module that handles the event. If the event is receiving/multicasting of a regular message, the message is passed to the Blowfish module that will decrypt/encrypt it. If the event is a membership change, it is passed to the Cliques Protocols module that will perform the key agreement protocol. Note that during initialization, the encryption algorithms and the key agreement algorithms need to register with Secure Spread. Secure Spread can handle different key agreement protocols and different encryption algorithms for different groups. The Secure Spread module defines two general interfaces: one for encryption and one for key agreement. Any encryption algorithm or key agreement protocol can plug-in to the Secure Spread module by implementing these interfaces.

The Blowfish module includes the Blowfish algorithm (a modification of the Bruce Schneider's free implementation) [7, 8] and the Blowfish interface to the Spread module. Secure Spread Clients specify the encryption algorithm to be used upon connecting to Secure Spread. When clients require to send encrypted messages, the Secure Spread module invokes the general encryption function which in turn will invoke the Blowfish specific call. For now Spread supports only the Blowfish encryption algorithm.

The Cliques Protocols module implements the key agreement protocols, using the Cliques library, by invoking the Cliques API. In order to communicate with the Secure Spread module, the Cliques Protocol module needs to register with the Secure Spread module. It specifies the special messages that are needed by the key agreement algorithm. FIFO ordered messages are used to communicate partial keys or key list either using multicast to the group or using unicast between two particular entities, as specified by the Cliques protocols [3, 4, 5]. When receiving an event, Secure Spread checks whether the Cliques protocol is registered for it, and in that case it passes

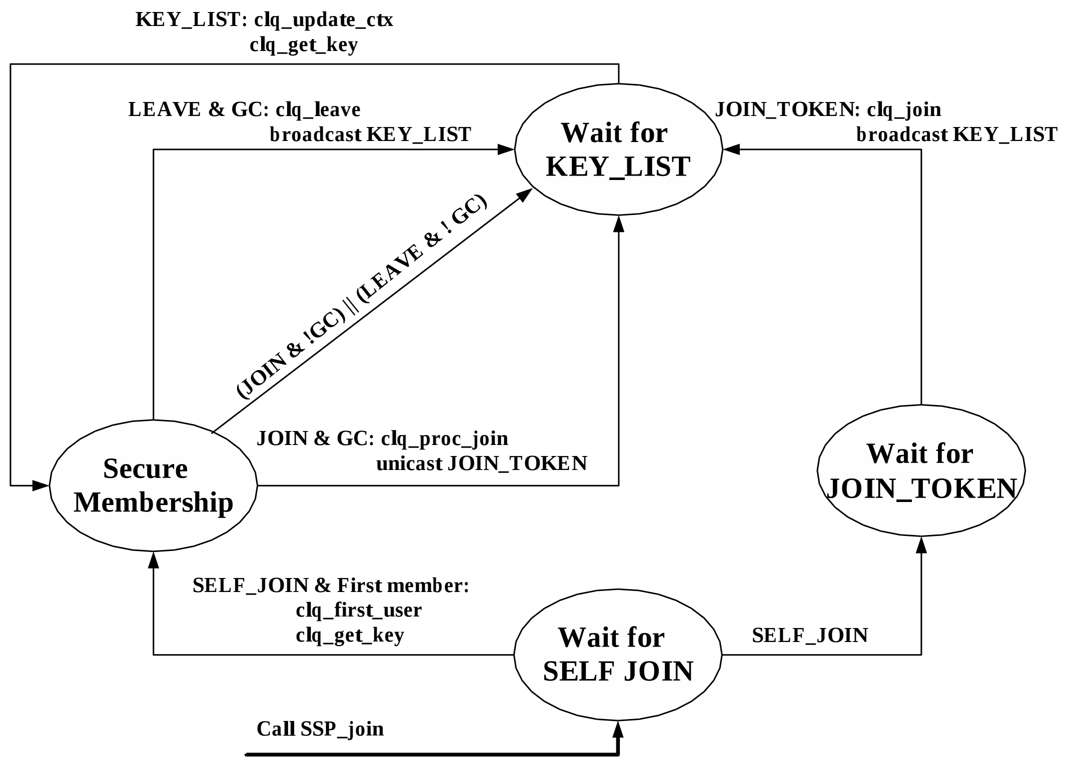

Figure 2. Distributed Cliques State Machine - join and leave

the event to the Cliques Protocol module.

Currently, the Cliques Protocol module implements two different algorithms for key agreement: Distributed Cliques key management and Centralized key management.

The Cliques protocols are defined by two state machines. A simplified state machine for the Distributed Cliques protocol is presented in Figure 2.

In each of the algorithms, each member of the group operates based on the relevant state machine, some particular information of the user(if he is or not the group controller, if he is or not the first member in a group), and the current event affecting the group. Two main actions are taken by a member: invoking Cliques API calls (this translates in computations), and sending (unicast or broadcast) the output of Cliques API calls via Spread to other members of the group, as required by the protocol.

A user starts executing the state machine upon invoking the *SSP_join* call of the Secure Spread API. The group communication infrastructure will detect that the group view has changed and will deliver a membership notification due to a join message (JOIN). To simplify the state machine, we name a Join event that is received by the user triggering the event, SELF_JOIN. All the other members of the group will see this event as a regular JOIN event. If the user is in Wait for SELF_JOIN state (see Figure 2, and receives the SELF_JOIN event, two situations are possible: that user joined a non-existent group (in this case the group is created and the user becomes the first member of the group), or the group already exists. In the first case, since no communication is necessary, the member can move directly to Secure Membership state, after computing a key. If the group already exists, the user moves to Wait for JOIN_TOKEN state, as part of reaching an agreed-upon key.

The Secure Membership state in Figure 2 is the stable state in which the secure group is functional. In this state, all of the members own the group key and can communicate securely.

When a new member joins the group, all the other members of the group will receive the JOIN event. All the members, but the group controller will move to the Wait for KEY_LIST state, waiting for a KEY_LIST message that will contain the list of partial keys. The group controller refreshes his share in the previous key and unicasts the new list from the previous group context to the new member (which at this point is in the Wait for JOIN_TOKEN state). When the new member receives that unicast from the group controller, it adds its contribution to the list and then broadcasts it to the group. All the members will receive this KEY_LIST message, will extract the key for the new membership, and install the new membership, moving to the Secure Membership state.

When a member leaves the group, the state machines behaves as follows. All the members, but the group controller will move to the Wait for KEY_LIST state. The group controller refreshes the list of partial keys and then broadcasts it to the group. Upon receiving the KEY_LIST message all the members obtain the new key and move to the Secure Membership state.

The state machine that we presented does not handle every possible combination of network events and assumes that no additional event happens while performing the key agreement algorithm. We discuss some of the issues that a complete solution must handle in Section 3.

3 Discussion

Two approaches are available when designing a secure group communication. The first is implementing the security services and key agreement protocol at the daemon level. The second approach is to implement them at the client level. Each of these approaches has its advantages and drawbacks. A design that integrates the security services at the daemon level will definitely trigger a major improvement in the latency, throughput and computation required when the membership of the group changes. Enforcing security policy is easier in such a model, where access to daemons and specified groups is easier to control. However, this approach is more complex as the security protocols are tailored into the reliability, ordering and membership protocols.

By integrating the security services at the client API level, a higher protection is achieved because the daemon's network is not trusted. It is much more dangerous to compromise a daemon than a single group. On the other hand, model has a higher cost in terms of performance. It should require, in addition, ensuring a secure communication link between clients and daemons, and at least some minimal access control at the daemon level.

A secure group communication system has to address, in addition, problems such as member certification and trust. It is more difficult to solve the trust problem in a group communication environment since trust becomes dynamic. One solution is to authenticate members at connection time, which has to be done at the daemon level. A scalable method of certifying the clients is needed. Also, a group authentication can be taken into consideration. This will confer a member the ability to sign messages either for itself, or as part of a group.

One desirable feature of a secure group communication system is to be robust and fault-tolerant. An invocation of the key management algorithm is not an atomic event (unfortunately). A change in the group membership triggers an agreement on a new key, which is an operation that requires computations done by more than one party and one or more broadcasts and unicasts, depending on the nature of the membership event that triggered the key change. As the system performs the key agreement algorithm, new, possibly cascading events may happen. Handling such situations is necessary for a correct system. A simple approach will restart the algorithm when receiving a cascaded event, drop all the work done so far, and restart the algorithm from scratch whenever a cascading event happened. However, a more efficient approach tries to save the work done as much as possible optimizing the total work to generate the new key. Implementing a fully optimized solution requires deeper modification in the Cliques library, in the sense that in the current implementation, while performing big chunks of computations, Cliques takes the control and it is not aware of network event occurrences. The problem is more delicate then that and is the subject of future work.

Another problem that needs to be handled is preserving all the guarantees that Spread provided (causal, agreed

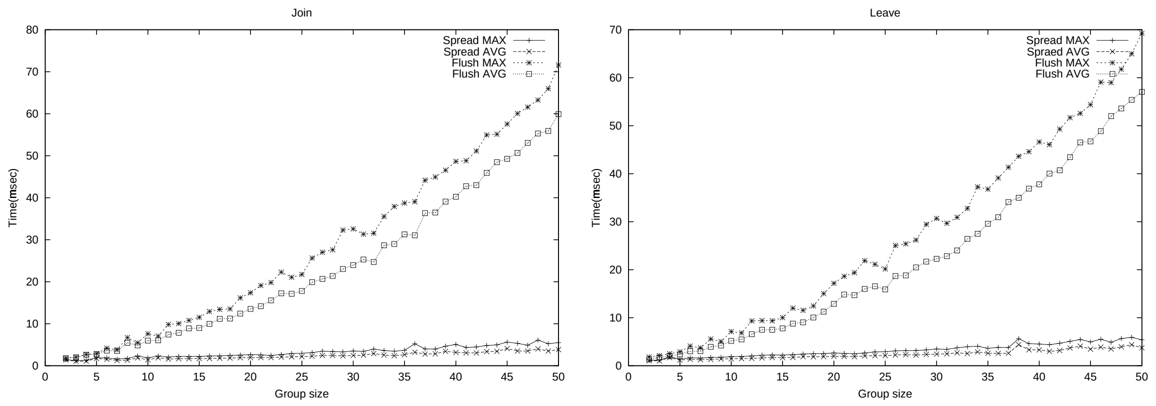

Figure 3. Timings - Spread and Flush

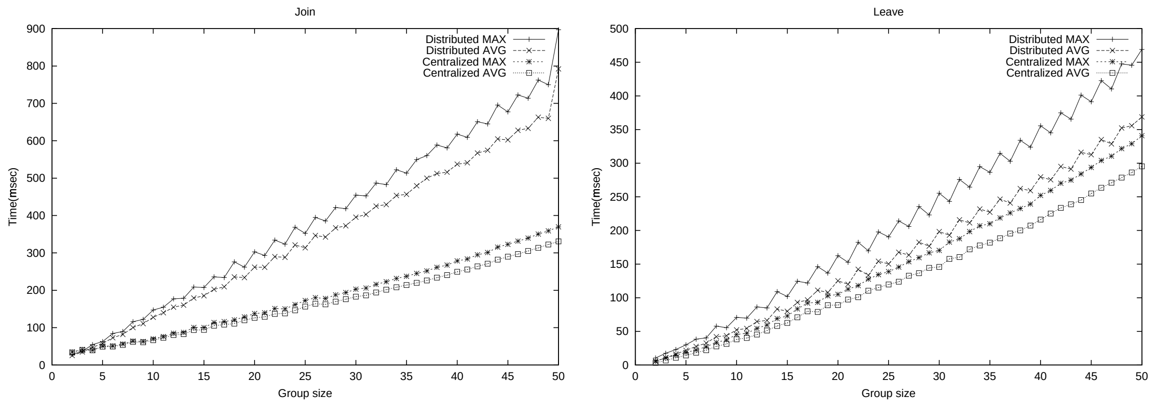

Figure 4. Timings - Centralized and Distributed Cliques Protocols

or total ordered), in the context where key agreement is an asynchronous event, some users might get a key before others. As we are dealing with an asynchronous system, once a member received the key he can securely multicast messages. If the message is sent FIFO, reliable or unreliable, there exists the possibility that other members will first see the message and only after it the membership that provides them with the new key. Restricting members of the group to send messages while they perform the key agreement (which is logical as they do not actually have a valid key) and buffering all the messages that a member can receive while waiting for the membership which provide them with a key, enables us to still provide all of the Spread services in the secure group communication environment in a correct manner.

4 Performance

This section presents an evaluation of the performance of our integrated secure group communication system. We are interested in two aspects: the first one is the time the group spends in establishing a new membership when external events happen (join and leave). The second one is the throughput of the system when data is encrypted and sent over the network.

We performed the all of the tests on a cluster consisting of four 450MHz Pentium II computers.

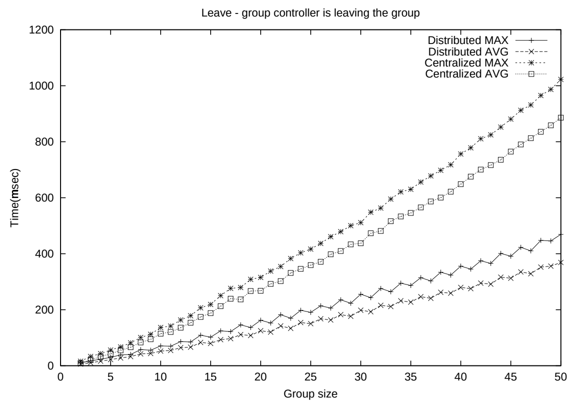

Figure 5. Timings - Centralized and Distributed Cliques Protocols, GC leaves the group

4.1 The cost of establishing a new membership - join and leave

Measuring the time the group spends in establishing a new membership is a difficult task. We are dealing with an asynchronous environment where different members of the group can see the same events at different times. In performing consistent and relevant tests two aspects are important: *what* time are we measuring and *who* is measuring it. From the cryptographic point of view, the performance of a key agreement algorithm is given by the time it takes to perform the exponentiations, so it seems that measuring CPU time will be a good approach. However, a more relevant measure for a group communication system would be the latency that a user experiences from the moment the group was affected by an event (new member joining the group or member leaving the group) until the new secure group membership is established.

While it is clear when an event is ending, it is unclear when it starts. Different members of the group take part in different phases of the key agreement. The secure group is not notified at the beginning of a membership event, it is only notified at the completion of the key agreement protocol. If we consider the Secure Spread layer a black box, then the events that can trigger a change in the group membership begin when a client invokes the secure group communication API (the join and leave primitives). In this context, having an outside observer that is not involved in the group communication system, playing the role of a timer is more natural. This approach overcomes the need to have the different computer clocks synchronized.

The timer process is notified by all the parties involved in the secure group membership protocol regarding the events affecting the group. We use unreliable UDP to signal these events. UDP packets, while not necessarily reliable, are usually very reliable in a local area network setting and have minimal latency, within two orders of magnitude smaller than our measured time.

In this way, the time is measured on a reference computer. Since the tests were performed on a 100Mb/sec local area network with a low collision rate this approach yields accurate results.

We conducted our experiments as follows: one of the four computers we used was chosen to run the timing program; the other three were running a Spread network (one daemon on each computer). The members of the group are distributed on two of the three computers running Spread. A user that joins and leaves the group is running on the third computer. The following notations will be used henceforth:

- TIMER: the program that keeps track of time.
- MEMBER: any member of the group, except the member that performs joins and leaves.
- TRIGGER: user that performs joins and leaves to an already established group of MEMBERS; its actions will trigger group events we want to evaluate.

The events that the timer program will see are:

- JOIN STARTED - the TRIGGER notifies TIMER immediately after joining the group.
- JOINED FINISHED - each MEMBER notifies the TIMER immediately after receiving the new membership message caused by the TRIGGER's joining the group.
- LEAVE STARTED - the TRIGGER notifies the TIMER immediately after leaving the group.
- LEAVE FINISHED - each MEMBER notifies the TIMER immediately after receiving the new membership message caused by the TRIGGER's leaving the group.

Note that, for every singular event (join or leave), the TIMER will see one EVENT STARTED and more than one EVENT FINISHED (as many as the group size, since different members of the group will see the same membership event at potentially different times), so the information the TIMER has is $started_{event}$ and $finished_{event}[1..N]$, N being the group size. For an outside observer the event took

$$Time_{max} = max(time_{event}[i])$$

where $time_{event}[i] = finished_{event}[i] - started_{event}$, $1 \leq i \leq N$.
However for a member of a group, the event took on average

$$Time_{avg} = \frac{\sum_{i=1}^{N} time_{event}[i]}{N}$$

We evaluated the time it takes to establish memberships for the following configurations: the EVS non-secure group communication system (Spread), the VS non-secure group communication system (Flush), and two different setups of secure group communication systems: Secure Spread using the Centralized key agreement protocol, and Secure Spread using a Distributed Cliques key agreement protocol.

Table 2. Total number of serial exponentiation

Operation	Join	Leave	Controller leaves
Number of members after operation	n	$n-1$	$n-1$
Distributed Cliques	$2n$	n	n
Cliques with Centralized key agreement	$n+6$	$n-1$	$3n-5$

As expected, the VS Spread implementation shows a linear increase with respect to the EVS Spread implementation(see Figure 3). The latency that a user using the Secure Spread system experiences when the group membership changes in reaction to a join or leave operation is presented in Figure 4. For a group size of 10 members, the time the group spends in establishing a new membership when a new member joins the group, increases from about 1 msec for the Spread layer to about 7 msec for the Flush layer, while the time needed to establish a new membership with Secure Spread with Centralized key agreement is about 70 msec and the time needed to establish the membership with Secure Spread with Distributed Cliques is about 140 msec. Leave operations are less expensive. For a group of 10 members, the time it takes to establish a new membership when a member leaves the group, increased from about 1 msec for the Spread layer to about 7 msec for the Flush layer, while the time needed to establish a new membership with Secure Spread using Centralized key agreement is about 45 msec and the time needed to establish a new membership using Secure Spread with Distributed Cliques is about 70 msec. As the group size increases, the time needed to establish a membership in Secure Spread increases linearly.

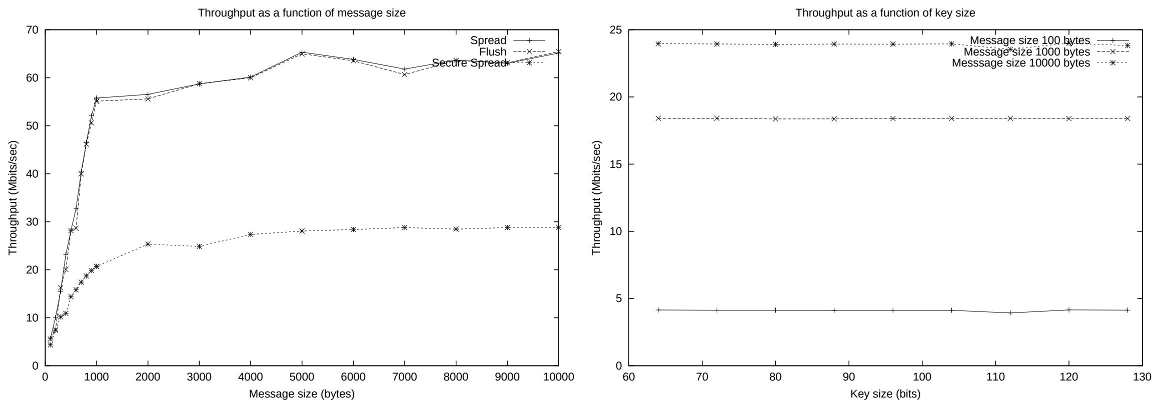

Figure 6. Timings - Throughput

Note that utilizing a centralized approach is less expensive. However, this solution does not allow for authentication of individual group members. Moreover, all members of the group need to trust the central member generating the keys and the quality of its random number generator. Figure 5 shows that the centralized mechanism is very expensive if the member leaving the group is the key generator.

The time values presented in Figure 4 include the time to establish a membership in Flush, the time to perform exponentiations, and the time to send the signaling UDP packets to the TIMER. Most of the overhead of the secure group comes from the number of serial exponentiations performed by the members while agreeing upon a new key. The number of serial exponentiations for both Centralized and Distributed Cliques are presented in Table 2. In theory, the last exponentiation step should be performed in parallel, therefore consuming one exponentiation time. However, in practice, the number of serial exponentiations can be slightly higher if more than one client will run on the same computer. In such cases the last exponentiation step will take more time than one exponentiation. For example, in the tests we performed for a group size of eleven, two of the spread daemons had five members each, so, instead of performing one final exponentiation, actually five were performed on each computer. As was shown in previous work [24], the measured CPU time tightly corresponds to the theoretical time derived from the number of serial exponentiations.

We did not perform tests on a wide area network but we expect that for a big latency, the relative time of a key computation compared with communication might change, as the cost will come not only from the exponentiations, but from communication latency as well.

4.2 The cost of encryption

The throughput of the Secure Spread system as a function of message size and as a function of key size is presented in Figure 6. It is interesting to note that the size of the key did not affect the throughput. Encrypting the messages lowered the throughput to about 40 percent of the throughput of the non-secure operation on a local area network, lowering it from about 65 Mbits/sec to about 28 Mbits/sec for a message size of 10000 bytes. We note that the decrease in performance will be almost negligible on slower wide area networks. Adding VS semantics practically costs nothing in throughput performance.

5 Conclusions

In summary, this report focuses on evaluating the cost of providing security services for a group communication system. Our tests show that the most important and costly aspect when providing secure group communication is the key agreement protocol.

We believe that a daemon-integrated architecture should be considered, as it will substantially increase the performance of the system and amortize the cost of expensive cryptographic operations.

Implementing secure and robust handling of cascading group events, using an approach optimized for the most frequent events (join and leave), is necessary in order to have a complete secure group communication system. Hardening security protocols to make them robust to asynchronous network events is difficult, but possible.

Finally, several necessary services are not discussed in this paper and could lead to interesting future work. They include services as group member certification, intra-group authentication, private communication within a group and private communication between members and non-members of the group.

Acknowledgements

The author thanks Yair Amir, Giuseppe Ateniese, Theo Schlossnagle, John Schultz, and Jonathan Stanton from the Department of Computer Science, Johns Hopkins University, Damian Hasse and Yongdae Kim from the Computers Network Division, USC Information Sciences Institute, and Gene Tsudik from the Information and Computer Science Department, University of California Irvine, who made the Secure Spread project possible. The valuable discussions and comments from Giuseppe Ateniese, Yair Amir, and Gene Tsudik helped in understanding better the interaction between security services and group communication systems. Many thanks to John Schultz, who wrote the Flush layer, for his help in implementing the Secure Spread library.

References

[1] Y. Amir and J. Stanton, "The spread wide area group communication system," Tech. Rep. 98-4, Johns Hopkins University Department of Computer Science, 1998.

[2] W. Diffie and M. Hellman, "New directions in cryptography," *IEEE Transactions on Information Theory*, vol. IT-22, pp. 644–654, Nov. 1976.

[3] M. Steiner, G. Tsudik, and M. Waidner, "Diffie-hellman key distribution extended to groups," in *3nd ACM Conference on Computer and Communications Security*, pp. 31–37, ACM Press, Mar. 1996.

[4] M. Steiner, G. Tsudik, and M. Waidner, "CLIQUES: A new approach to group key agreement," in *IEEE International Conference on Distributed Computing Systems*, May 1998.

[5] G. Ateniese, M. Steiner, and G. Tsudik, "New multi-party authentication services and key agreement protocols," in *IEEE Journal of Selected Areas in Communication*, 1999. Accepted for publication, to appear in early 2000.

[6] G. Ateniese, O. Chevassut, D. Hasse, Y. Kim, and G. Tsudik, "The design of a group key agreement api," in *Submitted to First International Workshop on Networked Group Communication*, July 1999.

[7] B. Schneier, "The blowfish encryption algorithm," *Dr. Dobb's Journal*, pp. 38–40, Apr. 1994.

[8] *Dr. http://www.counterpane.com/blowfish.*

[9] OpenSSL Project team, "Openssl," May 1999. http://www.openssl.org/.

[10] R. V. Renesse, K. Birman, and S. Maffeis, "Horus: A flexible group communication system," *Communications of the ACM*, vol. 39, pp. 76–83, April 1996.

[11] O. Rodeh, K. Birman, M. Hayden, Z. Xiao, and D. Dolev, "Ensemble security," Tech. Rep. TR98-1703, Cornell University, Department of Computer Science, September 1998.

[12] P. Zimmerman, *The Official PGP User's Guide*. prz@acm.org, 1994. The MIT Press In press. More in http://www.pegasus.esprit.ec.org/people/arne/pgp.html.

[13] O. Rodeh, K. Birman, and D. Dolev, "Optimized group rekey for group communication systems," in *Proceedings of ISOC Network and Distributed Systems Security Symposium*, February 2000.

[14] C. K. Wong, M. G. Gouda, and S. S. Lam, "Secure group communications using key graphs," in *Proceedings of the ACM SIGCOMM '98*, pp. 68–79, 1998.

[15] S. Mittra, "Iolus: A framework for scalable secure multicasting," in *Proceedings of the ACM SIGCOMM '97*, September, 1997.

[16] M. K. Reiter, "Secure agreement protocols: reliable and atomic group multicast in Rampart," in *Proceedings of the 2nd ACM Conference on Computer and Communications Security*, November, 1994.

[17] K. P. Kihlstrom, L. E. Moser, and P. M. Melliar-Smith, "The securering protocols for securing group communication," in *Proceedings of the IEEE 31st Hawaii International Conference on System Sciences*, vol. 3, (Kona, Hawaii), pp. 317–326, January 1998.

[18] P. Narasimhan, K. P. Kihlstrom, L. E. Moser, and P. M. Melliar-Smith, "Providing support for survivable corba applications with the immune system," in *Proceedings of the 19th IEEE International Conference on Distributed Computing Systems*, (Austin, TX), pp. 507–516, May 1999.

[19] J. G. Steiner, C. Neuman, and J. I. Schiller, "Kerberos: An authentication service for open network systems," in *Usenix Winter Conference*, pp. 191–202, Jan. 1988.

[20] L. E. Moser, Y. Amir, P. M. Melliar-Smith, and D. A. Agarwal, "Extended virtual synchrony," in *Proceedings of the IEEE 14th International Conference on Distributed Computing Systems*, pp. 56–65, IEEE Computer Society Press, Los Alamitos, CA, June 1994.

[21] Y. Amir, "Replication using group communication over a partitioned network", Ph.D Thesis from the Institute of Computer Science, The Hebrew University of Jerusalem, 1995.

[22] A. Fekete, N. Lynch and A. Shvartsman, "Specifying and using a partionable group communication service," in *Proceedings of the 16th Annual ACM Symposium on Principles of Distributed Computing*, pp. 53–62, Santa Barbara, CA, August 1997.

[23] Y. Amir, C. Danilov and J. Stanton, "A low latency, loss tolerant architecture and protocol for wide area group communication," *Accespted to FTCS-2000*.

[24] Y. Amir, G. Ateniese, D. Hasse, Y. Kim, C. Nita-Rotaru, T. Schlossnagle, J. Schultz, J. Stanton, and G. Tsudik "Secure group communication in Asynchronous Networks with Failures: integration and experiments," in *Proceedings of the 20th IEEE International Conference on Distributed Computing Systems*, pp. 330–343, Taipei, Taiwan, April 2000.