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

Ohad Rodeh, Kenneth P. Birman, Mark Hayden, Zhen Xiao, Danny Dolev
{ken,xiao}@cs.cornell.edu, {orodeh,dolev}@cs.huji.ac.il, hayden@pa.dec.com *

December 15, 1998

Abstract

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

*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 [1, 2, 3, 4, 5, 6, 7], some of them commercial products [8]. Example GCS applications include: group-conferencing, distributed simulation, server replication, and more. As the Internet emerged into mainstream use, the role of GCSs in Internet settings, and their security, has emerged as an important topic. A secure GCS must be efficiently protected against malicious behavior or outright attack. This paper describes the security architecture of Ensemble [6], our group communication system, which achieves the desired properties.

Fundamentally, a GCS introduces a *process group* abstraction. A process group coherently binds together many processes into one entity. Within the context of a group, reliable per-source ordered messaging is supported. Processes may dynamically join and leave a group. Groups may dynamically partition into multiple components due to network failures/partitions. The GCS is responsible for simplifying these complex scenarios, overcoming the asynchronous nature of the network and keeping the group abstraction consistent. Processes are provided with membership "views" specifying the list of currently alive and connected group members. A notification is provided whenever network connectivity changes or when processes join/leave the group. The *Virtual Synchrony Model* (VS) [9, 8] specifies the relationship between message delivery and membership notification.

Ensemble was developed at Cornell and the Hebrew Universities. It is written in a dialect of the ML programming language [10] in order to facilitate system verification. The design methodology behind Ensemble stresses modularity and flexibility [11]. Thus, Ensemble is divided into many *layers*, each implementing a simple protocol. Stacking together these layers, much like one uses lego blocks, the user may customize the system to suit its needs.

Underlying our work are two fundamental presumptions: First, we have access to a standard off-the-shelf authentication mechanism; second, the application itself can perform authorization. To secure group messages from tampering and eavesdropping, they are all signed and encrypted. While it is possible to use public key cryptography for this task, we find this approach unacceptably expensive. Since all group members are mutually trusted, we share a symmetric encryption and signature key among them. This key is used to protect all group messages, making the encryption and signature operations very fast (roughly 1000 times faster). Using such a key 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. This is challenging since switching to a new key should be done 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. One may designate a centralized key-server whose responsibility is to disseminate, revoke, and refresh group keys. Only group members in contact with the server have the key and hence can function. 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 protocol should not restrict the Ensemble stack, i.e., all legal layer combinations 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 [12]) 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 proportional to the number of added members. Our solution is much more efficient.

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 [7].
- We provide support for dynamic application-defined authorization polices.

We focus on benign failures and assume that authenticated members will not be corrupted. Byzantine fault tolerant systems have been built by other researchers [13, 14], 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 security architecture is composable with most other Ensemble layers. The user thus has the freedom to combine layers and properties including security.

Virtually Synchronous (VS) group communication has inherently limited scalability. For example, Transis [2] scales to 30 members and Ensemble (with a VS stack) scales to 100 members. Since a group should be resilient to all scenarios of network partitions, each group component should be completely autonomous. Therefore, our architecture does not rely on any centralized servers or services; when some form of leader is required it is elected dynamically.

Throughout this paper we use the term authentication or signature referring both to public key signature and to keyed-MD5[1] signatures.

The remainder of the paper is structured as follows: Section 2 describes the model we use to describe the system and the model of attack. Section 3 describes some Ensemble specifics. The subsequent two sections describe the architecture components situated on the message critical path. These parts are tailored to run efficiently. Section 4 describes Ensemble routers and the secure router we have added, and Section 5 describes the Encrypt layer. The next three sections describe the more subtle part of the architecture that is off the message critical path.

[1]MD5 where the IV (Initial Vector) is fed by a secret key.

Section 6 describes our key agreement protocol — Exchange, Section 7 sketches a proof of its correctness. Section 8 describes the Rekey protocol and its optimization. Section 9 describes system performance, and Section 10 lists related work. Section 11 gives our conclusions and Section 12 contains acknowledgments. The appendix contains some protocol details that were removed from the main body of the protocols for clarity of exposition.

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. We do not consider Byzantine failures.

Processes may get *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 processes in P_i. The subsets P_i are sometimes called *network-components*. We shall consider *dynamic partitions*, where in network-components dynamically merge and split. The partition model is more general than the more common "crash failure" model since crash failures may be modeled as partitions, but not the converse.

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. 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.

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

We assume the existence of an authentication service available to all group members. An authentication service allows members to authenticate each other as well as create private and authentic messages. When member p uses the authentication service to create a secure message m for member q, we shall say that p *sealed* message m. The reverse operation, performed by q to open the message will be called *unseal*.

In what follows p, q, and s denote Ensemble processes and V, V_1, V_2 denote views.

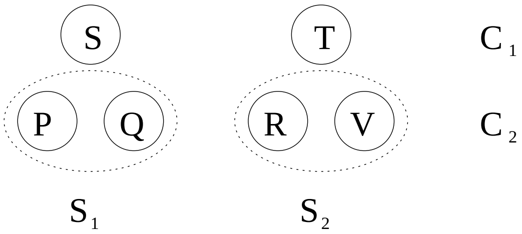
Each member decides on its own trust policy (more on this in the Ensemble section below). If p trusts q then we mark this by $p \overset{t}{\rightarrow} q$. Ensemble forms group components according to the symmetric transitive closure of this relationship, marked by $p \overset{t}{\equiv} q$, and named *st-trust*. St-trust is created as follows:

Symmetry: If $p \stackrel{t}{\rightarrow} q$ and $q \stackrel{t}{\rightarrow} p$ then $p \stackrel{t}{\equiv} q$

Transitivity: If $p \stackrel{t}{\equiv} q$ and $q \stackrel{t}{\equiv} s$ then $p \stackrel{t}{\equiv} s$

St-trust is a distributed relation. When trust policies are stable for sufficiently long at all processes in $\mathcal{U}$ then st-trust becomes an equivalence relation. At such a point, $\mathcal{U}$ is separated into disjoint equivalence classes of processes called *st-domains*. Partitions may prevent members of an st-domain from merging together. For example, assume that $\mathcal{U} = \{p, q, s, t, r, v\}$, the st-domains are $S_1 = \{p, q, s\}, S_2 = \{t, r, v\}$, and there are two network-components $C_1 = \{t, s\}, C_2 = \{p, q, r, v\}$. Then there are four components: $\{p, q\}, \{s\}, \{t\}$ and $\{r, v\}$.

Figure 1: $\mathcal{U} = \{p, q, s, t, r, v\}$, the st-domains are $S_1 = \{s, p, q\}, S_2 = \{t, r, v\}$, the network-components are $C_1 = \{s, t\}, C_2 = \{p, q, r, v\}$. There are four components: $\{s\}, \{p, q\}\{t\}$ and $\{r, v\}$.



If $p \stackrel{t}{\equiv} q$ then Ensemble allows p and q to be in the same component. A member may dynamically change its trust policy and request Ensemble to reform its component accordingly. The system will exclude untrusted members and allow trusted members to join.

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 would cause a breach in Ensemble security.

Throughout this paper we assume all members of $\mathcal{U}$ belong to a single Ensemble group. While describing the Exchange and Rekey protocols we assume that all machines able to authenticate themselves are trusted. Later we refine this model with an application trust policy. Thus, the trust relationship may be dynamic and may include multiple st-domains.

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 [6] 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.

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, that includes the new group key, and all members will 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. They are usually application-generated messages, though some messages may 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 an Ensemble group. A gossip message is multicast to $\mathcal{U}$, to anyone who can hear. Normally, communication is not possible between group components due to network partitions. Gossip messages are used to merge components together and they arrive at their destination when network partitions and link failures are repaired. Receipt of a gossip message when partitioned triggers the *merge* sequence by which separate components are fused together. Protocols that use gossip messages typically 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. Appl_intf 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 through the group. Through it, all members obtain the same symmetric key for encryption and signature. The Rekey layer performs group rekeying upon demand. Both layers manage the group-key that is part of 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.

[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.

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 and also the Exchange, Rekey, and Encrypt layers.

Regular	Secure
Top	
	Exchange
	Rekey
Gmp	
Appl_intf	
	Encrypt
RFifo	
Bottom	
Routers	

3.1 Policies

The user may specify a security policy for an application. The policy specifies for each address[3] whether that address is trusted or not[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 policy allows an application to specify the members that it trusts and exclude untrusted members from its component.

Members should use trust policies that are symmetric and transitive. Otherwise, member p, trusting member q, will be in a component containing members that q trusts but p does not. When a member changes its security policy, it request 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 [15] for signature, and RC4 [16] for encryption. Because these three functionalities are carried out independently any combination of supported authentication, signature, and encryption systems can be used. Other systems, such as Kerberos [17], IDEA [18], and DES [16], have been interfaced with Ensemble at various stages.

3.3 Random number generation

Cryptographically secure random numbers are a vital resource for 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

[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 sections 6,7, how the authenticity of members' addresses is ensured.

to our system.

4 The authentication router

We first describe the simplest part of the security architecture: the authentication 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 $\{q_1 \ldots q_k\}$. 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 signing router which is used when the application requests a secure protocol stack. A signed router adds a keyed-MD5 signature to each sent message and verifies the signature of each incoming message before handing it to the protocol stack.

Ensemble signs all outgoing messages 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 signing 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. Correctly signed: Pass up the stack. Message m was sent by a group member in A.
2. Incorrectly signed: 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. Correctly signed: 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. Incorrectly signed: 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 sealed by the authentication service. The Exchange layer will attempt to unseal 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.

The signing router uses the HMAC [19] standard to compute message signatures. A cryptographically secure one-way hash function, MD5, is used 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 signature. If there is a match — the message has been verified.

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

5 The Encrypt Layer

Ensemble optionally supports user message privacy. The Encrypt layer encrypts/decrypts all user messages with the group key. User messages are reliably delivered in fifo (sender) order allowing use of chained encryption[5]. Ensemble messages are signed, but not encrypted. Such messages do not contain any secret user information and their encryption would only degrade performance. Currently we use the group key for both authentication and encryption. Since MD5 keys are 16 bytes long, we use only the first 5 bytes for the RC4 key. To improve performance, upon a view change we create all security-related data structures and henceforth use them while the view remains current.

Using the group key for both signature and encryption makes the Encrypt layer as strong as the weaker cryptographic system. In the default configuration, the group-key would thus be as strong as the RC4 key. To prevent this from emerging as a weakness of our architecture, we shall switch the group key as frequently as needed to prevent the weaker encryption key from being cracked.

6 Exchange layer

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 to discover partitioned group components. It is active at each group component leader. Each leader gossips an *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. The layer functions via the creation and recognition of two types of gossip message headers. These are, for process p whose principal name[6] is R_p, and whose view key is key_p:

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

Id: Contains R_p and a nonce[7]. This header is cheap to create.

Ticket: Contains data to be sent securely to some process q. This header is created by sealing the data for q. The header is expensive to generate, since its creation involves the authentication service, and it is usually long (currently about 1/2KBytes).

The following event handlers are applied to gossip messages by process q:

- Onto each gossip message, add an Id header.
- Upon receiving an Id$(R_p, nonce_p)$, if it is *insecure*, p is trusted, and $R_q < R_p$[8], then create a Ticket to p and gossip it. The ticket data contains key_q and $nonce_p$ to prove message freshness.
- Upon receiving a Ticket from p intended for q, where p is trusted, authenticate it and check the freshness of the nonce. Decrypt it to get key_p. If $key_p == key_q$, then ignore it (we have the same key), otherwise $new\_key := key_p$. 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.

When a group component leader q receives a gossip message from a remote component leader p, it checks whether it has a lower id[9]. If so, q securely sends to p its key key_q. The remote leader p authenticates q, decrypts key_q and switches its component key to key_q . From now on, all gossip messages (correctly signed by key_q) will be accepted and the components will merge using the membership mechanism.

When a process fails within a component, the (new) leader initiates a view change. The group key is not switched: since we assume honesty of failed members, they will not use knowledge of the current key in a malicious manner.

6.1 Example execution

Figures 2,3,4, and 5 show an example execution of the Exchange protocol. Initially, two components, A and B, are executing. They begin by using different keys, key_A and key_B, so initially only gossip messages marked *insecure* are delivered to the coordinators. No other communication occurs between the components. The authentication sequence only involves the coordinators, so in the following steps we refer to the coordinators by the name of the component they are in.

1. Both coordinators, A and B, regularly broadcast gossip messages announcing their presence in the system. The gossip messages contain both Id headers from the Exchange protocol and $Heal$ headers from the Heal protocol (the Heal protocol heals group partitions). Only the Exchange layer will examine gossip messages marked as *insecure*.
2. Coordinator A receives an $Id(B, nonce_B)$ gossip message from B marked as *insecure*. A sends a $Ticket(key_A, nonce_B)$ gossip message to B.

[7]A one time random string used to prove message freshness.
[8]Any type of comparison function may be used here.
[9]Any comparison function can be used here.

Figure 2: Leaders A and B send out gossip messages.

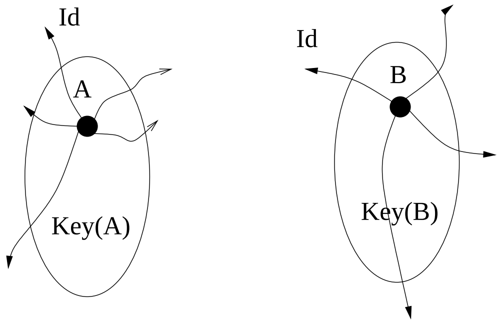
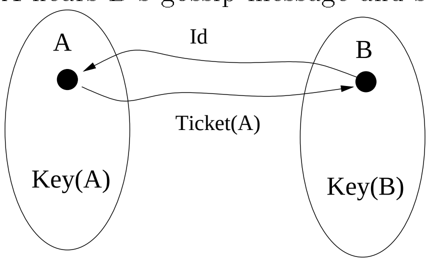


Figure 3: Leader A hears B's gossip message and sends its key to B.



3. B receives a $Ticket(key_A, nonce_B)$ gossip message from A, authenticates and verifies the freshness of the nonce. If the check is passed then B also gets key_A. B then prompts its group to proceed through an empty view change[10]. When installing the new view, B installs key_A in the group. Now both components are using the same key.

4. After B has installed key_A, when A and B broadcast gossip messages, they are accepted by the receiving coordinator and not marked as *insecure*. Now the membership layers will examine the gossip messages, and this results in the two components merging into a single component using key_A.

[10]We say that the view change is empty because there is no group membership changes. Only the key is switched.

Figure 4: B's component switches to $Key(A)$.

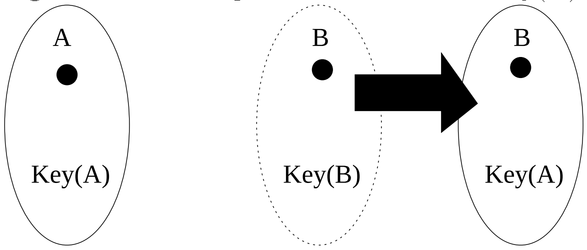
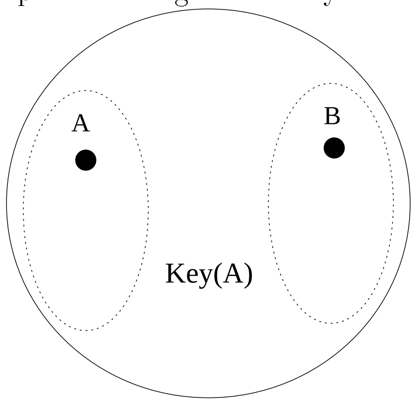


Figure 5: The components merge since they both use the same key.



We use nonces in this protocol so as not to rely on local clocks being in synchrony. In order to use local time as a nonce, one needs to use a secure time service which is not currently a standard Internet service.

7 Validity

In this section we discuss why the security of the Exchange protocol. Layers in the stack, not belonging to the security protocols, do not handle the group key. This is done in order to separate the security functionality from the rest of the stack. Hence, in order to verify that Ensemble is secure we only need to examine its security protocols. Here, we confine ourselves to discussing the Exchange layer, and the manner in which Exchange achieves secure key agreement.

The protocol has two properties:

Safety: Only st-trusted members learn the group key.

Progress: Assuming the network remains stable for a sufficiently long period then all members will eventually agree on the same group key.

We begin with a short discussion of the properties of the st-trust relation.

7.1 St-trust

St-trust is a distributed relation. When trust policies are stable for sufficiently long st-trust becomes an equivalence relation consistent throughout $\mathcal{U}$. It separates $\mathcal{U}$ into disjoint st-domains. For the purpose of this section we assume no partitions. In such conditions all disjoint components merge and form group components according to the trust relationship.

Components are set-wise contained in st-domains. For example, if a set of members S all trust each other then S is an st-domain. Clearly, any component containing a single member of S will include the rest. Hence, in this case we have a component equal to an st-domain.

It is possible for a component to be a proper subset of its st-domain. Assume $\mathcal{U} = \{p, q, s\}$, $p \overset{t}{\leftrightarrow} q$, and $q \overset{t}{\leftrightarrow} s$. Hence, $p \overset{t}{\equiv} q \overset{t}{\equiv} s$, but p and s do not trust each other. The components are $C_1 = \{p\}$

and $C_2 = \{s, q\}$ where the leaders are p and s. C_1 and C_2 will not merge since p and s do not trust each other.

7.2 Safety

Define a *safe* key to be one known only to members of a single st-domain. A key may serve as a group key through several views.

The Exchange protocol may be invoked at any time; it has several stages, all of which may be interrupted by failure. More specifically, the protocol is designed so that if a failure occurs, the protocol can be restarted without risk of a possible security compromise.

We classify failure into two types: malicious and benign.

Benign failures: These are simple message loss scenarios of which there are three cases:

- An IamAlive($R_p, nonce_p$) message may not reach its destination.
- A Ticket($R_q, nonce_p$) may not reach its destination, or reach an untrusted member.
- The view installation phase at p may fail.

We must show that none of these occurrences breaches safety.

- If an IamAlive($R_p, nonce_p$) message does not reach other component leaders then p's component will not merge. While this may (temporarily) prevent components from merging together, it does not breach safety.
- A Ticket($key_q, nonce_p$) may not reach its destination, or reach an untrusted member. If the ticket gets lost then no information is revealed. The ticket can be opened by p alone, thus its capture by a dishonest member does not breach safety.
- If the view installation phase at p fails then only some of p's component members learn key_q. All of p's component is st-trusted by q's component and only st-trusted members learn key_q.

Malicious case: The adversary may try to send a corrupt message:

- IamAlive: the adversary will pretend to be some trusted process q, and send IamAlive($q, nonce_q$). Process p receives this and sends a Ticket($key_p, nonce_q$) to q that only q can decrypt. The adversary cannot make use of this ticket since it cannot unseal it.
- Ticket: the adversary may send p a ticket from some trusted process q. However, the adversary cannot forge such Tickets, hence the ticket will be rejected.

Another form of attack is a replay attack. Resending old IamAlive messages will only create new Ticket messages which the attacker cannot decrypt (as above). Replayed Ticket messages are discarded since they contain a stale nonce.

The view-change protocol involves another step. Assume that two group components, A with leader a and key key_A, and B with leader b and key_B are merging together. Assuming that key_A and key_B are safe, then:

- Exchange: a passes key_A to b, key_A remains safe.
- Second step: b encrypts key_A with key_B and multicasts it to B. Since key_B is safe, only trusted members learn key_A, hence key_A remains safe.

If a partition occurs then new members learn nothing, hence the component key remains safe.

7.3 Progress

To show that the protocol makes progress, assume that no network faults occur for a sufficiently long period, and that sent messages are received in timely fashion without loss or corruption. Under such conditions, assume that in st-domain S all members are mutually trusted. If S is split into two components A and B then eventually A's IamAlive messages will reach B, causing B to send A its key and the components to merge. Hence, eventually, all of S's components will merge.

8 The Rekey protocol

Occasionally we need to switch the group key. There may be several reasons for doing this:

Lifetime expiration: Symmetric encryption keys have a bounded lifetime in which they are secure. After such time a dedicated adversary will be able to crack them. Currently, Ensemble uses RC4 [20] as the default encryption mechanism, this is a relatively weak encryption scheme with a 40bit key. Clearly, there are much stronger algorithms such as DES [16], triple DES [16], and IDEA [18] that employ longer keys (56bits, 112bits, and 128bits respectively). A 128bit key would be safe for years, even using top of the line machines to try cracking it. However, using a weak encryption key allows us to export the Ensemble system from the USA while maintaining a reasonable level of security. It takes 64 MIPS-years to break an RC4 key, which we believe is more than what a casual eavesdropper is willing to pay. Furthermore, RC4 is faster than stronger encryption algorithms. This promotes efficient communication.

Dynamic Trust Policy: Members may dynamically change their trust policy. Thus, old members may not be authorized to listen to current group conversations. This requires the ability to switch the group key, preventing old members from eavesdropping.

OS vulnerability: We work with an existing OS that is not perfect and may be penetrated by a persistent and knowledgeable intruder. The intruder may penetrate old group members and discover the group key. In order not to rely on the assumption that old members will erase their key, we need to switch the key after old members leave the group.

The Rekey protocol provides a way to securely and synchronously switch the communication key used by a group. Unlike the Exchange protocol where we use the old key to disseminate the new key, here the new key is unrelated to the old key and we do not rely on the old key for its dissemination. Thus, possession of the old group key does not allow discovery of the new key or eavesdropping on current group conversations. We authenticate members using their public keys, which we assume are never broken.

The Rekey protocol works as follows:

1. The leader chooses a new random key, unrelated to the old group key.
2. The leader seals the new key for each group member separately, and sends the sealed messages point-to-point to the members.
3. A group member, upon receipt of the new key, sends an acknowledgment to the leader.
4. When the leader receives acknowledgments from all group members it starts a view change. The new key will be installed in the router in the upcoming view.

If some members do not acknowledge the receipt of the new key, they may have crashed or became partitioned by a network failure. A new view excluding the faulty members takes place, and the old key stays unchanged. The user is notified that rekeying has failed and may ask to rekey again. The second invocation is likely to succeed since the faulty members have been removed.

8.1 Authorization polices

When a Rekey is invoked, the leader checks that all current members are trusted. If not, it removes the untrusted members from the view, installing a new (smaller) view. The Exchange protocol will prevent untrusted members from rejoining the group. Rekey requires the correct participation of untrusted members. It cannot exclude a Byzantine member.

The application may dynamically change its security policy. This entails the revocation of the old group key to prevent old untrusted members from eavesdropping and altering current group conversation. Thus, the Rekey protocol needs to be invoked by the application whenever a policy change is performed.

8.2 Optimizing the Protocol

The protocol as described above is fairly slow because seal/unseal operations are very costly in terms of CPU time and memory. For example, a single operation using PGP on a Pentium2 300Mhz takes about 0.25 seconds. Consider the latency of a rekey, operation taking into account only seal/unseal operations, in a group of 64 members. The leader performs 63 seal operations while each member performs another unseal operation prior to acknowledging the key. Thus the latency is: $(63 + 1) * 0.25sec = 16sec$

Our protocol aims to be efficient and scalable. Hence we added the following optimizations:

- We spawn a process to perform the seal/unseal operation in the background. This removes the expensive authentication service calls from the critical path so that the protocol stack can keep running as usual. A similar optimization was used in Horus [21].
- To increase the scalability of the protocol we use a tree structure to disseminate the new key. The leader sends the new key to its children, who in turn pass it down the tree (Figures 6,7). Each member sends an acknowledgment to its ancestor after it has collected acknowledgments from all its children (Figure 8). When the leader receives acknowledgments from all children it multicasts a *RekeyDone* messages to the group and starts a new view (Figure 9).

- Using the tree structure, the latency of a rekey operation can be improved substantially. Suppose a binary tree is used. The cost for each level of the tree, except the first, is: leader performs two seal operations, children perform an unseal operation each. This amounts to 3 operations, and the latency now becomes: $(log_2(64) - 1) * 3 * 0.25sec = 3.75sec$.

The latency using a tree-structure as analyzed above is still high. To improve it, we introduced *secure channels*. A secure channel between members p and q is created as follows:

1. p generates a random symmetric key k_{pq} and seals it for q.
2. p sends the sealed key to q.
3. q acknowledges the receipt of k_{pq}.

Henceforth any message sent on the secure channel is encrypted and signed using k_{pq}. A secure channel allows sending private information between two peers. In contrast, a group key allows multicasting private information to the whole group.

We added a Secchan layer to Ensemble that handles a cache of secure channels. Whenever private information needs to be passed from p to q, a secure channel between them is created if one does not already exist. The operation to create a secure channel is expensive: it takes two seal/unseal operations which cost approximately 0.5 seconds on our test platform. On the other hand, the next private message between p and q will be encrypted with symmetric key k_{pq}, a much quicker operation lasting a couple of microseconds.

Using this cache, a typical group rekey operation will run much quicker. Assuming a static view V, the first Rekey invocation will run several seconds since new channels must be set up [11]. The next invocation will run much faster since secure channels have already been set up. If we examine the 64 member case, and focus on the actual rekeying performed (without the following view installation), then the latency is:

- Ensemble achieves latency of around 1.2ms between members of a LAN. The *average* latency of a reliable multicast is the same.
- Sending the information down/up the tree costs: $1.2ms*2*(\text{tree depth}) = 2.4ms*(log_2(64+1)-1) = 2.4*6 = 14.4ms$.
- To this we must add he latency of a reliable multicast — $15.6ms$

All in all, the latency is less then 20ms, orders of a magnitude less then the latency of the initial implementation.

Channels are refreshed periodically to avoid exposure to cracking. Whenever a rekey operation is invoked we discard any channels that violate the authorization policy.

[11] Note that we do not need set up secure channels between each pair of members in the group. Only those pairs corresponding to the edges in the dissemination tree need be set up.

Figure 6: Leader sends the new group key, k, down the tree. $E_p(k), E_q(k)$ are sealed electronic envelopes containing k for members p and q, respectively.

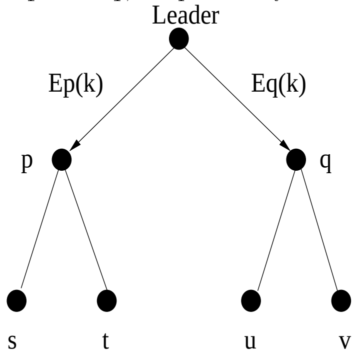
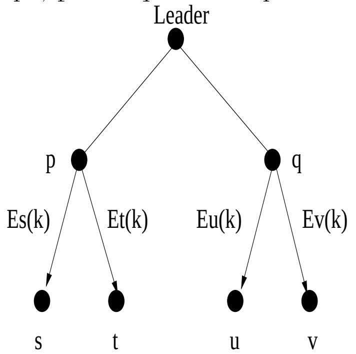


Figure 7: Upon receipt, p and q in turn pass k down their subtrees.



9 Performance

The division of work between the Exchange/Rekey layers and the Encrypt/Routers is computationally efficient. During normal run time we use the symmetric key which is fast and uses little memory. On the relatively rare occasion of a merge or a requested rekey we use the authentication service. These tickets require more computation and memory, one typically uses 1024bit RSA keys or waits for RPC style calls to an authentication server.

With one exception, measurements were taken on PentiumPro 200Mhz machines running the MOSIX operating system [22] connected with 2.5Gbit/sec Myrinet. Current OS and communication stack do not achieve maximal hardware performance.

In Figure 10 we depict the latency of an Ensemble stack. The numbers are given for a send/recv operation: a message arrives at the stack and then is handed to the application which sends an immediate response. The latency is measured from message arrival to message departure. The X-axis measures message size in bytes and the Y-axis time in seconds. As we can see, latency is constant for all message sizes with a regular stack. This is because the stack does not process message content at all. Basic latency increases for Authentication and Privacy stacks since they have not been as aggressively optimized as the regular stack. They also initialize encryption and

Figure 8: Acknowledgments climb up the tree

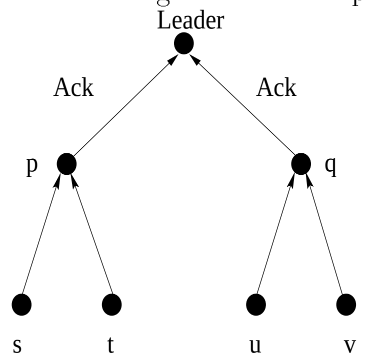
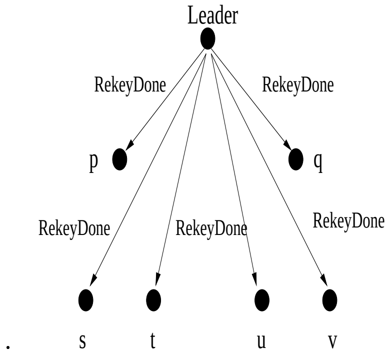


Figure 9: Once the leader receives acknowledgments from all its children, it multicasts a *Rekey-Done* message to the group.



authentication contexts and allocate and add 16bytes of signature space to each message. For such stacks latency also increases as a function of message size since MD5 hashing and RC4 encryption pass over message content. The theoretic processing times for an x-byte message are:

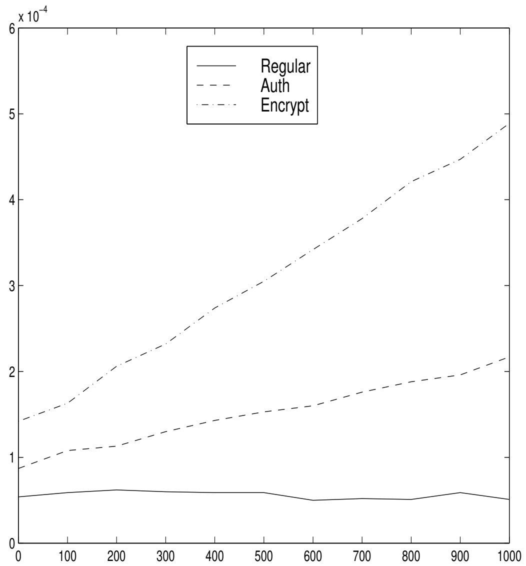
- Authentication: $2 * x/21.45 \sim 0.093x$
- Privacy: $2 * (x/21.45 + x/7.52) \sim 0.358x$

Disregarding the initial costs of encryption and hashing, these linear lines asymptotically approach the measured latencies.

Table 2: MD5 and RC4 performance on different CPUs. Performance is measured by the number of bytes processed in a microsecond.

	P120	PP200	P2/300
MD5	9.92	21.45	32.5
RC4	2.03	7.52	12.45

Figure 10: Latency of a send/recv operation using a regular stack, authenticated stack and a private authenticated stack. The X-axis show message size in bytes, and the Y-axis shows latency in seconds (the multiplier is 10^{-4}).



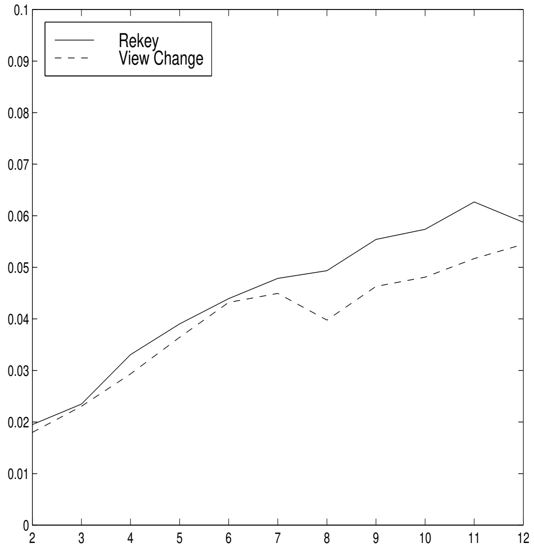
We also tested achievable throughput with Ensemble, as shown in Table 3. We ran an Ensemble application on two machines, one is chosen as leader and it sends as many 1000byte messages to the other member as possible. The maximal achievable throughput using a regular stack is $3330 Kbyte/sec$. As we add authentication, throughput drops to $2850 Kbyte/sec$. When we add encryption throughput drops to $1600 Kbyte/sec$. The bottleneck is the CPU. The third column in the table shows the amount of CPU time per second used for encryption and verification.

Table 3: Ensemble throughput with different stacks. As authentication and encryption are added to the stack, performance drops due to the increasingly heavy CPU load.

Stack	Kbyte/sec	CPU time/sec
Regular	3330	0.000
Auth	2850	0.132
Auth+Privacy	1600	0.286

Next, we measured the latency of the rekey operation. We used 8 PentiumPro 200Mhz and 4 Pentium 120Mhz machines. The Pentiums are inter-connected using a 10Mbit/sec shared hub and through a proxy to the PentiumPro's. We created a group of static size and performed 300 rekey operations in succession. Time was measured from the initiation of a rekey to the installation of the new view. In the same manner, We measured the time required for a view change. Since a rekey includes a view change, the added cost of rekeying is the difference between the two measurements. Figure 11 depicts times for groups of size 2 to 12 where each process is on a different machine. The difference between the two lines grows logarithmically as group size increases. Rekeying using the binary tree we employ for dissemination costs 2 * depth * (point-to-point latency) + (multicast latency). In our setting, the latency is $1.2ms$ and for a group of 12 members this should be $2 * 3 * 1.2ms + 1.2ms = 8.4ms$.

Figure 11: Latency of a rekey operation. The X-axis shows the number of members, and the Y-axis shows latency measured in seconds.



10 Related Work

Ensemble is descended from an earlier system named Horus, itself descended from the Isis system [23]. Early work on group communication security was performed in Horus [24, 21]. Our work extends the Horus security architecture but differs in many ways. We added support for multiple partitions (Horus permitted progress only in the primary partition), group rekey upon demand, application defined security policies, and plugged in off-the-shelf authentication systems.

A group communication system designed for Computer Supported Collaborative Work (CSCW) applications has been built in the university of London [25]. In the context of CSCW, objects and files are typically shared between applications. As such, different applications are allowed to perform different operations on shared objects. To enforce these restrictions the most trusted member of the group is chosen as leader. Any message a member wishes to multicast is forwarded to the leader. The leader filters all such messages: discards the malicious ones, enforces the shared objects security policy, and multicasts all legal messages. This work however is still in a preliminary stage, and at the time of this writing it does not provide for leader failure. Furthermore, the project is oriented only towards CSCW applications.

Rampart [13] 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 [14]. 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.

Other works in the IP multicast security area includes [26, 27, 28]. These papers describe the management of session keys for (very) large groups, such that the infrastructure required is scalable and efficient. Recent work [29, 12, 30] 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 required.

11 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. We believe that ours is the first freely available secure group communication system, and the highest performance secure system available at the time of this writing.

12 Acknowledgments

We would like to thank Tal Anker for improvements to the optimized Rekey protocol, Yaron Minsky for helping develop the Exchange protocol and for insightful comments, and Idit Keidar for helpful reviews.

References

[1] 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.

[2] Y. Amir, D. Dolev, S. Kramer, and D. Malki, "Transis: A Communication Sub-System for High Availability," in *FTCS conference*, July 1992, number 22.

[3] Y. Amir, L.E. Moser, P.M. Melliar-Smith, D.A. Agarwal, and P. Ciarfella, "Fast Message Ordering and Membership using a Logical Token-Passing Ring," in *13th International Conference on Distributed Computing Systems (ICDCS)*, May 1993, pp. 551–560.

[4] 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.

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

[6] 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.

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

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

[9] K.P. Birman and T.A. Joseph, "Exploiting virtual synchrony in distributed systems," TR 87-811, Department of Conmputer Science, University of Cornell, 1987.

[10] X. Leroy, "The Objective Caml system release 1.07," 1997, http://pauillac.inria.fr/ocaml.

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

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

[13] M.K. Reiter, "Secure agreement protocols: Reliable and atomic group multicast in rampart," in *ACM Conference on Computer and Communication Security*, November 1994, number 2, pp. 68–80.

[14] 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*, january 1998, vol. 3(31), pp. 317–326.

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

[16] US Goverment, "Data encryption standard," Tech. Rep. 46, National Bureau of Standards, Federal, 1977.

[17] B. Clifford Neuman and Theodore Ts'o, "Kerberos: An authentication service for computer networks," in *IEEE Communications*, September 1994, vol. 32(9), pp. 33–38.

[18] X. Lai, J.L. Massey, and S. Murphy, "Markov ciphers and differential cryptanalysis," in *Advances in Cryptology - EUROCRYPT*, 1991.

[19] H. Krawczyk, M. Bellare, and R. Canetti, "Hmac: Keyed-hashing for message authentication," RFC 2104, IETF, Febuary 1997.

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

[21] M.K. Reiter, K.P., Birman, and R.V. Renesse, "A security arcihtecture for fault-tolerant systems," *ACM Transactions on Computer Systems*, vol. 4, no. 12, November 1994.

[22] A. Barak and O. La'adan, "The mosix multicomputer operating system for high performance cluster computing," *Journal of Future Generation Computer Systems*, vol. 13, no. 4-5, pp. 361–372, March 1998.

[23] K. Birman , R. Cooper, T.A. Joseph, K. Marzullo, M. Makpangou, K. Kane, F. Schmuck, and M. Wood, *The ISIS System Manual*, Dept of Computer Science, Cornell University, Sep 90.

[24] 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.

[25] A. Rowley and J. Dollimore, "Secure group communication for groupware applications," in *European Research Seminar on Advances in Distributed Systems*, march 1997.

[26] A. Ballardie, "Scalable multicast key distribution," Tech. Rep. 1949, IETF, May 1996.

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

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

[29] S. Mittra, "Iolus: A framework for scalable secure multicasting," in *ACM SIGCOMM*, September 1997.

[30] 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.

13 Appendix

13.1 Running Rekey and Exchange concurrently

The Rekey and Exchange protocols may be invoked concurrently if a merge occurs after the user requested to rekey the group. The leader on the one hand runs the Rekey protocol and disseminates a new random key k_{rnd}, on the other hand it (w.l.o.g.) receives a new key k_{new} from a remote component. Which key should the leader install?

We solve the problem by observing which protocol terminates first. If the Rekey dissemination finishes first, then k_{rnd} will be the new key. If Exchange terminates first then k_{new} is installed and k_{rnd} is discarded.