The Architecture and Performance of Security Protocols in the Ensemble Group Communication System

Using Diamonds to Guard the Castle

Ohad Rodeh, Kenneth P. Birman, Danny Dolev*

October 17, 2000

Abstract

Ensemble is a Group Communication System built at Cornell and the Hebrew universities. It allows processes to create *process groups* within which scalable reliable fifo-ordered multicast and point-to-point communication are supported. The system also supports other communication properties, such as causal and total multicast ordering, flow control, etc.

This paper describes the security protocols and infrastructure of Ensemble. Applications using Ensemble with the extensions described here benefit from strong security properties. Under the assumption that trusted processes will not be corrupted, all communication is secured from tampering by outsiders.

Our work extends previous work performed in the Horus system (Ensemble's predecessor) by adding support for multiple partitions, efficient rekeying, and application defined security policies. Unlike Horus, which used its own security infrastructure with non-standard key distribution and timing services, Ensemble's security mechanism is based on off-the shelf authentication systems, such as PGP and Kerberos.

We extend previous results on group rekeying, with a novel protocol that makes use of diamond-like data structures. Our Diamond protocol allows the removal of untrusted members within milliseconds.

*The authors were supported in part by DARPA ITO under ARPA/ONR contract N0014-96-1-10014 and ARPA/RADC contract F30602-95-1-0047, in part by grants from Microsoft and AT&T, and by the Israeli Ministry of Science grant number 032-7892.

1 Introduction

Group Communication Systems (GCSs) are used today in industry where reliability and high-availability are required. Group Communication is a subject of ongoing research and many GCSs have been built throughout the world [6, 47, 23, 30, 3, 40, 41, 20, 49]. Example GCS applications include: group-conferencing, distributed simulation, server replication, and more (see [4]). In industry, GCSs are mainly used for cluster management. The SP2, and other IBM clustering systems use the Phoenix system [7], IBM AS/400 clusters use the Clue system [11], and Microsoft Wolfpack clusters use Group Communication technology at the core of their system [43]. A secure GCS is one protected against malicious behavior or outright attack. This paper describes the security architecture of Ensemble [40], our group communication system, which achieves the desired properties.

Ensemble was developed at Cornell and the Hebrew universities. It is written in a dialect of the ML programming language [46] in order to facilitate system verification. The design methodology behind Ensemble stresses modularity and flexibility [17]. Thus, Ensemble is divided into many *layers*, each implementing a simple protocol. Stacking these layers together, much like one uses lego blocks, the user may customize the system to suit its needs.

Ensemble supports the *fortress* security model. In this model, the "good guys" are protected by a castle from the hordes of barbarians outside. In the context of a GCS this corresponds to a situation where the (honest) group members need this walls to protect them from the adversary. Since the members are distributed across the network, they cannot be protected by a firewall. An alternative vehicle for protection is the use of cryptography. One alternative is to use public keys to encrypt all group messages. However, public-key cryptography is roughly 1000 slower than symmetric key cryptography, making it prohibitively expensive. The second alternative, which we have chosen, is to dynamically agree upon and use a shared symmetric key. Assuming all members agree on the same key, all group messages can be MAC-ed[1] and encrypted. If we assume that the adversary has no way of retrieving the group-key, the members are protected.

The group-key needs special handling as it can be distributed *only* to authenticated and authorized group members. This raises two challenges:

A rekeying mechanism: This is the problem of secure replacement of the current group key once it is deemed insecure, or if there is danger that it was leaked to the adversary. Rekeying is challenging since switching to a new key must occur without using the old, possibly compromised key for dissemination. Naturally, one could use public keys for this task, yet doing so leads to high latency.

If one assumes the simple "primary partition" model, where only a single component of the group may function, then a simple solution is available. In this case it suffices to designate a centralized key-server which will have responsibility for disseminating, revoking, and refreshing group keys. Only group members in contact with the server will have access to the key and hence be capable of functioning.

Our work is the first to support the more general multiple-partition model first suggested by Dolev et al. in [9]. Supporting multiple-partitions is more difficult since one cannot rely on any centralized service.

Secure key agreement in a group: This is the problem of providing a protocol whereby secure agreement can be reached among group members which need to select a mutual key. Such a

[1] MAC is a Message Authentication Code algorithm. As we explain later, the group key is split into two portions: one used for MAC-ing, the other for encryption.

protocol should not restrict the Ensemble protocol stack, i.e., all previously legal combinations of layers should still be possible, it should be unobtrusive, and support multiple partitions. That is, the protocol should "compose" cleanly with Ensemble stacks, regardless of their functionality.

Our protocol must efficiently handle the case where two group components merge after a network partitioning, where the network partitions into two or more components, and the resulting group components use different keys. A simple approach (taken for example in [8]) is to add members one by one, in effect transferring them from the smaller group to the larger one. However, this is potentially slow since members are added one at a time; it incurs cost quadratic in the number of added members: O(n) members × join protocol (which is also O(n)). Our solution is much more efficient.

We focus on benign failures and assume that authenticated members will not be corrupted. Byzantine fault tolerant systems have been built by other researchers [38, 22], but suffer from limited performance since they use costly protocols and make extensive use of public key cryptography. We believe that our failure model is sufficient for the needs of most practical applications. As demonstrated in the performance section, our system has good performance and scalability.

Our contributions are:

- We demonstrate how security properties can be decomposed and introduced to a layered protocol architecture.
- We support security properties for multiple partitions. Earlier work either does not address the issue of group partition or only supports security semantics for the primary partition [41].
- We provide support for dynamic application-defined authorization polices.
- We use a novel algorithm that allows removing untrusted members in a matter of milliseconds.

Our security architecture is composable with most other Ensemble layers. The user thus has the freedom to combine layers and properties including security.

2 Model

Consider a universe that consists of a finite group $\mathcal{U}$ of n processes. Processes communicate with each other by passing messages through a network of channels. The system is asynchronous: clock drifts are unbounded and messages may be arbitrarily delayed or lost in the network. Processes may crash and later restart.

To model both network and process failures, we use the *partitioning model*. Sets of processes may become *partitioned* from each other. A partition occurs when $\mathcal{U}$ is split into a set $\{P_1, \ldots, P_k\}$ of disjoint subgroups. Each process in P_i can communicate only with other processes in P_i. The subsets P_i are sometimes called *network-components*. We consider an extended *dynamic partitions* model [9] , where in network-components dynamically merge and split. A process crash can be modeled as a partition, and a restart can be modeled as a merge, hence, in our algorithms we mainly consider partition and merge scenarios.

As described earlier a GCS creates process groups in which reliable ordered multicast and point-to-point messaging is supported. Processes may dynamically join and leave a group. Groups may dynamically partition into many *components* due to network failures/partitions; when network partitions are healed group components remerge through the GCS protocols. Information about groups is provided to group members in the form of *view* notifications. For a particular process p a *view* contains the list of processes currently alive and connected to p, ordered lexicographically. In a distributed asynchronous system, it is not possible to provide accurate views, therefore, the notifications provided to processes can only be an approximation of the actual set of connected processes. Under "normal" network conditions, the view reflects the actual network connectivity, under heavy load, and link fluctuations, the view is a mere approximation. When a membership change occurs due to a partition or a group merge, the GCS goes through a (short) phase of reconfiguration. It then delivers a new view to the applications reflecting the (new) set of connected members.

In what follows p, q, and s denote Ensemble processes and V, V_1, V_2 denote views. The generic group is denoted by G. G's members are numbered from p_1 to p_n.

For this paper, we focus on messages delivered in the order they were sent: a "fifo" or "sender-ordered" delivery property.

Ensemble follows the Virtual Synchrony (VS) model. This model describes the relative ordering of message deliveries and view notifications. It is useful in simplifying complex failure and message loss scenarios that may occur in distributed environments. For example, a system adhering to VS ensures "atomic failure". If process q in view V fails then all the members in $V \setminus \{q\}$ observe this event at the "same time".

To achieve fault-tolerance, GCSs require all members to actively participate in failure-detection, membership, flow-control, and reliability protocols. Such protocol implementations have inherently limited scalability. We have managed to scale Ensemble to a few hundred members per group, but no more. For a detailed study of this problem, the interested reader is refered to [21, 4]. In this paper, we do not discuss configurations of more than a hundred members.

We assume that the processes in a group have access to trusted authentication and authorization services, as well as to a local key-generation facility. We also assume that the authentication service allows processes to sign messages. This means that process p can send an authenticated message M to process q, and q can verify that M has been signed by p. Furthermore, no attacker can modify M without damaging the signature. To denote that message M has been signed by member p for member q we write $[M]_{S_{pq}}$. In particular only q can verify this message. This requirement allows us to use systems such as Kerberos which create certificates that are good only for point-to-point communication.

The adversary has access to all untrusted (potentially dishonest) machines and may corrupt or eavesdrop on any packet traveling through the network. Our goal is to protect messages sent between trusted members of $\mathcal{U}$. We do not provide protection against denial of service or traffic analysis attacks. Rather, we restrict ourselves to the authenticity and secrecy of message content. We work with an existing operating system and assume its security and correctness. An OS vulnerability or a compromised authentication service would cause a breach in Ensemble security.

Each member decides on its own trust policy (more on this in section 3). Ensemble is responsible for enforcing this trust policy, making sure that only mutually trusting members enter the same group component. Note that G may now be split into network components, and further into mutually distrusting sub-components.

The system should support Perfect Forward Secrecy (PFS) with respect to the group key. Briefly, this means that past members cannot obtain keys used in the future. The dual requirement that current members cannot obtain keys used in the past is termed Perfect Backward Secrecy (PBS).

To support PFS, the group key must be switched every time members join and leave. This may incur high cost, therefore, we attempt to relax PFS without breaking security. To this end, a three pronged approach is used:

- Fast rekeying algorithms are included in the system.
- The system rekeys itself one every 24 hours, to prevent cryptanalysis.
- Apart from the above, rekeying is a user initiated action. The user should decide when to switch the group key, trading off security against performance.

Applications specify the list of trusted members to Ensemble (see more on this in section 3.1), in the form of an Access Control List (ACL) which is treated as replicated data within the group. Ensemble allows any trusted member into a group without rekeying. Ensemble allows members to change the group ACL at runtime, for example, by multicasting totally ordered updates to it, state-machine style. An ACL can become more restrictive, corresponding to the removal of present group members. First, such members are removed from the group, and then, a rekey operation is performed. During the intervening period, an untrusted member still holds group keys and hence may represent a security breach. We assume that such a process does not leak information.

To explain our intuition, examine a meeting of military staff that have support from civilian advisors. During the low-security part of the meeting, low-security information is revealed to the advisors. Later, the meeting is declared classified, civilians leave the room, and the military staff continue the meeting using high-security information. Until the civilians have left, high-security information is not revealed.

In a group setting, the initial ACL G_1 includes both military personal and civilians, and all group information is encrypted in key K_1. Later, the ACL is changed to G_2, including only military personal. Civilians leave the room, and the group is rekeyed to key K_2. The rest of the meeting, now classified, is encrypted with K_2. The civilians are trusted to behave according to group protocols while they are in the group, and to leave the group when ordered to do so. They are also expected not to reveal information encrypted with K_1 to members outside G_1.

To summarize, 1) Ensemble does not rekey itself, but leaves it up to the user to decide when to rekey, normally as part of an ACL management policy. 2) Fast rekey protocols are implemented. 3) Rekeying is not performed when trusted members join. For a thorough discussion of security policies in a group context see the Antigone system [33].

3 Ensemble

Ensemble is a GCS supporting process groups as described above. In addition to reliable fifo-ordered multicast and point-to-point communication, it also supports many other protocols and communication properties such as: multicast total order, multicast flow control, protocol switching on the fly, several forms of failure detection, and more (see [40] for more details).

Ensemble is typically configured as a user-level library linked to the application. It is divided into many *layers*, each implementing a simple protocol. Applications may customize the Ensemble library to use the set of layers they require: the set of layers desired is composed into an Ensemble *stack*. All members in a group must have the same stack to communicate.

Ensemble keeps *view-state* information. This information is replicated at all group members and includes such data as: current protocol stack in use, group member names and addresses, the number of members, the group key, etc. In order to change any of this information, a new view has to be installed, under control of a view agreement protocol.

In Ensemble, each view has a unique leader known to all view members. The leader is selected automatically by ranking group members, and the VS model ensures that, in a given view, all members have a consistent belief concerning which member is the leader.

If the group key needs to be changed the group will be prompted for a view change. During the process the leader will broadcast the new view-state, which includes the new group key. All members will then begin to use the new group key in the upcoming view.

Ensemble divides messages into two classes. There are *intra-group* or *regular* messages sent between members of a view. These are usually application-generated messages, though some may also be generated as part of the Ensemble protocols on behalf of the application. In addition, there are *inter-component* messages or so-called *gossip* messages. These are messages generated by Ensemble for communication between separate components of a partitioned Ensemble group. Recall that Ensemble is designed to detect partitioning and to merge partitioned groups when network connectivity is restored. To this end, Ensemble periodically multicasts a gossip message to $\mathcal{U}$, to anyone who can hear. Normally, communication is not possible between separate group components due to network partitions. Reception of a Gossip message thus triggers the components merge protocol, whereby separate components are fused together. Ensemble protocols that use gossip messages make very few assumptions about them: they may be lost, reordered, or be received multiple times.

The regular and secure Ensemble stacks are depicted in Table 1. The Top and Bottom layers cap the stack from both sides. The Group Membership Protocol (GMP) layer[2] computes the current set of live and connected machines. The Appl_intf layer interfaces with the application and provides reliable send and receive capabilities for point-to-point and multicast messages. It is situated in the middle of the stack to allow lower latency to user send/receive operations. The RFifo layer provides reliable per-source fifo messaging.

The Exchange layer guarantees secure key agreement throughout the group. Through it, all members obtain the same symmetric key for encryption and signature. The Rekey layer performs group rekeying upon demand. At the time of this writing, four different rekeying layers are implemented, see section 5 for more details on the *diamond* algorithm. The SecChan layer provides a secure point-to-point messaging service to Rekey. These three layers manage the group-key within the view-state and hence are regarded as GMP extensions. Furthermore, these layers are not on the message critical path. Normally, they are dormant, they become active either when the user asks for a rekey or when components merge. The Encrypt layer encrypts all user messages. It is on the message critical path, situated below the Appl_intf layer.

3.1 Policies

The user may specify a security policy for an application. The policy specifies for each address[3] whether or not that address is trusted[4]. Each application maintains its own policy, and it is up to Ensemble to enforce it and to allow only mutually trusted members into the same component.

A security policy can also be viewed as a list of trusted addresses, or an *Access Control List* (ACL). We shall use this notation interchangeably.

---

[2]Some "layers", as discussed here, are actually sets of layers in the implementation. Also, some layer names have been changed for clarity of exposition.

[3]An Ensemble address is comprised of a set of identifiers, for example an IP address and a PGP principal name. Generally, an address includes an identifier for each communication medium the endpoint is using {UDP,TCP,MPI,ATM,..}.

[4]We shall see later, in section 4 how the authenticity of members' addresses is ensured.

Regular	Secure
Top	Top
	Exchange
	Rekey
	SecChan
GMP	GMP
Appl_intf	Appl_intf
	Encrypt
RFifo	RFifo
Bottom	Bottom
Routers	

Table 1: **The Ensemble stack. On the left is the default stack that includes an application interface, the membership algorithm and a reliable-fifo module. The secure stack, to the right, includes all the regular layers (shown in pale gray) and also the Exchange, Rekey, SecChan, and Encrypt layers. The critical path for application messages is between Appl_intf and Bottom.**

As we shall see in section 4, due to efficiency considerations, members should use trust policies that are symmetric and transitive. Other types of policies are currently not supported under Ensemble. When a member changes its security policy, it requests Ensemble to rekey. During the rekey members that are no longer trusted will be excluded and a new key will be chosen for the component. Thus, old untrusted members will not be able to eavesdrop on the group conversations.

3.2 Cryptographic infrastructure

Our design supports the use of a variety of authentication, signature and encryption mechanisms. By default the system uses PGP for authentication, MD5 [36] for signature, and RC4 [37] for encryption. Because these three functionalities are carried out independently any combination of supported authentication, signature, and encryption systems can be used.

Insofar as freeware cryptographic libraries are available, we prefered to interface with them, rather than coding our own implementations of the various MAC, encryption, and authentication algorithms. As cryptographic standards and algorithm progress and evolve, this permits us to easily keep apace. Currently, an interface with OpenSSL [27] allows using RC4, DES [13], IDEA [45], and Diffie-Hellman [44]. For authentication, in addition to PGP [34], we have a Kerberos [2] interface, though it is out of date.

3.3 Random number generation

Cryptographically secure random numbers are vital to any secure system. It is not possible to generate truly random numbers and therefore one uses pseudo-random number generators. We have plugged in an off-the-shelf, cryptographically strong, random number generator to our system [19].

3.4 The group key

The group key is split into two separate 16-byte subkeys, one used for keyed-hashing, and the other for encryption. The MAC-router (see below) uses the hashing sub-key, while the Encrypt layer (see below) uses the encryption sub-key.

Spliting the group key into two subkeys increases security. The encryption and hashing methods known today are not perfect, and none have rigorously been proven secure, hence all have weakness some of which are known. Had we used a single key for both functionalities, the attacker would have been able to exploit weaknesses from both systems to break the key. Using a different key for each system forces the attacker to break one of the systems without "help" from the other, a task considered very difficult today.

3.5 The MAC router

We first describe the simplest part of the security architecture: the *MAC router* module. Ensemble routers reside at the bottom of each protocol stack, as seen in Table 1.

In Ensemble, the router is the module responsible for getting messages from member p to some set of members $\{q_1 \dots q_k\}$. Routers use transport-level protocols such as MPI, UDP, TCP, and IP-multicast to send and receive messages. An Ensemble application may use several stacks, all sharing a single router. Hence, routers need to decide through which transport to send a message, and when one is received — which protocol stack to deliver it to.

We have modified the normal router to create a *MAC router* which is used when the application requests a secure protocol stack. The MAC router uses a cryptographically secure one-way hash function, MD5, to hash the message content. MD5 is keyed with the current group key such that the adversary will not be able to forge messages. The router at the sender calculates the keyed hash of M — $H(M)$. Then it sends $H(M)$ concatenated to the clear-text message M. On receipt, $H(M)$ is recalculated from M with the receiver's key and compared with the received hash value. If there is a match — the message has been verified.

All outgoing messages are MAC-ed using the group key. Regular messages may be verified by other group members since they all share the group key. Gossip messages are problematic since, initially, different components do not share the same group key. Hence, they are protected using the authentication service.

When message m arrives at a MAC-router, belonging to group component A, the router attempts to verify m using the group key. There are several cases:

m is a regular message:

1. Correct hash: Pass up the stack. Message m was sent by a group member in A.
2. Incorrect hash: Drop. Message m may come from a different group component that shares no key with A. It may also be a message sent by an attacker (that does not know the key).

m is a gossip message:

1. Correct hash: Pass up the stack. Message m is of gossip type, it was sent by a member of a different component that shares the same group key.
2. Incorrect hash: Mark as *insecure* and pass up the stack. This is a message from a different component B that is signed with B's group key. We ignore the keyed-MD5 signature, since we cannot verify it. Possibly, the inner message is signed by the authentication service. The Exchange layer will attempt to verify it, if successful, it will process m's contents. Exchange is the only layer that examines such messages, while other protocol layers that use gossip messages ignore *insecure* gossip messages.

To summarize, the MAC router attempts to authenticate all messages. Regular unauthenticated messages are dropped, gossip unauthenticated messages are still delivered but marked *insecure*.

3.6 The Encrypt layer

Ensemble optionally supports user message privacy. The Encrypt layer encrypts/decrypts all user messages with the group key. Note that only the encryption sub-key is used. User messages are reliably delivered in fifo (sender) order allowing use of chained encryption[5]. Ensemble messages are MAC-ed, but not encrypted. Such messages do not contain any secret user information and their encryption would only degrade performance. To improve performance, upon a view change we create all security-related data structures and henceforth use them while the view remains current.

3.7 The SecChan layer

The SecChan layer provides a *secure channel* abstraction. A secure channel allows two members to exchange private information. The layer maintains a cache of secure channels that is updated on demand. SecChan allows layers above it to send private point-to-point messages to other members.

When member p's SecChan layer receives a private message m to be passed to member q then the cache is queried. If a secure channel to q with key K_{pq} already exists then m is encrypted with K_{pq} and sent pt-2-pt to q. If the channel does not exist, then a Diffie-Hellman handshake protocol is used to securely agree on a key K_{pq} between p and q. The channel is added to the cache, and m is encrypted with K_{pq} and sent to q.

The cache has a very simple structure, using two rules: (1) only connections to present group members are kept. (2) The cache is flushed every 24 hours[6] to prevent cryptanalysis.

To measure how expensive a Diffie-Hellman exchange is, we used a 500Mhz PentiumIII with 256Mbytes of memory, running the Linux2.2 OS for speed measurements. An exponentiation with a 1024bit key using the OpenSSL cryptographic library was clocked at 40 milliseconds. Setting up a secure channel requires two messages containing 1024bit[7] long integers, where both sides perform an exponentiation.

In light of this, we view the establishment of secure channels as expensive, in terms of both bandwidth and CPU. This is the rational for caching connections at the SecChan layer.

3.8 Performance assessment

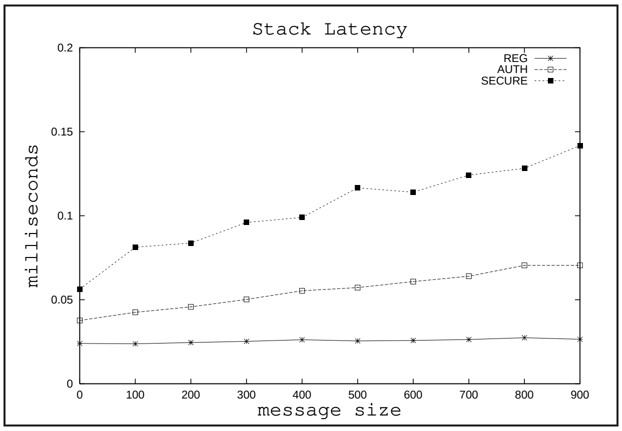
In this section the performance of our security subsystem is described. Our test bed is a set of 20 PentiumIII 500Mhz Linux2.2 machines, connected by a switched 10Mbit/sec Ethernet. The machines were lightly loaded during testing, and the network was, for the most part, clear of other traffic. Throughput and latency were measured. Each measurement was taken three times, once with a standard, insecure stack (REG), second, with an authenticated stack (AUTH) , and third, with an authenticated encrypted stack (SECURE). The encryption used was the default RC4.

Figure 1(1) shows the latency for a send/recv operation inside the stack. This is a "ping" test in which a message is received, and an immediate response is sent back to the origin. The amount of time spent inside the Ensemble stack is measured. As we can see, the regular and authenticated stack are quite close, meaning that the computational overhead of an MD5 hash over a message is not significant. On the other hand, the encrypted stack is relatively expensive. As message size grows the computation required grows. Note that the base line is a low and constant 24 milliseconds. Hence, the basic overhead imposed by the system is very low. Furthermore, the cost

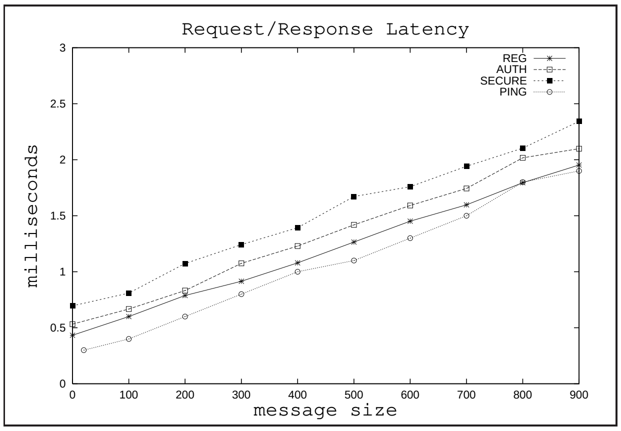
[5] Modern encryption ciphers separate a message into fixed sized blocks. One can encrypt each block separately, or, using *chained encryption*, use early blocks to help encrypt the current block.

[6] This is a settable parameter

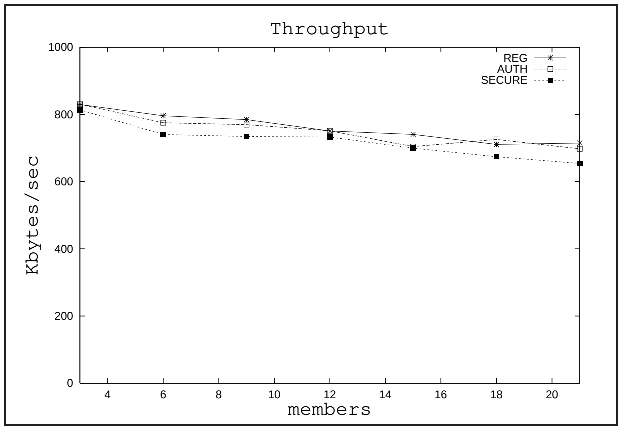
[7] At the time of writing this is considered secure.



(1)



(2)



(3)

Figure 1: **A standard stack is denoted REG. An authenticated stack is denoted AUTH. An authenticated encrypted stack is denoted SECURE. (1) send-recv latency in the Ensemble stack. (2) Total Latency for point-to-point send/recv. (3) Throughput.**

difference between the different stacks is almost entirely due to the MAC and encryption algorithms, not to the layering structure.

Figure 1(2) shows the latency of a “request/response” scenario. Two machines using Ensemble are used. The initiating machine sends a point-to-point message to the second machine, which sends back an immediate response. This scenario was repeated 1000 times, and various message sizes were used. A comparison with the Unix `ping` utility was also included. As we can see, the difference between the three stacks is not very significant. Furthermore, the standard stack is fairly close `ping`. We conclude that the latency is mostly due to the network and operating system.

Figure 1(3) shows the throughput achievable in a lightly used network. The maximal bandwidth in a 10Mbit/sec Ethernet is 1.2Mbyte/sec. Out of the maximum, throughput of 750Mbyte/sec can be achieved for a 21 member group. The loss of bandwidth is attributed to the high level of guarantees provided. In order to achieve reliability, one must perform retransmissions, to achieve sender-order multicast, messages must be number, for flow-control, a back-off protocol must be used etc.

4 Secure Merge – The Exchange protocol

In the event of a network failure, a process group may become partitioned into several disjoint components, communication among which is impossible. Ensemble automatically elects a leader for each group component. Later, such a partitioned group may need to merge if communication is restored. Ensemble treats the former situation as the failures of one or more group members (the system does not distinguish communication failures to operational processes from process crashes). The system uses gossip messages to discover opportunities to merge a group.

More specifically, it is the responsibility of the Heal protocol, part of the standard Group Membership protocol, to discover partitioned group components. It is active at each group component leader. Each leader gossips a multicast *IamAlive* message periodically that includes its name and address. When a leader hears a remote leader from the same group, it initiates the merge sequence.

Group components cannot communicate with each other unless they possess the same key: only insecure gossip messages are allowed to pass through by the router. The Exchange layer uses these messages to achieve secure agreement on a mutual group key. The idea is that one of the components securely switches its key to that used by the other component. The Heal layer will activate the merge sequence after both components have the same key. The Exchange layer is active at each component leader acting as a filter of gossip messages. All outbound/inbound gossip messages pass through it.

While a merge protocol is taking place, components continue to distrust each other. It is possible that the merger is actually an attacker that is attempting to waste local computational resources, or attempting to maliciously modify the protocol and cause the system to crash. Therefore, the protocol should be as simple and foolproof as possible. Specifically, it should be:

- Stateless: a component may engage in multiple AKEs concurrently. Each such authenticated exchange is termed a *session*. Since an attacker may engage in several sessions with an honest component, it is desirable not to keep per-session state.
- No synchronized clocks: The protocol should not use synchronized clocks, since this requires some fault-tolerant clock synchronization service.
- No reliability: since we wish to avoid session-state, in particular we cannot use reliable messaging. This would require a numbering scheme for messages, and a retransmission mechanism. All messages sent in Exchange are gossip, non-reliable messages.

Exchange is similar to the well-known Bellare-Rogaway authenticated key exchange scheme [26]. Our protocol also employs the Diffie-Hellman exchange [44]. First, we describe a naive version of the protocol, which is not resilient to replay attacks. Then, we enhance the protocol to handle such attacks.

4.1 Naive version

The layer functions via the creation and recognition of three types of messages and headers. These are, for process p whose *principal name*[8] is R_p, and whose view key is K_p:

Id: This is a header added to an outgoing gossip message. It contains R_p. This header is cheap to create.

[8] This is the name, by which the user is known to the authentication service.

Ga,Gb: Point-to-point gossip messages. Contain data to be sent securely to some process p. These messages are created by sealing the data for p. The header is expensive to generate, since its creation involves the authentication service, and it is usually long (currently about 1/2KBytes).

The layer maintains four constant fields: (1) the public 1024bit prime modulo n (2) a generator g for the finite field Z_n (3) a random value $v \in Z_n$, chosen upon initialization (4) A pre-computation of $g^v \ mod \ n$. All members in G are initiated with the same values for n and g. We denote member s's value v by v_s.

The following event handlers are applied to gossip messages by process q, when q is leader of its component:

- Onto each gossip message, add an Id header.
- Upon receiving an Id(R_p), if it is *insecure*, p is trusted, and $R_q < R_p$[9], then create a message: Gb($[q, p, g^{v_q}]_{S_{qp}}$), and send it to p.
- Upon receiving a message Gb(...) from p, check that (1) the message is intended for q (2) p is allowed to join (3) the signature is correct. If all conditions are true, then extract g^{v_p} and send back to p: $Ga([q, p, g^{v_q}, \{K_p\}g^{v_p v_q}]_{S_{qp}})$.
- Upon receiving a message Ga(...) from p, check that: (1) the message is intended for q (2) p is allowed to join (3) the signature is correct. If all conditions are true, extract g^{v_p} compute $g^{v_p v_q}$, and decrypts K_p. If $K_p == K_q$, then ignore it (we have the same key), otherwise *new_key* $:= K_p$. Prompt the component to go through a view change, with *new_key* as the group key. The group key is part of the view-state, when the view change is complete *new_key* will be installed at all the group's routers.

  Note that the new key is encrypted with K_q before being multicasted as part of the view state. Only members of q's component will be able to decrypt and install K_p.

Figure 2 describes an example run, where member p is the leader of component P with group key K_p, and member q is the leader of component Q with group key K_q. Member q receives p's IamAlive message, and initiates an authenticated Diffie-Hellman exchange, in which p hands over to q its component key K_p.

In the example, members p and q keep secrets v_p and v_q respectively. In general, each leader s keeps a random value $v_s \in Z_n$ for which it precomputes g^v. This value is not revealed by distributing g^{v_s}, nor by raising it to the power of $w \in Z_n$. This allows us to employ a single value v_s for all sessions. To increase security v_s can be switched periodically.

In order to reveal a group-key K, the attacker can either: (1) break a Diffie-Hellman exchange in which K was transfered, or (2) Discover some other group key K' with which K was encrypted. The signature keys are not used to protect actual group keys. This allows using long-term signature keys, such as those afforded by Public Key Infrastructures (PKIs).

Note that messages may get lost in Exchange. While *reliable* key-exchange would be preferable to a non-reliable one, this is sufficient for our purposes. For example, assume q loses p's last (third stage) message in the above example. Member q will be able to restart the protocol when it receives p's next IamAlive message. In other words, Exchange is *reentrant*.

[9] Any type of comparison function may be used here.

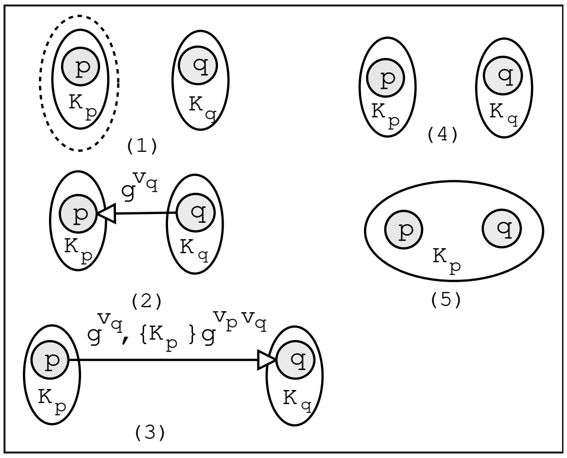


Figure 2: **Overview of the merge sequence. (1) Two components with keys K_p and K_q. Each leader sends IamAlive messages. (2) q sends g^{v_q} to p. (3) p sends g^{v_p} and K_P encrypted with $g^{v_p v_q}$ to q. (4) q decrypts K_p and switches his component key to K_p. (5) Components P and Q now merge, after both have the same key.**

4.2 Handling replay attacks

We now strengthen the protocol to handle replay attacks. The layer at member s keeps track of its local time in a variable lt_s, and maintains a function f_s (see below). The Bellare-Rogaway protocol requires using unpredictable nonces, in practice, we use a pseudo-random function f_s, that takes the local time, and creates a 64-bit pseudo-random value from it. Below, we use the notation $t_s = (lt_s, f_s(lt_s))$. This means, for member s, a nonce created by concatenating the local time in s, lt_s, with $f_s(lt_t)$. The function f_S is different for each member, and it is not revealed to other members.

The event handlers from member q, where q is a component leader are:

- Onto each gossip message, add an Id header. We modify slightly the format of the message to: IamAlive(R_q, t_q).
- Upon receiving an Id(R_p, t_p), if it is *insecure*, p is trusted, and $R_q < R_p$[10], then create a message: Gb$([q, p, t_q, t_p, g^{v_q}]_{S_{qp}})$ and send it to p.
- Upon receiving a message Gb(...) from p, check that (1) the message is intended for p (2) p is allowed to join (3) the signature is correct (4) The nonce from q is fresh. If all conditions are true, then extract g^{v_p} and send back to p: Ga$([q, p, t_q, g^{v_q}, \{K_q\}g^{v_p v_q}]_{S_{pq}})$.
- Upon receiving a message Ga(...) from p, check that: (1) the message is intended for q (2) p is allowed to join (3) the signature is correct (4) the nonce from q is fresh. If all conditions are true, then proceed as in the naive protocol.

A member must verify that a specific value $t'_s = (lt'_s, z)$ was generated by itself at a time close to the current time. To verify this, member s checks that $lt_s - lt'_s < 10seconds$[11] and that

[10] Any type of comparison function may be used here.
[11] 10 is a parameter, this can be changed by the application.

$z = f_s(lt'_s)$. This is a simple computation that only requires remembering f_s. This was designed to allow avoiding any need for perf-session state.

4.3 Access Control Lists

In the above exposition, we have not discussed specific ACL considerations. Our AKE provides only a certain level of authorization checking. If leaders p and q are in each other's ACLs then components P and Q will merge. This assumes that the ACL is symmetric and transitive, i.e., an equivalence relation. While it is possible to allow each member in P to check all members in Q, and vice-versa, we have decided this was too expensive in terms of communication and computation.

In Ensemble, it is up to the application developer to make sure that the ACL is in fact an equivalence relation. While this may sound impractical, there are simple ways of implementing this. For example, the designer could employ a centralized authorization server, or simply a static ACL. Such an ACL must separate the list of trusted hosts into several disjoint subgroups. Trust is complete in each subgroup, but no member trusts members outside its subgroup.

The system allows applications to dynamically change their ACL, however, this may temporarily break the equivalence relation. For example, it is possible that ,temporarily, member p trusts q, q trust s but p does not trust s. This may allow the creation of a group $\{p, q, s\}$ where not all members trust each other. Therefore, care is required while changing ACLs. It is up to the application to make sure that the new ACLs form a consistent equivalence relation throughout the group.

While it is difficult to change the ACL dynamically, this capability is important when untrusted members need to be removed. For example to remove member q, the group must perform the following steps: (1) switch its ACL to exclude q (2) perform a view change without q (3) rekey so that q will not have the group key. To summarize, although not every method of switching the ACL may work, there are simple ways of achieving this goal.

4.4 ACL's and Perfect Forward Secrecy

The exchange protocol does not provide PFS. For example, in Figure 2, member p sends to q its group key K_P. This allows members of Q to decipher previous messages sent by members of P. It is possible to build a slight variation on the protocol that solves this problem, and guarantees PFS. The idea is to creates a new key every time a merge sequence is started, see Figure 3.

The problem with this variation is that it requires component P to switch its key to K_N. This is a costly operation, especially since the merge is not guaranteed to succeed, hence, we may be blocking P and switching its key without gain.

Therefore, instead of guaranteeing PFS, we guarantee it only with respect to the ACL. Each time the application changes its ACL, it must also ask Ensemble to rekey. Henceforth, the exchange protocol will enforce this ACL, and prevent untrusted members from recovering the new key.

5 Efficient Rekeying

As we have argued above, a secure GCS must provide a means to switch the current group key. This requires efficient, low-latency rekeying algorithms. Such algorithms must provide effective handeling for common scenarios: member join and leave.

We have designed and implemented several efficient rekeying protocols [31, 32]. In this section, we describe a particularly fast protocol that is designed for solving a specific case: a single member

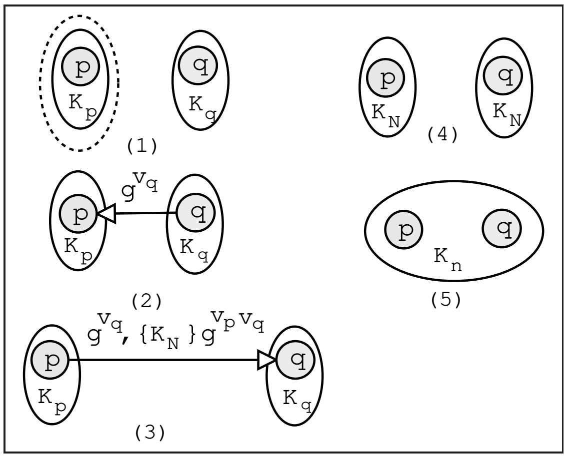


Figure 3: **A variation on the merge sequence that guarantees PFS. (1) Two components with keys K_p and K_q. Each leader sends IamAlive messages. (2) q sends g^{v_q} to p. (3) p chooses a new key K_N, sends g^{v_p} and K_N encrypted with $g^{v_p v_q}$ to q. (4) q decrypts K_N and switches his component key to K_N. Component P switches its key to K_N. (5) Components P and Q now merge, after both have the same key (K_N).**

leave. We also show that its performance in the join case is on par, or even better, than previous results.

A rekeying layer uses services provided by other layers in the stack: Point-to-Point and multicast reliable communication, secure channels provided by SecChan, and the ability to prompt the stack to perform a view change.

The basic communication pattern of a rekeying protocol can be described using a very simple protocol, Basic, that we have studied elsewhere [32].

The Basic algorithm uses a simple method to rekey a group. The leader chooses a new key and disseminates it to the members using secure channels. The members send acknowledgments back to the leader. Once the leader receives acknowledgments from all the members, it performs a view-change, and installs the new key. Algorithm Basic is illustrated in Figure 4. In the figure, and henceforth, a full arrow head denotes a secure message, and an empty arrow head denotes a clear-text message. Members are marked by simple numbers, for example member p_5 is simply denoted by the number five.

A failure can occur during the run of Basic. In such a case, a view-change will occur at all members, and they will all abort the protocol. The application can request a rekey in the new view.

Elsewhere, we have reported fairly fast rekeying times using a distributed version of the Wong-Gouda-Lam algorithm [8, 35, 10]. The WGL algorithm uses the Key Distribution Center (KDC) to disseminate a tree of keys to members of the group. The total number of keys is n, and each member has knowledge of $\log_2 n$ keys. This approach suffers from a single point of failure, if the KDC dies, the whole tree is lost. Our approach splits the KDC functionality evenly among the group members. In this scheme each member knows $\log_2 n$ keys, and no member knows all the keys. Thus, when a fault occurs, only $\log_2 n$ must be replaced. This can be performed quite efficiently [31]. The new protocol was called dWGL, its performance is depicted in Figure 5.

Figure 5 describes a test performed on our set of 20 machines. To create groups larger than 20, several processes were run on the same machine. A large number of member join/leave operations

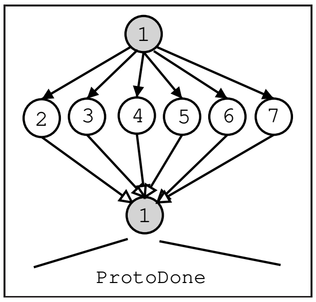


Figure 4: **The communication pattern of algorithm Basic. Member p_1 chooses a new key and sends it to members $p_2, \ldots, p_7$. Members send acknowledgements back to p_1. When p_1 receives all acks, it multicasts a *ProtoDone*.**

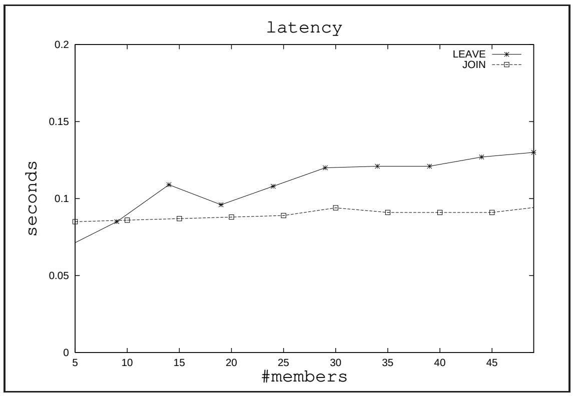


Figure 5: **Performance of the dWGL algorithm.**

was performed, and rekey times were clocked. To simulate real conditions, we flushed the cache once every 30 rekey operations, and discarded "cold-start" results. All of the tests described in this section were conducted in this manner.

What hampered protocol latency was the performance of local integer exponentiations. A member leave caused $\log_2 n$ Diffie-Hellman exchanges, which cost $80ms$ each. The actual communication for a 30 member group was under $20ms$. In what follows, we describe our new Diamond algorithm, which seeks to reduce the number of secure-channels as much as possible.

5.1 The Diamond protocol

The Diamond protocol is based on a graph where the nodes are group members, and the edges are secure channels that connect them. This graph describes the up-to-date status of connectivity between group members. In order to overcome a single failure, the graph should remain connected after a single failure, hence, it must be two-connected. The simplest method for building a two-connected graph for the group is to use a circle. Any node which is removed from the circle will leave it one-connected. Hence, it will be possible for the leader to choose a new key, and pass it to the remaining members without the need to create new secure channels. The problem with the circle structure is that it has diameter $n/2$. This will increase protocol latency. Hence, we require a structure that has logarithmic diameter.

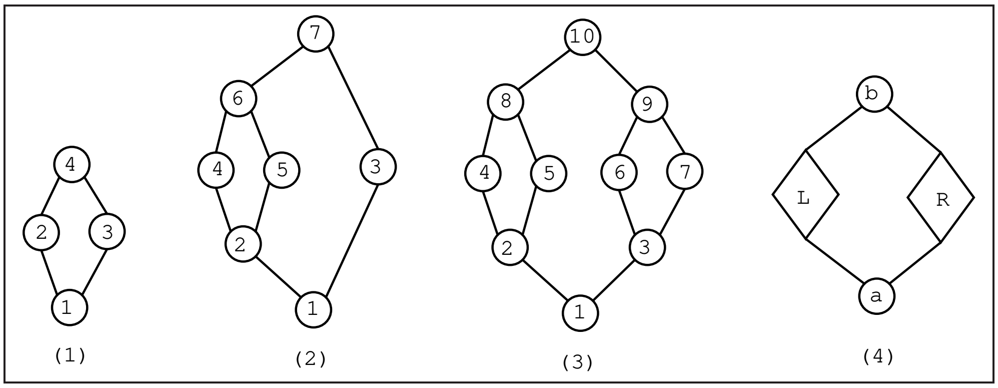


Figure 6: **Two Examples of diamond structures. (1) An example with 4 nodes. (2) An example with 7 nodes. (3) An example with 10 nodes. (4) The general diamond structure. Member b is First, member a is Last, L is the left sub-diamond, and R is the right sub-diamond.**

We use a diamond like graph exapmles of which can be seen in Figure 6. Diamond graph D contains a set of members and edges, it is a diamond-graph if and only if:

The diamond graph has logarithmic diameter. It is defined recursively. Graph D contains a set of members and edges, it is a diamond-graph if and only if:

- D has a First and Last element.
- D contains a left sub-diamond L, and a right sub-diamond R.
- The First element of D is connected to the First elements of L and R.
- The Last element of D is connected to the Last elements of L and R.
- It is possible for L and R to be empty. If both are empty, then there has to be an edge between First and Last.
- There are no other edges in D.

A *balanced* diamond, is one where the difference in height between the left and right sides is no more than 2 (similar to AVL trees). Balanced diamonds have an appealing property: their depth is guaranteed to be logarithmic. As we shall see later, the graph's depth defines the protocol's latency, hence, we would like to minimize it. The group connection graph is dynamic, since members join and leave. In particularly bad scenarios the graph can become unbalanced, with linear depth. To reduce depth, new edges can be added. However, we pay a significat computational price for each edge. Therefore, we rebalance the diamond-graph, making maximal use of existing edges (see below).

First, we describe the general framework of the protocol and provide an example run. Then, we go into the details. The protocol is as follows:

1. A view change occurs
2. Since the new view may be comprised of several merging components, each component has its own diamond-graph. A representative from each component sends its diamond structure to p_1.

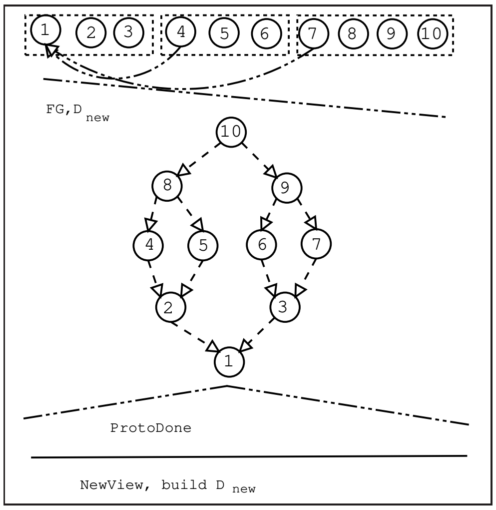


Figure 7: **The communication pattern of the diamond protocol.**

3. Member p_1 merges together the different diamonds. It computes a quick schedule Q for passing the new key. It also computes a new diamond D_{new}. The structures FG and D_{new} are multicasted to the group.

4. Each member receives Q, and D_{new}. The first member of Q chooses a new key K and sends it to its children in. The last member in Q, when it receives K, multicasts a *ProtoDone* message.

5. Members that receive *ProtoDone* switch to the new key, and start sending messages again. They also rebuild the new diamond structure D_{new}.

In Figure 7 an example with ten members is depicted. The group results from the merging of three disjoint components $\{p_1, p_2, p_3\}, \{p_4, p_5, p_6\}$, and $\{p_7, p_8, p_9, p_{10}\}$. The representatives p_1, p_4 and p_7 send their graphs to p_1. The leader, p_1, computes Q, D_{new} and multicasts them to G. The fast-graph FG is then used to rekey the group. Member p_{10} starts the rekey process, it chooses a key and passes it securely to p_8 and p_9. Members p_8 and p_9 pass the key down the graph. When p_1 receives messages from both p_2 and p_3, it multicasts a *ProtoDone* message to the group. All members install the new key chosen by p_{10}, and later engage in a protocol to build D_{new}.

Note that we do not require explicit acknowledgements in our protocol. The last member in the diamond *knows* that all members have received the new key. This saves an additional ack-collection phase requiring another $n-1$ messages. In the above 10 member example, 12 messages are used. In general, a diamond contains no more than $4/3n$ edges (see the Appendix 9). This is close to optimal since any protocol communicating with all members must use at least $n-1$ messages.

The new-diamond structure is created after the new view has been installed. Structurally, it is a balanced two-connected diamond based on the set of existing connections. The members build all connections in the repaired diamond, in preparation of a future rekey request.

The quick schedule is used for a fast-rekey. In Figure 8 several examples are shown. The basic scheme for a ten member group is shown in example (1). Member p_{10} chooses a new key and passes it to its children p_9 and p_8. The new key is passed down the graph until p_1 receives keys from both p_2 and p_3.

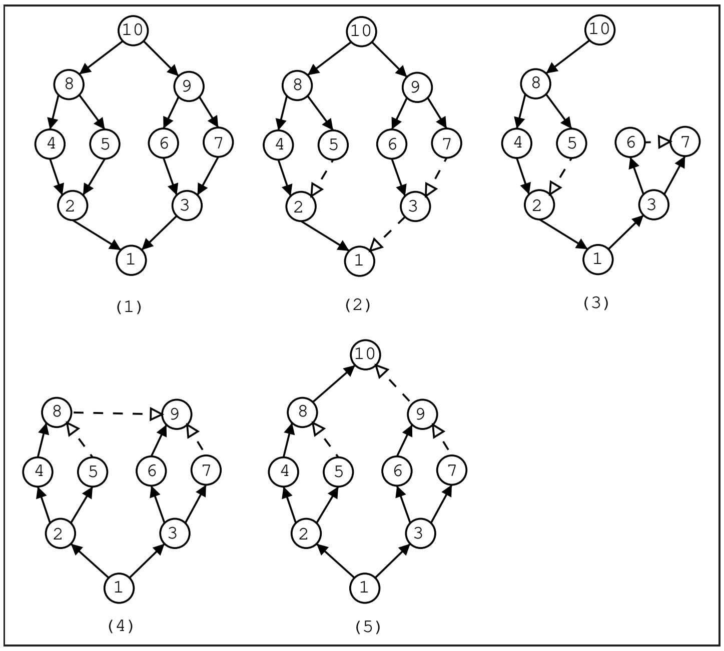


Figure 8: **Examples of quick schedules. (1) The basic protocol. (2) A 10 member group. (3) Member 9 dies. (4) Member 10 dies. (5) Member 10 rejoins. A full arrow head denotes an encrypted message. An empty arrow head denotes a clear-text message (an ack).**

To improve this schedule, we show how some of the secure messages can be turned into clear-text acknowledgements. Note that p_1 receives the key from two different sources: p_2 and p_3. The second key is superfluous, an ack from p_3 to p_1 can be sent instead. In example (2) we show how three secure messages can be saved.

Examples (3) and (4) describe two cases of a single member leave. We must be careful not to create new channels in this case. Note that in case (3) there is no channel between p_6 and p_7. In case (4) there is no channel between p_8 and p_9. In case (5), member p_{10} rejoins the group. The key is passed to p_{10} from p_8, instead of from both p_8 and p_9, this reduces the number of required new channels to one (this is optimal).

The performance we gained using this structure, is two orders of a magnitude better than what we were able to achieve using previous approaches. See figure 9 for performance measurements.

5.2 Balanced Diamonds

The connection-graph after joins and merges occur may not be connected. It is the task of the balancing algorithm to forge a diamond graph out of existing edges, and add as few new edges as possible. Furthermore, it is entrusted with balancing the graph such that its depth is logarithmic.

The procedure followed when creating a balanced diamond, is comprised of two stages: (1) reconnection (2) rebalancing.

To make sure that D_{new} is two-connected we apply a recursive procedure, Fix, to D_{new}.

Procedure Fix:

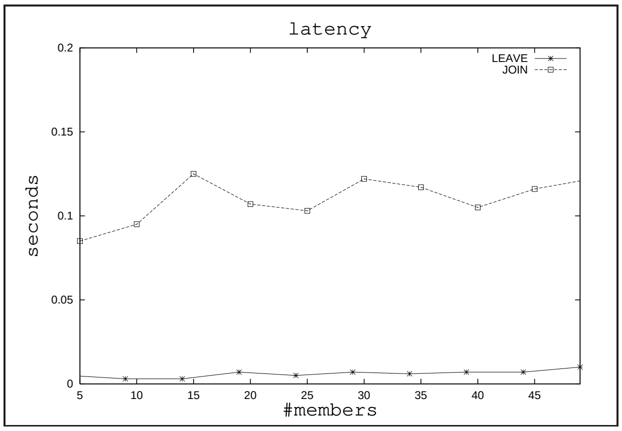


Figure 9: **Performance of the Diamond algorithm.**

- If D has one member, then do nothing.
- If D has two members or more, we must put it into *canonical* form. In canonical form, a diamond has both a First and a Last member. Assume, for example, that D has only a Last member. If Last fails, then D will be cut off from the rest of the graph. To complete First and Last, if they do not exist, we *steal* (see below) members from the larger of the sub-diamonds Left and Right.

The *Fix* algorithm uses a *node stealing* technique. When a node is taken from a diamond structure D, one should cause the least damage to D. It is generally difficult to decide which node makes the best choice. Our heuristic is to take an inner-node, that is, a node that is not a top or bottom node in any diamond. Such nodes are members in only two edges, whereas top and bottom nodes are connected by three.

The *Fix* procedure ensures that each sub-diamond of D_{new} is one-connected. However, this is not enough. It is still possible that D_{new} will have a First, Last, and Left, but no Right. This would make it only one-connected. Therefore, we make sure that D_{new} has non-empty Left and Right. This ensures that the whole graph is two-connected.

Once D_{new} is two-connected, we can rebalance it. In the balance stage we recursively examine D_{new}. For each diamond D we check the height difference between the left and right sides. If this difference is greater than two, then we apply a balance step, see Figure 10. This step involves the creation of four new graph edges. Since this is relatively expensive, we apply balancing only at extreme situations.

5.2.1 Performance

To measure the performance of the balancing algorithm, we measured the number of exponentiations performed on average during the reconstruction phase. Figure 11 depicts the number of exponentiations as a function of the number of members.

For the join case, there are just under two operations performed on average. The nodes in the tree have degrees varying between two and three. Generally, there are more degree-three nodes, than degree-two nodes. In the usual Join case, no rebalancing is required, and the total number of channels that require building is between two and three. This is split into the quick phase, and the reconstruction phase. In the quick phase, a bit more than one channel is created. The optimum

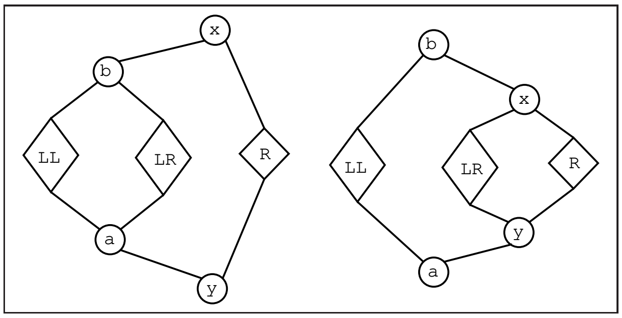


Figure 10: **A rebalancing step. Diamond R is much smaller than L. We then move it down and merge it with LR. The dotted lines denote new edges.**

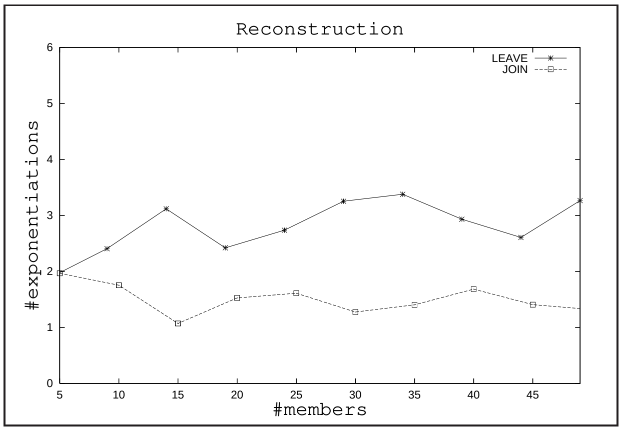


Figure 11: **Performance of the reconstruction algorithm.**

is a single new channel, since that would suffice to pass a fresh new key to the joiner. In the reconstruction phase, we can see that about 1.6 exponentiations are performed.

In the leave case, more complex tree operations are required, since many times, a leave breaks tree connectivity. The removal of a degree three node necessitates moving a different member into its place. In this case, we are interested in the minimization of work. We can see that about 3 exponentiations are performed on average.

6 Related Work

Ensemble is a direct descendent of three systems: Isis [20], Horus [41], and Transis [47]. Early work on group communication security was performed in Horus [29, 39]. Our work extends the Horus security architecture but differs in many ways. We added support for multiple partitions (the secure version of Horus permitted progress only in the primary partition), group rekey upon demand, application-defined security policies, and plugged in off-the-shelf authentication systems.

Many other GCSs have been built around the world. The secure GCSs that we know of are: Antigone [33], Spread [48], Totem [23, 22], and Rampart[38]. Spread splits the GCS functionality into a server and client sides. Protection, in the form of a shared encryption and MAC key, is offered to the client while the server is left unprotected. Access control is not supported. The shared group-key is created using Cliques cryptographic toolkit [28]. Cliques uses contributed shares from each member to create the group-key. Cliques's keys are stronger than our own, however, they require substantially more computation.

Antigone has been used to secure video conferences over the web, using the VIC and VAT tools. In Antigone, the issues of group ACLs, and the trade-off between security and performance when groups are large, and members join and leave often were studied. However, to date, it has not been provided with a fault tolerance architecture.

Rampart [38] is a group communication system built in AT&T which is resistant to Byzantine attacks. Up to a third of the members in a Rampart group may behave in Byzantine manner yet the group would still provide reliable multicast facilities. A system providing similar guarantees has been built in the university of Santa-Barbara in California [22]. Byzantine security is rather costly however, and it is difficult to develop applications resistant to such faults. We chose not to support such a fault model in Ensemble.

The Enclave system [12] allows a set of applications to create a shared security context in which secure communication is possible. All multicast communication is encrypted and signed using a symmetric key. The security context is managed by a member acting as leader. Security is afforded to any application implementing the Enclave API. The Enclave system addresses the security concerns of a larger set of applications than our own, however, fault-tolerance is not addressed. Should the group-leader fail, the shared security context is lost and the group cannot recover.

The Cactus system [24] is a framework allowing the implementation of network services and applications. A Cactus application is typically split into many small layers (or *micro-protocols*), each implementing specific functionality. Cactus has a security architecture [25] that allows switching encryption and MAC algorithms as required. Actual micro-protocols can be switched at runtime as well. This allows the application to adapt to attacks, or changing network conditions at runtime.

Of all systems, ours is closest to Reiter's security architecture for Horus [39]. Horus is a group communication system, sharing much of the characteristics of Ensemble. The system followed the fortress security model, where a single partition was allowed, and members could join and leave the group, protected by access control, and authentication barriers. Group members share a symmetric group key used to encrypt and MAC all inner group messages. Furthermore, the system allocated public keys for groups, that clients could use to perform secure group-RPC. Horus was built at a time when authentication services were not standard, therefore, it included a secure time service, and a replicated byzantine fault-tolerant authentication service. Symmetric encryption was optimized through the generation of one-time-pads in the background.

By comparison, our system uses off-the-shelf authentication services, it does not handle group-RPC, and symmetric encryption is not a bottleneck. In Ensemble we handle the "next tier" of issues: supporting efficient group merge (not just join and leave), allowing multiple partitions (not just primary partition), and efficient group rekeying with PFS (and a weak form of BFS).

Other work in the IP multicast security area includes [1, 14, 15]. These papers describe the management of session keys for (very) large groups, such that the infrastructure required is scalable and efficient. Recent work [42, 8, 35] has dealt with the efficient rekeying of large multicast groups. IP multicast is concerned mainly with one-to-many multicast, where a single application multicasts to many clients whose membership is dynamic and not necessarily known. Ensemble is concerned mainly with many-to-many multicasts where any member may multicast to the group and where membership is known. In secure IP multicast, trusted centralized servers may be used to disseminate group keys; in Ensemble, which possesses a completely distributed architecture, no such single point of failure is allowed.

Our diamond-rekeying protocol touches on the field of fault-tolerant communication graphs, for example [5, 18, 16]. However, work in that field has mostly been oriented towards static networks, where nodes and links can fail, but not recover, and new nodes and links cannot be created on the fly. This is the major difference between our work and other works in the field. An interesting open question is to extend our work to tolerate more than a single failure.

7 Conclusions

We have developed a security architecture for Ensemble, which supports multiple partitions (not just primary partition), group rekeying upon demand, application-specific security policies and off-the-shelf authentication. Our software is freely available as part of the Ensemble project.

8 Acknowledgments

We would like to thank Tal Anker, Yaron Minsky, Michael Ben-Or, and Benny Pinkas for their help. Zhen Xiao wrote the PGP interface, and the initial implementation of the rekeying protocol. Mark Hayden built the Ensemble system, and the basic security infrastructure.

9 Appendix

9.1 Computing the number of edges in a diamond

The maximal number of edges in a diamond is achieved when the diamond is "full". This means that there are no holes, and it contains the maximal number of nodes possible for its depth. If we denote the number of edges as a function of the number of members by $e(n)$ then: $e(n) = 4 + 2 * e(n/2 - 1)$. This is justified by: (1) the first and last members have a total of 4 edges to members in the left and right sub-diamonds (2) the sub-diamonds are full, hence, they have an equal number of edges.

To figure out what the function e(n) looks like, we use our recursive equation:

$$e(n) = 4 + 2 * e(n/2 - 1)$$

and add knowledge about the values of $e(n)$ for small n.

First, we complete e to a continuous derivable function on the field of real numbers. We derive both sides of the equation and get:

$$e'(x) = 2 * e'(x/2 - 1) * (1/2)$$
$$e'(x) = e'(x/2 - 1)$$

This can only occur with a function whose derivative is constant. Hence, e is a linear function.

$$e(n) = an + b.$$

We know that:

$$e(4) = 4$$
$$e(10) = 12$$

We conclude that:

$$e(n) = 4/3n - 4/3$$

This shows that the number of edges in a diamond graph is small. In fact, it is close to the number of edges of a circle.

References

[1] A. Ballardie. Scalable multicast key distribution. Technical Report 1949, IETF, May 1996.

[2] B. C. Neuman and T. Ts'o. Kerberos: An authentication service for computer networks. In *IEEE Communications*, volume 32(9), pages 33–38, September 1994.

[3] B. Whetten, T. Montgomery, and S. Kaplan. A high perfomance totally ordered multicast protocol. In K. P. Birman, F. Mattern, and A. Schipper, editors, *Theory and Practice in Distributed Systems: International Workshop*, pages 33–57. Springer, 1995. Lecture Notes in Computer Science 938.

[4] K. P. Birman. A review of experiences with reliable multicast. *Software, Practice and Experience*, 29(9):741–774, Sept 1999.

[5] C. Dwork, D. Peleg, N. Pippinger, and E. Upfal. Fault tolerance in networks of bounded degree. *SIAM Journal on Computing*, pages 17:975–988, 1988.

[6] C. Malloth and A. Schiper. View Synchronous Communication in Large Scale Networks. In *Proc 2nd Open Workshop of the ESPRIT projet BROADCAST (#6360)*, July 1995.

[7] E. Chiakpo. *RS/6000 SP High Availability Infrastructure*. IBM, November 1996.

[8] C.K. Wong, M. Gouda, and S.S. Lam. Secure group communication using key graphs. In *ACM SIGGCOM*. ACM, September 1998.

[9] D. Dolev, D. Malki, and R. Strong. A framework for partitionable membership service. Technical Report 95-4, Institute of Computer Science, The Hebrew University of Jerusalem, March 1995.

[10] D. M. Wallner, E. J. Harder, and R. C. Agee. Key management for multicast: Issues and architectures. Internet Draft draft-wallner-key-arch-01.txt, IETF, Network Working Group, September 1998.

[11] G. Goft and E. Y. Lotem. The as/400 cluster engine: A case study. In *International Workshop on Group Communication (IWGC'99)*, September 1999.

[12] L. Gong. Enclaves: Enabling secure collaboration over the internet. *IEEE Journal on Selected Areas in Communications*, 15(3):567–575, April 1997.

[13] U. Goverment. Data encryption standard. Technical Report 46, National Bureau of Standards, Federal, 1977.

[14] H. Harney and C. Muckenhirn. Group key management protocol architecture. RFC 2094, IETF, 1997.

[15] H. Harney and C. Muckenhirn. Group key management protocol specification. RFC 2093, IETF, 1997.

[16] F. Harary. The maximum connectivity of a graph. In *The National Academy of Sciences*, pages 48:1142–1146, 1962.

[17] M. Hayden. *The Ensemble System*. Phd thesis, Cornell University, Computer Science, 1998.

[18] J. Bruck, R. Cypher, and C.T. Ho. Fault-tolerant meshes with small degree. In *5th Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 1–10, 1993.

[19] R. J. J. Jr. Isaac. In *Fast Software Encryption, Third International Workshop*, pages 41–49. Springer-Verlag, 1996. 1039 Lecture Notes in Computer Science (D. Gollman, ed.).

[20] K. Birman and R. V. Renesse. *Reliable Distributed Computing with the Isis Toolkit*. IEEE Computer Society Press, 1994.

[21] K. P. Birman, M. Hayden, O. Ozkasap, Z. Xiao, M. Budiu, and Y. Minsky. Bimodal multicast. *ACM Transactions on Computer Systems*, May 1999.

[22] K.P. Kihlstrom, L.E. Moser, and P.M. Melliar-Smith. The securering protocols for securing group communication. In *Hawaii International Conference on System Sciences*, volume 3(31), pages 317–326, January 1998.

[23] L. E. Moser, P. M. Melliar-Smith, D. A. Agarwal, R. K. Budhia, and C. A. Lingley-Papadopoulos. Totem: A fault-tolerant multicast group communication system. In *Communications of the ACM*, April 1996. homepage: http://beta.ece.ucsb.edu/totem.html.

[24] M. A. Hiltunen and R. D. Schlichting. Adaptive distributed and fault-tolerant systems. *International Journal of Computer Systems Science and Engineering*, 11(5):125–133, September 1996.

[25] M. A. Hiltunen, S. Jaiprakash, R. D.Schlichting, and C. A. Ugarte. Fine-grain configurability for secure communication. Technical Report TR00-05, Department of Computer Science, University of Arizona, June 2000.

[26] M. Bellare and P. Rogaway. Entity authentication and key distribution. In *Advances in Cryptology*, August 1993.

[27] M. J. Cox, R. S. Engelschall, S. Henson, B. Laurie, E. A. Young, and T. J. Hudson. Open ssl. http://www.openssl.org.

[28] M. Steiner, G. Tsudik, and M. Waidner. Cliques: A new approach to group key agreement. In *IEEE International Conference on Distributed Computing Systems (ICDCS'98)*, May 1998.

[29] M.K. Reiter, K.P. Birman, and L. Gong. Integrating security in a group oriented distributed system. TR 92-1269, Department of Conmputer Science, University of Cornell, February 1992.

[30] O. Babaoglu, R. Davoli, and A. Montresor. Partitionalbe Group Membership: Specification and Algorithms. TR UBLCS97-1, Department of Conmputer Science, University of Bologna, January 1997.

[31] O. Rodeh, K. P. Birman, and D. Dolev. Optimized group rekey for group communication systems. In *Symposium Network and Distributed System Security*, Febuary 2000.

[32] O. Rodeh, K. P. Birman, and D. Dolev. A study of group rekeying. Technical Report TR2000-1791, Cornell University Computer Science, March 2000.

[33] P. D. McDaniel, A. Prakash, and P. Honeyman. "Antigone: A Flexible Framework for Secure Group Communication". In *Proceedings of the 8th USENIX Security Symposium*, August 1999.

[34] P. Zimmermann. Pretty good privacy. http://www.pgpi.com.

[35] R. Canetti, J. Garay, G. Itkis, D. Micciancio, M. Naor, and B. Pinkas. Multicast security: A taxonomy and efficient constructions, March 1999. To appear in Infocom99.

[36] R. Rivest. The md5 message digest algorithm. RFC 1321, SRI Network Information Center, April 1992.

[37] R. Thayer and K. Kaukonen. A stream cipher encryption algorithm. Internet draft, IETF, July 1997.

[38] M. Reiter. Secure agreement protocols: Reliable and atomic group multicast in rampart. In *ACM Conference on Computer and Communication Security*, pages 68–80, November 1994.

[39] M. Reiter, K.P., Birman, and R. Renesse. A security arcihtecture for fault-tolerant systems. *ACM Transactions on Computer Systems*, 4(12), November 1994.

[40] R.V. Renesse, K. P. Birman, M. Hayden, A. Vaysburd, and D. Karr. Building adaptive systems using ensemble. TR 97-1638, Cornell University, July 1997.

[41] R.V. Renesse, K.P. Birman, and S. Maffeis. Horus, a flexible group communication system. *Communications of the ACM*, April 1996.

[42] S. Mittra. Iolus: A framework for scalable secure multicasting. In *ACM SIGCOMM*, September 1997.

[43] W. Vogels, D. Dumitriu, K. Birman, R. Gamache, R. Short, J. Vert, M. Massa, J. Barrera, and J. Gray. The design and architecture of the microsoft cluster service – a practical approach to high-availability and scalability. In *Symposium on Fault-Tolerant Computing*, number 28, June 1998.

[44] W.Diffie and M.Hellman. New directions in cryptography. *IEEE Transactions on information Theory*, IT-22:644–654, November 1976.

[45] X. Lai, J.L. Massey, and S. Murphy. Markov ciphers and differential cryptanalysis. In *Advances in Cryptology – EUROCRYPT*, 1991.

[46] X. Leroy. The Objective Caml system release 3.00, 2000. http://pauillac.inria.fr/ocaml.

[47] Y. Amir, D. Dolev, S. Kramer, and D. Malki. Transis: A Communication Sub-System for High Availability. In *FTCS conference*, July 1992.

[48] Y. Amir, G. Ateniese, D. Hasse, Y. Kim, C. Nita-Rotaru, T. Schlossnagle, J. Schultz, Jonathan Stanton, and Gene Tsudik. Secure group communication in asynchronous networks with failures: Integration and experiments. In *International Conference on Distributed Computing Systems*, April 2000.

[49] Y. Amir and J. Stanton. The spread wide area group communication system. TR CNDS-98-4, Department of Computer Science, 1998.