

Secure Group Communication

Dissertation submitted for the degree “Doctor of Philosophy”

Ohad Rodeh

Submitted to the Senate of the Hebrew University in Jerusalem (2001)

This work was carried out under the supervision of
Prof. Danny Dolev and **Prof. Kenneth P. Birman**.

To Michael, Bruria, and Yifat

Acknowledgments

I thank my advisors, professors Danny Dolev and Ken Birman, for making this research possible, posing challenging problems, and insisting on both practical and theoretical significance of the results. I benefitted from collaboration with many people on various parts of this work. They are: Tal Anker, Tim Clark, Mark Hayden, Dalia Malki, Yaron Minsky, Robbert Van Renesse, and Zhen Xiao. I'd also like to thank the Transis group for my happy years here: David Breitgand, Gregory (Grisha) Chockler, Osnat Mokryn, and Idit Keidar.

Contents

1	Introduction	1
1.1	Introduction	1
1.2	Preliminaries	4
1.2.1	Group Communication	6
1.2.2	Secure Groups	7
1.3	Related Work	8
1.3.1	ACLs and Groups	10
1.3.2	Virtual Private Networks	11
2	Ensemble Security	13
2.1	Introduction	13
2.2	Ensemble	15
2.2.1	Policies	17
2.2.2	Cryptographic infrastructure	18
2.2.3	Random number generation	18
2.2.4	The group key	18
2.2.5	The MAC router	18
2.2.6	The Encrypt layer	20
2.2.7	The SecChan layer	20
2.2.8	Performance assessment	21
2.3	Secure Merge – The Exchange protocol	22

2.3.1	Naive version	23
2.3.2	Handling replay attacks	26
2.3.3	Access Control Lists	26
2.3.4	ACL's and Perfect Forward Secrecy	27
2.4	Efficient Rekeying	28
2.4.1	The Diamond protocol	30
2.4.2	Balanced Diamonds	34
2.5	Conclusions	36
2.6	Computing the number of edges in a diamond	37
3	Using AVL Trees for Efficient Group Rekey	39
3.1	Introduction	39
3.2	The centralized solution (\mathcal{C})	40
3.2.1	Tree balancing	43
3.3	The basic (\mathcal{B}) distributed protocol	47
3.3.1	Virtual Synchrony	51
3.4	Optimized solution (\mathcal{O})	52
3.4.1	Three round solution (\mathcal{O}_3)	57
3.4.2	Costs	58
3.5	Performance	59
3.6	Conclusions	61
4	Virtual Private Networks	63
4.1	Preface	63
4.2	A standard VPN	66
4.3	Model	68
4.4	Our Solution	69
4.5	The driver	72
4.5.1	Key security	72

4.6	The mngr	73
4.6.1	Key Lifetimes	74
4.7	Available Address Service (AAS)	74
4.8	Performance	75
4.9	Scalability	78
4.10	Future Work	78
4.11	Conclusions	80
	Bibliography	80
5	Appendix: K-diamonds	89
5.1	Introduction	89
5.2	Merge	93
5.3	Leave	93

Abstract

Group Communication Systems (GCSs) are a well known form of middleware. A GCS creates process groups in which reliable ordered multicast and point-to-point messaging are supported. Processes may dynamically join and leave a group. Group membership is agreed, as group membership changes over time, notifications are sent to group members consistently. A GCS can be used to provide primitives for distributed computing such as reliable agreed total-ordering of messages, sender-ordered Fifo messaging, state-machine replication and more.

Use of GCSs is now common in industry when clustering technology is required. The SP2, and other IBM clustering systems use the Phoenix system [Chi96], IBM AS/400 clusters use the Clue system [GL99], and Microsoft Wolfpack clusters use Group Communication technology at the core of their system [VDB⁺98]. GCSs are used for other purposes as well. The first industrial strength GCS, Isis [BR94], has been used for air-traffic control, large-scale simulations, in the New-York and Swiss stock exchanges and more. Some of these projects are described in [Bir99]. Adding security to Group Communication Systems will allow us to use this technology where it was not possible before, in unprotected wide area networks.

In this work the *fortress* security model is used. In this model, the “good guys” are protected by a castle from the hordes of barbarians outside. In the context of a GCS this maps to a situation where the (honest) group members need thick walls to isolate them from the adversary. A secure GCS allows the creation of *secure groups*. In a secure group:

1. Only trusted authenticated members are allowed to join.
2. Integrity and confidentiality of group-messages is guaranteed.
3. Perfect Forward Secrecy: Compromising long-term secrets should not reveal group keys.
4. Forward Confidentiality: Past members cannot obtain keys used in the future (FCY).

Furthermore, the system should be modular, security features should be optional, and should not penalize the rest of the system.

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. Public key cryptography is relatively expensive, hence, we use a shared symmetric key. Assuming

all members agree on the same key, all group messages can be MAC-ed¹ and encrypted. Assuming the adversary has no way of retrieving the group-key, the members are protected.

The group-key needs special handling as it must 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, if there is danger that it was leaked to the adversary, or if the set of members has changed.

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.

This thesis is comprised of four parts, in the first we describe our model, and assumptions. In the second, we describe our GCS, Ensemble, and the security architecture that we have built for it. The third part describes an efficient rekeying algorithm. The fourth part describes an application that makes use of our secure GCS: a Dynamic Virtual Private Network (VPN).

A Dynamic VPN extends traditional Virtual Private Network with fault-tolerance and dynamic membership properties, defining a Dynamic Virtual Private Network (DVPN). We require no new hardware and make no special assumptions about line security. An implementation exhibits low overhead, provides guarantees of authenticity and confidentiality to any IP application running over the virtual network. Our system is lightweight, allowing the use of multiple fine-grained VPNs. Instead of using many point-to-point secure connections to bridge insecure communication paths we share a single symmetric encryption key throughout the VPN. This permits tight control of the VPN membership and fast dynamic membership change.

The low cost of a DVPN allows using multiple DVPNs to implement fine grained security. By enforcing policies over communication among DVPNs, our scheme supports multilevel security.

¹MAC 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.

Chapter 1

Introduction

1.1 Introduction

Group Communication Systems (GCSs) are a well known form of middleware. A GCS creates process groups in which reliable ordered multicast and point-to-point messaging are supported. Processes may dynamically join and leave a group. Group membership is agreed, as group membership changes over time, membership notifications are sent to group members consistently. A GCS can be used to provide primitives for distributed computing such as reliable agreed total-ordering of messages, sender-ordered Fifo messaging, state-machine replication and more.

Use of GCSs is now common in industry when clustering technology is required. The SP2, and other IBM clustering systems use the Phoenix system [Chi96], IBM AS/400 clusters use the Clue system [GL99], and Microsoft Wolfpack clusters use Group Communication technology at the core of their system [VDB⁺98]. GCSs are used for other purposes as well. The first industrial strength GCS, Isis [BR94], has been used for air-traffic control, large-scale simulations, in the New-York and Swiss stock exchanges and more. Some of these projects are described in [Bir99]. Adding security to Group Communication Systems will allow us to use this technology where it was not possible before, in unprotected wide area networks.

Modern GCSs implement many *mini-protocols*. Each mini-protocol guarantees a particular property, for example, fifo-ordered messages, reliable multicast, causal-ordering, reliable point-to-point etc. The user configures a *protocol stack* by choosing the set of mini-protocols it requires. Since different protocols incur different costs, the user can “pay (only) for what it uses”. Our GCS *Ensemble* (see chapter 2) is implemented in such a way.

In this work the *fortress* security model is used. In this model, the “good guys” are protected by a castle from the hordes of barbarians outside. In the context of a GCS this maps to a situation where the (honest) group members require thick walls to isolate them from the adversary. A secure GCS allows the creation of *secure groups*. In a secure group:

Authentication & Authorization : Only trusted authenticated members are allowed to join.

Integrity: Integrity and confidentiality of group-messages is guaranteed.

Perfect Forward Secrecy: Compromising long-term secrets should not reveal group keys.

Forward Confidentiality: Past members cannot obtain keys used in the future (FCY). The dual requirement, that current members cannot obtain keys used in the past is termed Backward Confidentiality (BCY).

Furthermore, the system should be modular, security features should be optional, and should not penalize other protocols. In other words, the security protocols should “compose” well with other protocols.

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 using 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, that we have chosen, is to use a shared symmetric key. Assuming all members agree on the same key, all group messages can be MAC-ed¹ and encrypted. Assuming the adversary has no way of retrieving the group-key, the members are protected.

The group-key needs special handling as it must 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.

¹MAC 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.

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 [DMS95]. 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 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 [WGL98]) 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 \times join protocol (which is also $O(n)$). Our solution is much more efficient.

After we expound on securing the GCS abstraction (Chapter 2), and describe efficient rekeying methods (Chapter 3), we describe an application using secure groups (Chapter 4). A common secure abstraction today, is a Virtual Private Network (VPN) [Che, Rad, Cis, Mic]. While providing strong security to applications spread across the globe, today’s VPNs are not very dynamic, new machines are hard to add and remove, and security policies are not easily changed. A Dynamic VPN extends traditional Virtual Private Network with fault-tolerance and dynamic membership properties, defining a Dynamic Virtual Private Network (DVPN). We require no new hardware and make no special assumptions about line security. An implementation exhibits low overhead, provides guarantees of authenticity and confidentiality to any IP application running over the virtual network. The system is lightweight, allowing the use of multiple fine-grained VPNs. Instead of using many point-to-point secure connections to bridge insecure communication paths we share a single symmetric encryption key throughout the VPN. This permits tight control of the VPN membership and fast dynamic membership change.

The low cost of a DVPN allows using multiple DVPNs to implement fine grained security. By enforcing policies over communication between DVPNs, the scheme supports multilevel security.

In appendix 5, we describe an extension to a rekeying protocol described in chapter 2.

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 [RBM96].
- We provide support for dynamic application-defined authorization policies.
- We demonstrate how the notion of key-graphs [WGL98, WHA98] (also known as logical-key-hierarchies) previously used for managing keys in large IP-multicast groups, can be adapted to GCSs.

The standard protocol requires a centralized key-server that has knowledge of the full key-graph. dLKH does not delegate this role to any one process. Rather, members enlist in a collaborative effort to create the group key-graph. The key-graph contains n keys, of which each member learns $\log_2 n$.

The protocol balances the key-graph, a result that is applicable to the centralized protocol. dLKH is optimized so that it is competitive with the centralized solution.

- We demonstrate how to build a fast rekeying protocol that allows the fast removal of an untrusted member within milliseconds. This protocol, named **Diamond**, is based on diamond like graphs that can tolerate a single failure.
- We demonstrate how to use a GCS to implement an interesting form of virtual private network: a DVPN.

1.2 Preliminaries

This work is concerned with machines connected through the Internet. We model this world as a “universe” consisting 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 $\{P_1, \dots, P_k\}$ 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 [DMS95], 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.

We assume that processes 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 that creates 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 will be to protect messages sent between mutually trusting members of \mathcal{U} . Denial of service or traffic analysis attacks are out of the scope of this work. Rather, we restrict ourselves to guaranteeing authenticity and secrecy of message content. We work with existing operating systems and assume their security and correctness. An OS vulnerability or a compromised authentication service would cause a security breach.

We assume the adversary is computationally bounded, so that standard cryptographic algorithms such as DES [US 77], IDEA [LMM91], MD5 [Riv92], and RSA [RSA78] can be assumed secure. In fact, most symmetric key and MAC-ing algorithms in use today have not been proven secure, rather, they have not been found *defective*. It is assumed that the only viable attack is a search of the key space. At the time of writing 56-bit DES keys are assumed insecure, while 128-bit IDEA keys are assumed secure. As time goes by and processor speed increases, key size will have to increase. We do not rely on any single algorithm, but rather built a system such that new cryptographic algorithms can easily be plugged-in and used.

Security is provided on a machine granularity (since most off the shelf operating systems could readily be compromised by a knowledgeable user). Hence, We do not handle issues of *minimal Trusted Computing Bases* [Dep85], secure bootstrapping, etc.

Our work is performed over a communication stack, typically IP. Hence, we are exposed to attacks at the level of the stack itself (“poison pill” attacks). We also assume the correct functioning of the communication infrastructure: routing, name-service (DNS),

etc. In today's Internet this requires secure-DNS and secure routing protocols, which have not been deployed yet.

We assume members behave according to group protocols, and that they do not leak information they are trusted with. We do not handle Byzantine failures, nor covert channels (such as timing channels).

It is possible to share secret information using out-of-band communication, e.g., by cdroms containing cryptographically strong keys. We do not consider such options, the only assumption we make is that a common authentication infrastructure exists, allowing the exchange of secure information.

It is possible to use other means to protect a process group. For example, place all processes behind a firewall, or use a Virtual Private Network to connect them together. These options are not considered.

1.2.1 Group Communication

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 .

A GCS is capable of ordering multicast group messages in various ways: *FIFO* or "sender-ordered", causal order, total order, and more. Of these, FIFO is the most basic, and all our protocols will use FIFO ordered messages.

Ensemble follows the Virtual Synchrony (VS) [BJ87] 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” w.r.t to views. If process q in view V fails then all the members in $V \setminus \{q\}$ observe this event at the “same time”. This means that if the set of multicasts sent in view V is M , then the surviving members deliver an agreed upon subset of the messages in M before delivering the new view $V \setminus \{q\}$.

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 referred to [BHO⁺99, Bir99]. For example, reliable fifo in the virtual synchrony model requires all members to buffer each other’s messages until they are known to be stable (delivered at all destinations). This requirement prohibits scaling to large groups. In this paper, we do not discuss configurations of more than a hundred members.

1.2.2 Secure Groups

We define a secure group as one in which:

1. Only trusted authenticated members are allowed to join.
2. Integrity and confidentiality of group-messages is guaranteed.
3. Perfect Forward Secrecy (PFS): Compromising long-term secrets should not reveal group keys.
4. Forward Confidentiality: Past members cannot obtain keys used in the future (FCY). The dual requirement, that current members cannot obtain keys used in the past is termed Backward Confidentiality (BCY).

To support FCY, the group key must be switched every time members join and leave. This may incur high cost, therefore, we shall attempt to relax FCY without breaking security in both Ensemble, and the DVPN system.

Members describe their trust policies by an *Access Control List* (ACL). In addition to the above 3 requirements, we add an additional requirement, that we would like to allow members to dynamically change their ACLs.

To explain the motivation for the fourth requirement, 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 .

1.3 Related Work

Ensemble is a direct descendent of three systems: Isis [BR94], Horus [RBM96], and Transis [ADKM92]. Many other GCSs have been built around the world. The secure GCSs that we know of are: Horus [RBR94], Antigone [MPH99], Spread [AAH⁺00], Totem [MMA⁺96, KMM98], and Rampart [Rei94].

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 [STW98]. 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.

Rampart [Rei94] 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 [KMM98]. 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 [Gon97] 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 [HS96] 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 [HJSU00] 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 [RBR94]. 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 FCY.

The secure IP multicast community is also interested in group key management, although their focus is on very large scale groups. The Secure Multicast Group (SMuG) has defined three problem areas [HT99, HCB00]: 1) data handling (source authentication, and data encryption), 2) secure and scalable group key management, 3) group policy.

Basic work on group key management was performed in [Bal96, HM97a, HM97b, Mit97]. These papers describe the management of session keys for (very) large groups, such that the infrastructure required is scalable and efficient. Recent work [WHA98, WGL98] has suggested using a *Logical Key Hierarchy* (LKH) for rekeying large groups. In LKH, a logical key hierarchy rooted at a key-server is created. The key-server opens secure channels with all group members and disseminates group sub-keys to group subsets. To remove a member, all keys known to that member must be discarded, and replacement keys are chosen in their place. Storage space at the server is $O(n)$, at the client it is $O(\log n)$, and message size for removing/adding a member is $O(\log n)$. Improvements to LKH were suggested in [BMS99, CGI⁺99], and optimal bounds were derived in [CMN99, PB99]. Work on policy for large scale groups has been described in [MPH99, JMT98, HCM01].

IP multicast is concerned mainly with one-to-many multicast, where a single ap-

plication multicasts to many clients whose membership is dynamic and not necessarily known (so called $M - N$ situations are also discussed). 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. Ensemble leaves it up to the user to specify group policy, while providing efficient tools for enforcement.

The diamond rekeying protocol touches on the field of fault-tolerant communication graphs, for example [DPPU88, BCH97, Har62]. 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. There has been some work on using graphs to check member liveness in the context of a GCS [FMG99], however, this was not in the security context.

1.3.1 ACLs and Groups

When groups are large, membership management becomes an issue. Two scenarios dominate secure process groups: Collaborative work groups, and authorized download of content. In the first case, group communication of the type used here is appropriate. In the second case, completely different methods are required. Here we describe the second case, though we note that its methods can be applied to the first case as well.

In work that handles these issues such as [Mit97, WHA98, WHA99, HCD99a, HCD99b], a group typically holds several role-holders: Group Owner, Group Controller (GC), and Sub Group Controller (SGC). The group owner initiates the creation of the group. The GC handles membership changes — join, leave, and rekeying when this becomes necessary. Sub-Group controllers share the load with the GC, using some form of hierarchy to split responsibility on group members.

In such settings, group policies are also an issue. For example, in [MP00], the GC gathers policies from joining members, and reconciles them into a single group policy. When new members join they can examine the existing policy, and leave if it does not suit their security requirements.

The Antigone system [MPH99] has been built with these issues in mind. It has been used to secure video conferences over the web, using the VIC [MJ96b] and VAT [MJ96a] 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, it has not been provided with a fault tolerance architecture.

The Akenti [JMT98] project aims at addressing issues raised by allowing restricted access to resources which are controlled by multiple parties. Akenti provides a way to express and to enforce an access control policy without requiring a central enforcer and administrative authority. The system’s architecture is intended to provide scalable security services in highly distributed network environments.

1.3.2 Virtual Private Networks

The notion of virtual networks was proposed by Bellovin in [Bel90]. Secure encapsulating drivers were introduced by Ioaniddis and Blaze [IB93]. They use an encapsulating driver in the kernel to create secure point-to-point connections between machines. Our work differs from theirs in many ways: smooth re-keying, the notion of grouping machines into DVPNs, allowing a machine to exist in several DVPNs simultaneously, and the notion of inter-DVPN policies.

Our work exhibits some resemblance to an application level solution proposed by Gong [Gon97]. Gong creates a security context in which applications can communicate securely. All multicast communication is encrypted and signed using a symmetric key. His work, although using a similar notion of grouping, is limited to applications implemented over an API provided by the *Enclaves* software. Furthermore, the security context is managed by a member acting as leader. Should it fail, the shared context is lost and the group cannot recover.

The SURF [GSSC96] firewall is an attempt to implement an “Academic Firewall”. SURF is a VPN substantially more permissive and flexible than commercial VPNs, constructing a security perimeter around groups of machines in a fairly rigorous manner. However, it does not address firewall fault tolerance nor dynamicity concerns.

The *Secure Virtual Enclaves* project [SYJS00] allows different organizations to share local resources and distributed application objects in a controlled manner, respecting each organization’s restrictions. It attempts to extend Internet firewalls and selected hosts to include protection mechanisms against unsafe services and protocols (e.g. FTP, HTTP, NFS). Protection is enforced using a low level mechanism based on *Domain and Type enforcement*. Negotiation is used to agree on the terms of using local objects by peer organizations. Negotiation allows clients and servers to protect themselves and establish mutual and appropriate levels of trust.

Chapter 2

Ensemble Security

2.1 Introduction

This chapter describes a general method for securing group communication systems. To demonstrate our solution, we required an implementation platform. We chose Ensemble due to its modular structure, allowing the addition of properties and capabilities without penalizing the rest of the system. We stress that the principals demonstrated here should be portable to other GCSs as well.

When this work was undertaken, at the beginning of 1998, the only previous attempt at providing an extensive GCS security architecture was carried out by Reiter [RBR94]. Reiter started with an the Horus [RBM96] system developed in Cornell. Horus is the precursor to Ensemble, sharing many of its characteristics. The Horus security architecture 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.

As Ensemble is the successor to Horus, it was natural to continue and extend Reiter's work to Ensemble. Our work shows how security properties are decomposed and introduction into a layered protocol architecture. The major differences between the systems are:

Fortress model: We choose to continue using the fortress model, since the other possi-

ble model — Byzantine security has proven expensive [Rei94, KMM98]. However, we extended the model to allow for dynamic ACLs. This allows a group to exclude untrusted members on the fly.

Partitions: We extend Horus to support multiple partitions.

Access control, authentication: These are more challenging, since multiple partitions are supported. For example, a distributed authentication system must be used, and access-control cannot be provided by a single server.

Symmetric group key: We use a symmetric group key (as Horus), public key cryptography is too expensive.

Public keys for groups — group RPC: This topic was considered open research in 1993, however, in 1998 several Internet drafts contained standardization proposals for a scheme mapping public keys to groups of machines or principals [RL96, Ell99, EFL⁺99]. Hence, we chose not to pursue this topic.

Authentication services: In 1998, authentication services were becoming standardized. For example: Kerberos [NT94], and PGP [Zim00, CDB⁺98]. Hence, we needed to interface with existing systems, rather than build our own.

Symmetric encryption: processor speed advanced to a stage where encryption and MAC overhead was not an issue.

Rekeying: Horus did not perform group rekeying. We implemented several rekeying protocols, studied and improved their performance. Our rekeying protocols guarantee Perfect Forward Secrecy. In particular, this chapter describes the **Diamond** protocol, allowing the removal of a single untrusted member within several milliseconds.

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

To support FCY, the group key must be switched every time members join and leave. This may incur high cost, therefore, we attempt to relax FCY without breaking security. Other researchers have also noted that rekeying is a costly operation, and have attempted to perform periodic rekeying [SKJ00, MK99], or to tie it with the ACL mechanisms [MPH99]. Our approach is three pronged:

- Fast rekeying algorithms are included in the system.
- The system rekeys itself once 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.

The default lifetime of session and group keys is 24 hours, the maximum suggested by the Photuris [KS95] protocol. For stronger security the user may shorten key lifetime. Applications specify the list of trusted members to Ensemble (see more on this in section 2.2.1), in the form of an Access Control List (ACL) which is treated as replicated data within the group. 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. Ensemble allows any trusted member into a group without rekeying. Members may 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 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 [MPH99].

2.2 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 [RBH⁺97, Hay98] 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 among 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 among 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 2.1. The Top and Bottom layers cap the stack from both sides. The Group Membership Protocol (GMP) layer¹ 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 2.4 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.

¹Some “layers”, as discussed here, are actually sets of layers in the implementation. Also, some layer names have been changed for clarity of exposition.

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

Table 2.1: **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.

2.2.1 Policies

The user may specify a security policy for an application. The policy specifies for each address² whether or not that address is trusted³. 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.

As we shall see in Section 2.3, due to efficiency considerations, members should use trust policies that are symmetric and transitive. Other types of policies are currently not supported under Ensemble. If the policy does not adhere to the equivalence relation restriction, Ensemble does not guaranty security. 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.

²An 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,..}.

³We shall see later, in section 2.3 how the authenticity of members' addresses is ensured.

2.2.2 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 [Riv92] for MAC, and RC4 [TK97] for encryption. Because these three functionalities are carried out independently any combination of supported authentication, MAC, and encryption systems can be used.

Insofar as freeware cryptographic libraries are available, we preferred 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 [CEH⁺00] allows using RC4, DES [US 77], IDEA [LMM91], and Diffie-Hellman [DH76]. For authentication, in addition to PGP [Zim00], we have a Kerberos [NT94] interface, though it is out of date.

2.2.3 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 [Jen96].

2.2.4 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.

Splitting 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.

2.2.5 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 2.1.

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. We are using the standard HMAC algorithm [KBC97]. 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*.

2.2.6 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 encryption⁴. 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.

2.2.7 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 hours⁵ 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 1024bit⁶ long integers, where both sides perform an exponentiation.

⁴Modern 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.

⁵This is a settable parameter

⁶At the time of writing this is considered secure.

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

2.2.8 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 2.1(1) shows the latency for a send/rcv 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. For a 900 byte message the latency is 140 microseconds. Note that the base line is a low and constant 24 microseconds. Hence, the basic overhead imposed by the system is very low. Furthermore, the cost difference between the different stacks is almost entirely due to the MAC and encryption algorithms, not to the layering structure.

Figure 2.1(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 to `ping`. We conclude that the latency is mostly due to the network and operating system.

Figure 2.1(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 750Kbyte/sec can be achieved by a single sender 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 numbered, for flow-control, a back-off protocol must be used etc. In this particular experiment, network traffic was problematic and some variance shows in the results.

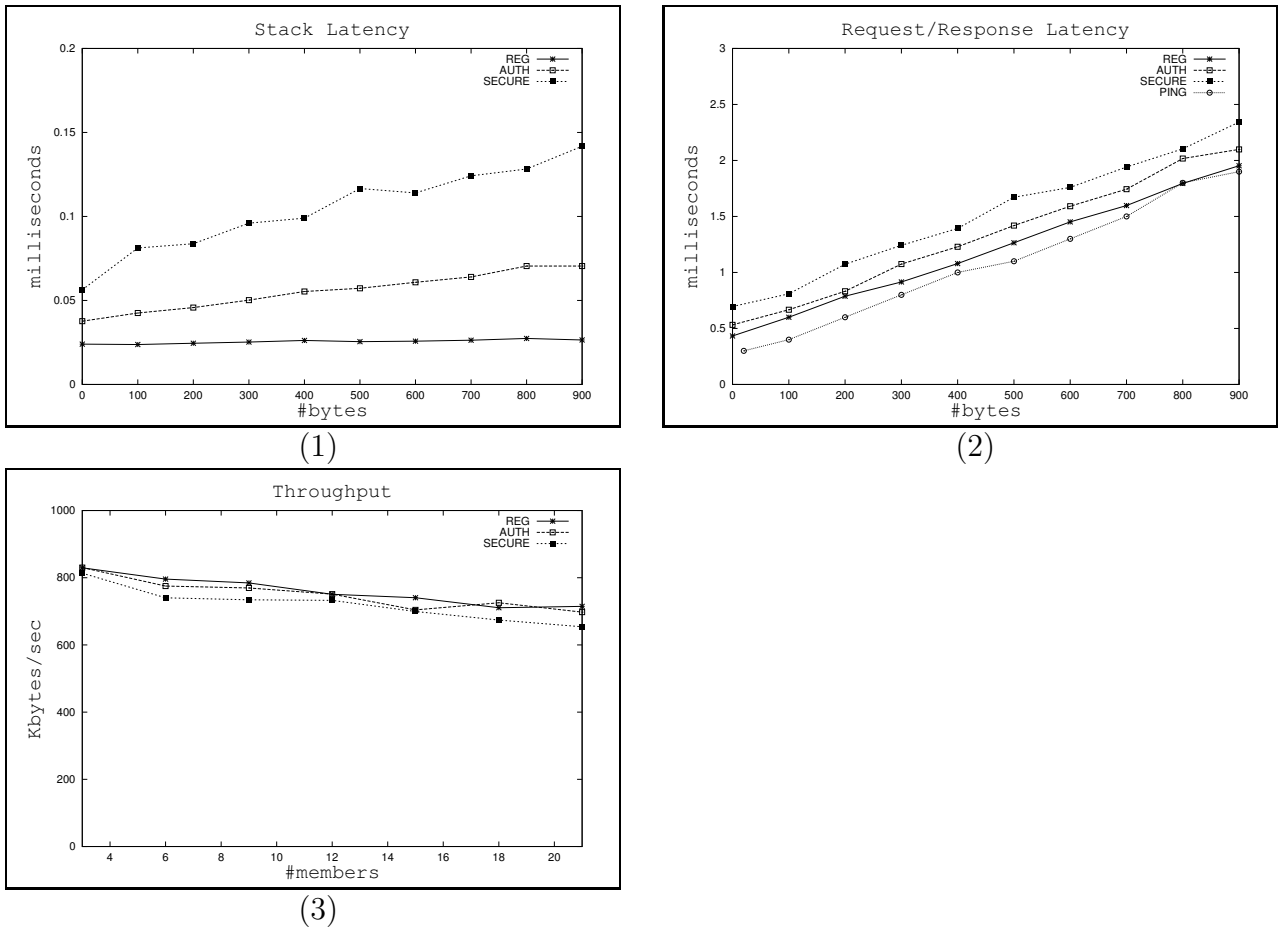


Figure 2.1: A standard stack is denoted REG. An authenticated stack is denoted AUTH. An authenticated encrypted stack is denoted SECURE. (1) send-recv latency in the Ensemble stack. (2) Total Latency for point-to-point send/recv. (3) Throughput.

2.3 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 [BR93]. Our protocol also employs the Diffie-Hellman exchange [DH76]. 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.

2.3.1 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*⁷ is R_p , and whose view key is

⁷This is the name, by which the user is known to the authentication service.

K_p :

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

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 $g^v \bmod n$. 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 $\text{Id}(R_p)$, if it is *insecure*, p is trusted, and $R_q < R_p^8$, then create a message: $\text{Gb}([q, p, g^{v_q}]_{S_{qp}})$, and send it to p .
- Upon receiving a message $\text{Gb}(\dots)$ 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 : $\text{Ga}([q, p, g^{v_q}, \{K_p\}g^{v_p v_q}]_{S_{qp}})$.
- Upon receiving a message $\text{Ga}(\dots)$ 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 $\text{new_key} := K_p$. 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 2.2 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 .

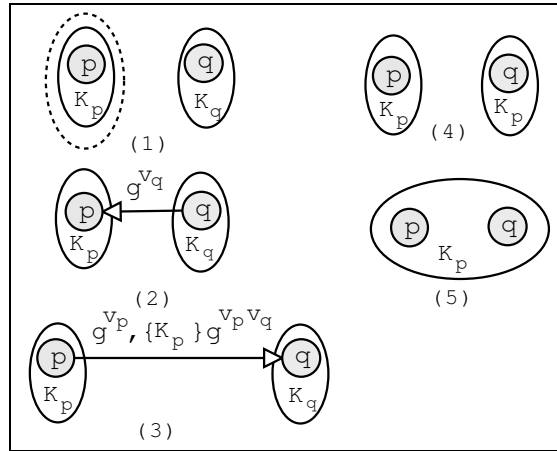


Figure 2.2: **Overview of the merge sequence.** (1) Two components with keys K_p and K_q . Each leader sends `IamAlive` messages. (2) q sends g^{v_q} to p . (3) p sends g^{v_p} and K_p encrypted with $g^{v_p v_q}$ to q . (4) q decrypts K_p and switches his component key to K_p . (5) Components P and Q now merge, after both have the same key.

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 $v_s \in \mathbb{Z}_n$ for which it precomputes g^{v_s} . This value is not revealed by distributing g^{v_s} , nor by raising it to the power of $w \in \mathbb{Z}_n$. 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 transferred, 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*.

⁸Any type of comparison function may be used here.

2.3.2 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 $f_s(lt_s)$. 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: $\text{IamAlive}(R_q, t_q)$.
- Upon receiving an $\text{Id}(R_p, t_p)$, if it is *insecure*, p is trusted, and $R_q < R_p$ ⁹, then create a message: $\text{Gb}([q, p, t_q, t_p, g^{v_q}]_{S_{qp}})$ and send it to p .
- Upon receiving a message $\text{Gb}(\dots)$ 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 : $\text{Ga}([q, p, t_q, g^{v_q}, \{K_q\}g^{v_p v_q}]_{S_{pq}})$.
- Upon receiving a message $\text{Ga}(\dots)$ 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 $lt_s - lt'_s < 10\text{seconds}$ ¹⁰ and that $z = f_s(lt'_s)$. This is a simple computation that only requires remembering f_s . This was designed to allow avoiding any need for perf-session state.

2.3.3 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

⁹Any type of comparison function may be used here.

¹⁰10 is a parameter, this can be changed by the application.

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.

2.3.4 ACL's and Perfect Forward Secrecy

The exchange protocol does not provide PFS. For example, in Figure 2.2, 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 create a new key every time a merge sequence is started, see Figure 2.3.

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.

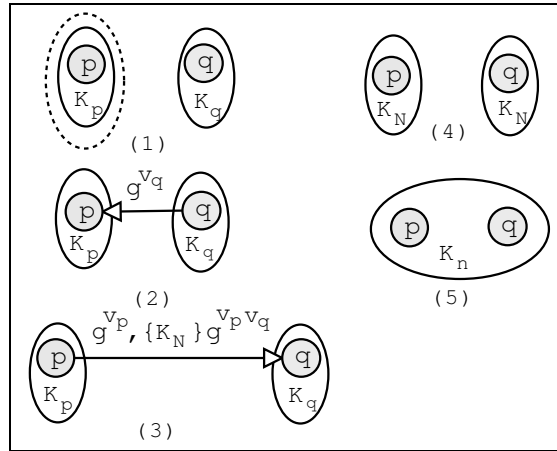


Figure 2.3: **A variation on the merge sequence that guarantees PFS.** (1) Two components with keys K_p and K_q . Each leader sends `IamAlive` messages. (2) q sends g^{v_q} to p . (3) p chooses a new key K_N , sends g^{v_p} and K_N encrypted with $g^{v_p v_q}$ to q . (4) q decrypts K_N and switches his 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).

2.4 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 handling for common scenarios: member join and leave.

We have designed and implemented several efficient rekeying protocols [RBD00a, RBD00b]. In this section, we describe a particularly fast protocol that is designed for solving a specific case: a single member 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 [RBD00b].

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

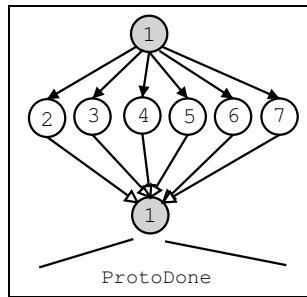


Figure 2.4: The communication pattern of algorithm **Basic**. Member p_1 chooses a new key and sends it to members p_2, \dots, p_7 . Members send acknowledgements back to p_1 . When p_1 receives all acks, it multicasts a *ProtoDone*.

illustrated in Figure 2.4. 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 Logical Key Hierarchy (LKH) family of algorithms [WGL98, WHA98]. In principal, LKH uses a 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 [RBD00a]. The new protocol was called **dLKH**, its performance is depicted in Figure 2.5.

Figure 2.5 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 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. Latency does not grow monotonically because **dLKH** has logarithmic behaviour as a function of group size.

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.

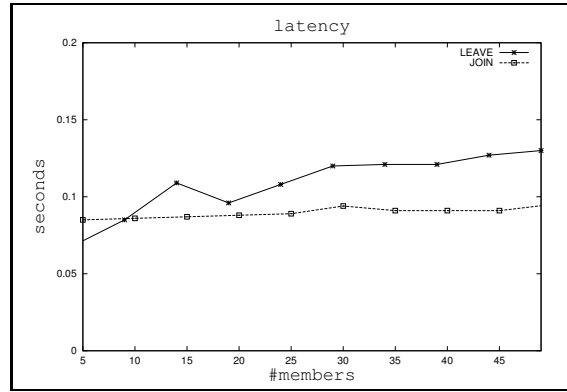


Figure 2.5: Performance of the dLKH algorithm.

The actual communication for a 30 member group was under $20ms$. The Diamond protocol is not based on a virtual tree concept, rather it attempts to make maximal usage of the current set of secure channels, employing it as an infrastructure for key dissemination.

2.4.1 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.

We use a diamond like graph examples of which can be seen in Figure 2.6. 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 .

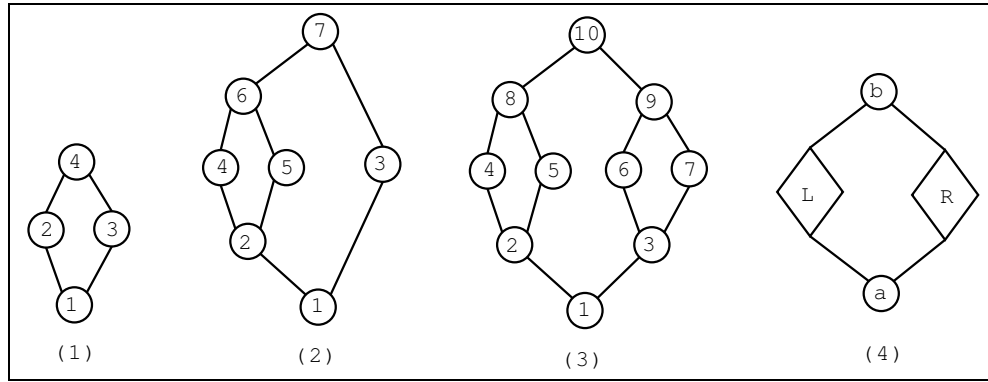


Figure 2.6: Three Examples of diamond structures, and the general scheme. (1) An example with 4 nodes. (2) An example with 7 nodes. (3) An example with 10 nodes. (4) The general diamond structure. Member b is First, member a is Last, L is the left sub-diamond, and R is the right sub-diamond.

- 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 significant 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 .

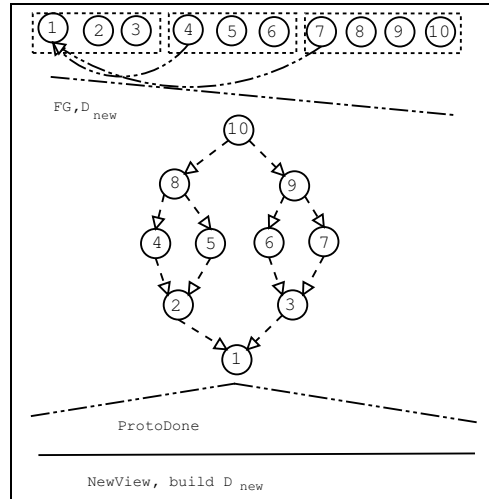


Figure 2.7: The communication pattern of the diamond protocol.

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 2.7 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 Q 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

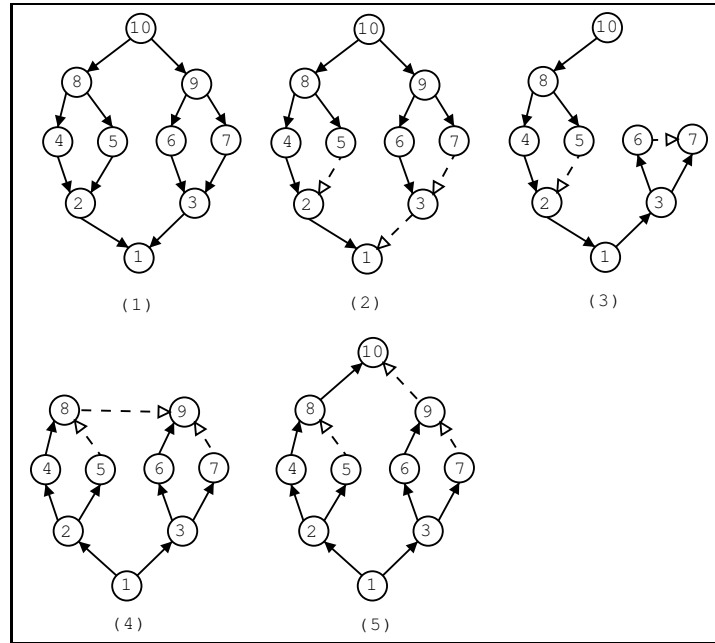


Figure 2.8: Examples of quick schedules. (1) The basic protocol. (2) A 10 member group. (3) Member 9 dies. (4) Member 10 dies. (5) Member 10 rejoins. A full arrow head denotes an encrypted message. An empty arrow head denotes a clear-text message (an ack).

$4/3n$ edges (see Section 2.6). 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 2.8 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 .

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

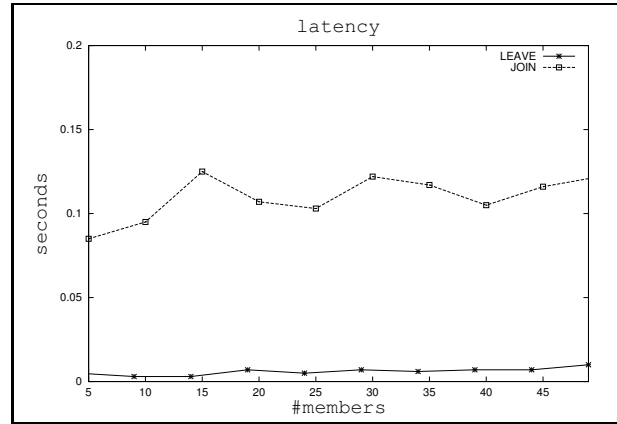


Figure 2.9: Performance of the Diamond algorithm.

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 2.9 for performance measurements.

2.4.2 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*:

- If D has one member, then do nothing.

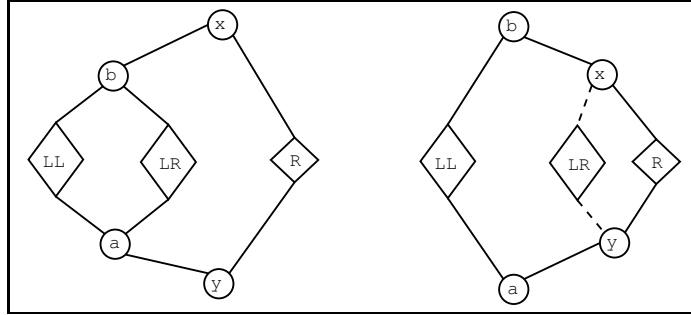


Figure 2.10: A rebalancing step. Diamond R is much smaller than L . We then move it down and merge it with LR . The dotted lines denote new edges.

- 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 2.10. This step involves the creation of two new graph edges. Since this is relatively expensive, we apply balancing only at extreme situations.

Performance

To measure the performance of the balancing algorithm, we measured the number of exponentiations performed on average during the reconstruction phase. Figure 2.11 depicts the number of exponentiations as a function of the number of members.

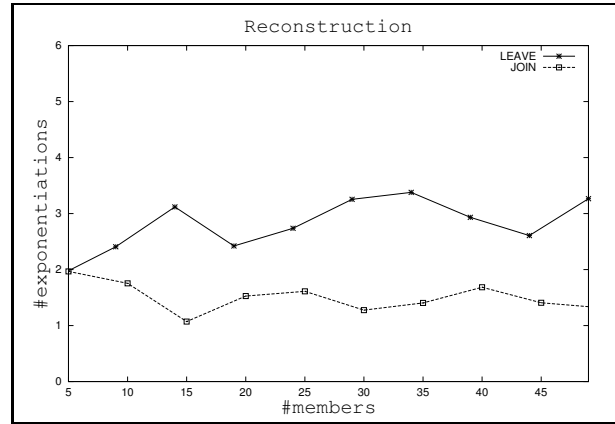


Figure 2.11: Performance of the reconstruction algorithm.

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 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.

2.5 Conclusions

In this chapter we set out to create a security architecture for group communication systems. In the context of the Ensemble system, we created an architecture supporting several novel features:

- modularity.
- Security properties for multiple partitions.
- Dynamic application-defined authorization polices.

- Rekeying in general, and fast rekeying in the presence of a single fault.
- Use of off-the-shelf authentication.

All these represent contributions to the current state-of-the-art.

2.6 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:

$$\begin{aligned} e'(x) &= 2 * e'(x/2 - 1) * (1/2) \\ e'(x) &= e'(x/2 - 1) \end{aligned}$$

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

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

We know that:

$$\begin{aligned} e(4) &= 4 \\ e(10) &= 12 \end{aligned}$$

We conclude that:

$$e(n) = \frac{4}{3}n - \frac{4}{3}$$

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.

Chapter 3

Using AVL Trees for Efficient Group Rekey

3.1 Introduction

This chapter describes an efficient rekeying protocol, based on the idea of a *Logical Key Hierarchy* (LKH) first presented by Wong et al. [WGL98] and Wallner et al. [WHA98].

LKH was designed to solve the rekeying problem for large IP-multicast groups that require protection. This problem, and more generally, the security architecture for IP-multicast groups has received much attention over the years [Bal96, HM97a, HM97b, WHA98, WHA99, WGL98, HCD99a, HCD99b, Mit97, HCB99, BMS99, CGI+99]. The IETF has established the Secure Multicast Group (SMUG) to suggest and standardize a solution. Example applications for secure IP-multicast include Internet video transmissions, news feeds, stock quotes, software updates, and more.

In these setting, a single symmetric key is used to authenticate and optionally encrypt all group messages. A key-server holds the group key, and it is responsible for handing the key to all authenticated and authorized group members. Members may dynamically join and leave the group. Each member is connected to the key-server through a secure channel. To maintain Forward and Backward Confidentiality the group must be rekeyed every membership change. A naive solution is for the key-server to choose a new random key, encrypt and send it to each group member. This solution incurs linear overhead in the number of clients, hence, it does not scale to large groups.

LKH improves the rekey overhead from linear to logarithmic. However, it uses a central key-server which is a single point of failure. It is possible to replicate the key-server, but we chose a more difficult approach. This chapter investigates the adaption of

LKH to a fully distributed setting where no servers may be used. Such a transformation can yield an inefficient algorithm. We shall show a completely distributed, fault-tolerant, efficient protocol for rekeying a GCSs based on LKH.

This chapter is structured as follows:

1. Description of the centralized solution.
2. A naive extension that makes it fully-distributed.
3. Optimization of the initial extension and a study of its performance.

In comparison to the *Diamond* protocol from chapter 2, the protocol described here (*dLKH*) is based on a completely different principal, a *logical key hierarchy*. Where the diamond protocol uses a graph wherein nodes are group members, and edges represent secure channels that connect them together, the *dLKH* protocol uses a graph where nodes are subkeys and edges represent relationships between subkeys.

3.2 The centralized solution (\mathcal{C})

Here we describe the basic LKH protocol using the notion of *key-graphs*. Key-graphs have been put to use in other security protocols, for example, Cliques [KPT00].

A key-graph is defined as a directed tree where the leafs are the group members and the nodes are keys. A member knows all the keys on the way from itself to the root. The keys are chosen and distributed by a key-server. While the general notion of a key-graph is somewhat more general, we focus on binary-trees. In Figure 3.1 we see a typical key-graph for a group of eight members. This will serve as our running example.

Each member p_i shares a key with the server, K_i , using a secret channel. This key is the secure-channel key used for private communication between p_i and the server. It is called the *basic-key*. Each member also knows all the keys on the path from itself to the root. The root key is the group key. In the example, member p_1 knows keys $K_1, K_{12}, K_{14}, K_{18}$. It shares K_1 with the server, K_{12} with p_2 , K_{14} with $\{p_2, p_3, p_4\}$, and K_{18} with members $\{p_2, \dots, p_8\}$. In what follows we denote by $\{M\}_{K_1, K_2}$ a tuple consisting of message M encrypted once with key K_1 , and a second time with K_2 .

The key server uses the basic keys to build the higher level keys. For example, the key-graph in the figure can be built using a single multicast message comprised of three parts:

- Part I: $\{K_{12}\}_{K_1, K_2}, \{K_{34}\}_{K_3, K_4}, \{K_{56}\}_{K_5, K_6}, \{K_{78}\}_{K_7, K_8}$

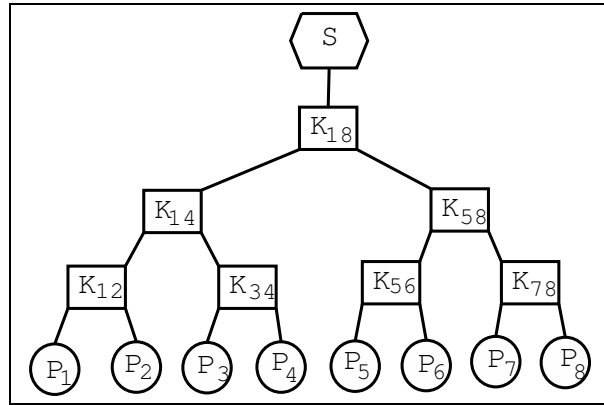


Figure 3.1: A key-graph for a group of eight members.

- Part II: $\{K_{14}\}_{K_{12}, K_{34}}, \{K_{58}\}_{K_{56}, K_{78}}$
- Part III: $\{K_{18}\}_{K_{14}, K_{58}}$

All group members receive this multicast, and they can retrieve the exact set of keys on the route to the root. For example, member p_6 can decrypt from the first part (only) K_{56} . Using K_{56} , it can retrieve K_{58} from the second part. Using K_{58} it can retrieve K_{18} from the third part. This completes the set K_{56}, K_{58}, K_{18} . In general, it is possible to construct any key-graph using a single multicast message.

The group key needs to be replaced if some member joins or leaves. This is performed through key-tree operations.

Join: Assume member p_9 joins the group. S picks a new (random) group-key K_{19} and multicasts: $\{K_{19}\}_{K_{18}, K_9}$. Member p_9 can retrieve K_{19} using K_9 , the rest of the members can use K_{18} to do so.

Leave: Assume member p_1 leaves, then the server needs to replace keys K_{12}, K_{14} , and K_{18} . It chooses new keys K_{24} , and K_{28} and multicasts a two part message:

- Part I: $\{K_{24}\}_{K_2, K_{34}}$
- Part II: $\{K_{28}\}_{K_{24}, K_{58}}$.

The first part establishes the new key K_{24} as the subtree-key for members p_2, p_3, p_4 . The second part establishes K_{28} as the group key.

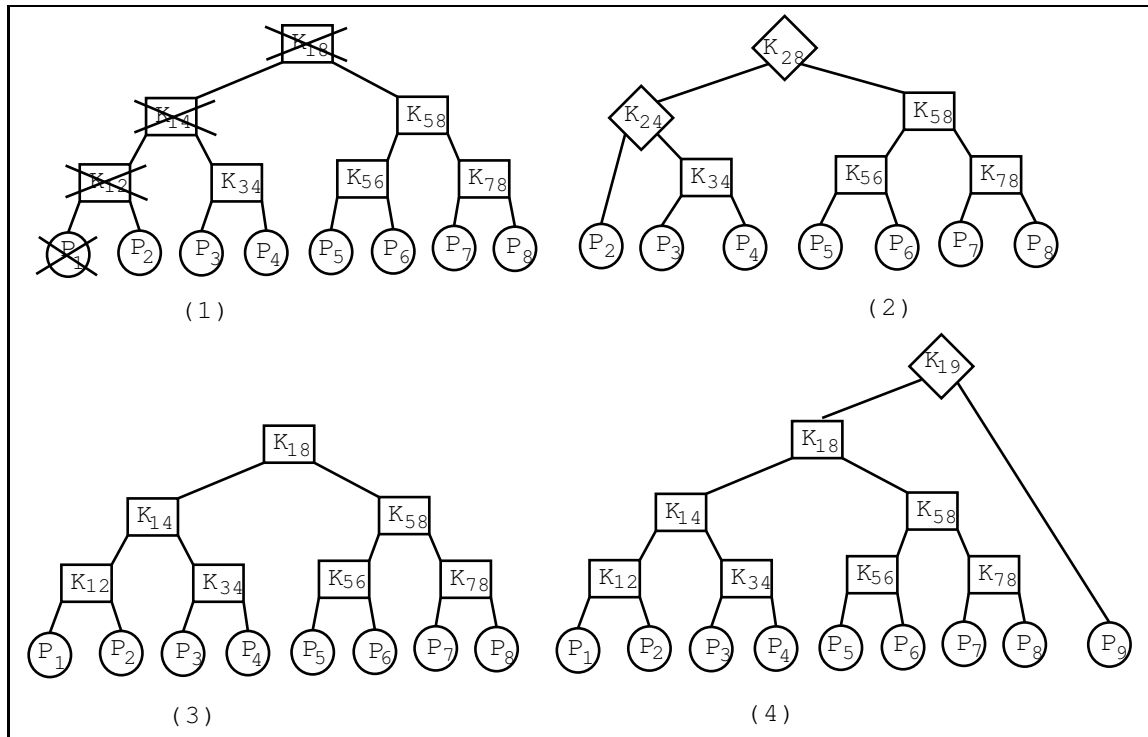


Figure 3.2: **Examples for join/leave cases.** Examples one and two depict a scenario where member p_1 leaves. Examples three and four depict a scenario where member p_9 joins. New keys are marked by diamonds.

In this scheme each member stores $\log_2 n$ keys, while the server keeps a total of n keys. The server uses n secure channels to communicate with the members. The protocol costs exactly one multicast message (acknowledgments are not discussed here). The message size, as a multiply of the size of a key K , is as follows:

Construction	Leave	Join
$2Kn$	$2K \log_2 n$	$2K$

Trees become imbalanced after many additions and deletions, and it becomes necessary to rebalance them. A simple rebalancing technique is described in [MRR99]. However, their scheme is rather rudimentary. For example, in some full binary key-tree T , if most of the right hand members leave, the tree becomes extremely unbalanced. The simple scheme does not handle this well. Our scheme can handle such extreme cases.

The example can be generalized to a *balanced merge* (**Bal_Merge**) procedure. In what follows we denote by $h(T)$ the height of tree T . To describe the structure of a tree we use a recursive data structure that contains nodes of two kinds:

- $Br(L, K, R)$: An inner-node with left subtree L , right subtree R , and key K .
- $Leaf(p_i)$: A leaf node with member p_i .

The simplest tree contains a single member, $Leaf(p_x)$. A single member tree does not have a key, since a single member can communicate with itself alone. Such communication does not need protection. The height of a tree $Leaf(p_x)$ is 1.

Assume that L and R are two subtrees that we wish to merge. W.l.o.g we can assume that $h(L) \geq h(R)$. **Bal_Merge** will find a subtree L' of L such that $1 \geq h(R) - h(L') \geq 0$, and create a new tree T where L' is replaced by $Br(L', K_{new}, R)$. The merge procedure is performed at the key-server, that has complete knowledge of all keys and the structure of existing key-trees. The local computation is a simple tree recursion. The message size required to notify the group is of size $O(\log_2 n)$. This is because R 's members must know all keys above K_{new} .

In a general key-graph, there is no bound on depth as a function of the number of leaves. If a member that is situated very deep in the tree is removed, then the tree is split into many disconnected fragments. Therefore, it is desirable to have balanced trees. We propose using AVL trees [AL62] for this purpose. Assume that T_1 and T_2 are AVL trees. We shall show that the merged tree is also AVL. First, we state more carefully the **Bal_Merge** algorithm, in the form of pseudo-code.

Algorithm **Balanced Merge**(T_1, T_2):

```

if  $h(T_1) == h(T_2)$  then {
  choose a new key  $K_{new}$ 
  return  $Br(T_1, K_{new}, T_2)$ 
}
if  $h(T_1) > h(T_2)$  then {
  assume  $T_1 = Br(L, K_{T_1}, R)$ 
  if  $h(L) < h(R)$  then return ( $Br(merge(L, T_2), K_{T_1}, R)$ )
  else return  $Br(L, K_{T_1}, merge(T_2, R))$ 
}
if  $h(T_1) < h(T_2)$  then {
  the same as above, reversing the names  $T_1, T_2$ .

```

}

Claim 1 : Merging two AVL trees T_1 and T_2 results in an AVL tree whose height is at most $1 + \max(h(T_1), h(T_2))$.

Proof: The proof is performed by induction on the difference in height: $h(T_1) - h(T_2)$.

- Base case: If $h(T_1) = h(T_2)$ then the merged tree is $T = Br(T_1, K_{new}, T_2)$. The depth of the T is $h(T_1) + 1$.
- Induction: We assume that if $\|h(T_1) - h(T_2)\| < k$ then the theorem holds. W.l.o.g $h(T_1) > h(T_2)$. If $h(T_1) - h(T_2) = k$ then T_1 is of height at least two, and $T_1 = Br(L, key, R)$. There are two cases:
 - $h(L) < h(R)$: The result will be, $T = Br(merge(T_2, L), key, R)$. By induction we have that $merge(T_2, L)$ is AVL, and that its height is at most $h(R) + 1$. Hence, the difference in height between the left and right children of T is no more than 1, and T 's height conforms to $h(T) \leq h(R) + 2 = 1 + \max(h(T_1), h(T_2))$
 - $h(L) \geq h(R)$: The result will be $T = Br(L, key, merge(T_2, R))$. By induction we have that $merge(T_2, R)$ is of height that is at most $h(L) + 1$. Hence, the difference in height between the left and right children of T is no more than 1, and T 's height conforms to $h(T) \leq h(L) + 2 = 1 + \max(h(T_1), h(T_2))$

□

Two AVL trees can be merged into a single AVL tree at low cost. This can be generalized to merging a set of trees. Sets of trees are interesting since they occur naturally in leave operations. For example, if a member leaves an AVL tree T , then T is split into a set of $\log_2 n$ subtrees. These subtrees, in turn, are AVL and they can be merged into a new AVL tree.

The cost of a merge operation is measured by the size of the required multicast message. We are interested in the price of merging m AVL trees, with n members. First, we shall use a naive but imprecise argument to show that the price is bounded by $2K(m - 1) + K(\max\{h(T_i)\} - \min\{h(T_i)\})$. Then we shall provide a proof showing this to be in fact $3K(m - 1) + K(\max\{h(T_i)\} - \min\{h(T_i)\})$.

We sort the trees by height into a set $\{T_1, T_2, \dots, T_m\}$. They are merged two at a time: first the smallest two into T_{12} , then T_{12} and T_3 into T_{13} etc. The price for merging two trees is $2K + K\|h(T_1) - h(T_2)\|$. Assuming naively that $h(\text{Bal_Merge}(T_x, T_y)) = \max(h(T_x), h(T_y))$ then the total price is the height difference between the deepest tree, and the shallowest one. Hence, the total price is: $2K(m - 1) + K(\max\{h(T_i)\} - \min\{h(T_i)\})$.

The naive argument fails to take into account the fact that merged trees conform to $h(\text{Bal_Merge}(T_x, T_y)) \leq 1 + \max(h(T_x), h(T_y))$. Therefore, we insert the trees into a heap. Iteratively, the smallest two trees are popped from the heap, merged together, and put back into the heap.

Claim 2 : *The cost of a balanced merge of m trees $\mathcal{T}_m = \{T_1, \dots, T_m\}$ is bounded by $3K(m - 1) + K(\max\{h(T_i)\} - \min\{h(T_i)\})$.*

Proof: The proof is by induction. For two trees T_1 and T_2 , the price is simply $2K + K\|h(T_1) - h(T_2)\|$.

Assume the claim is true for up to $m - 1$ trees. We shall show for m . We proceed according to the heap discipline and merge the two smallest trees, T_1 and T_2 into $T_{1\oplus 2}$. We get a heap with trees $\mathcal{T}_{m-1} = \{T_{1\oplus 2}, T_3, \dots, T_m\}$.

The cost for merging the trees in \mathcal{T}_{m-1} is, by induction, $3K(m-2) + K(\max\{h(T_i)\}_{\mathcal{T}_{m-1}} - \min\{h(T_i)\}_{\mathcal{T}_{m-1}})$. The cost for merging T_1 and T_2 is $K\|h(T_1) - h(T_2)\| + 2K$.

We know that:

- 1) $\max\{h(T_i)\}_{\mathcal{T}_{m-1}}$ is bounded by $\max\{h(T_i)\}_{\mathcal{T}_m} + 1$.
- 2) $\min\{h(T_i)\}_{\mathcal{T}_{m-1}} \geq \min\{h(T_i)\}_{\mathcal{T}_m} + \|h(T_1) - h(T_2)\|$

Hence, the sum:

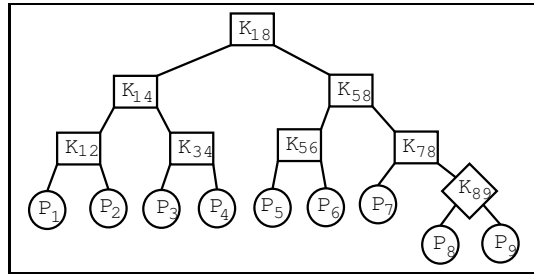
$K(\max\{h(T_i)\}_{\mathcal{T}_{m-1}} - \min\{h(T_i)\}_{\mathcal{T}_{m-1}}) + 3K(m - 2) + 2K + K\|h(T_1) - h(T_2)\|$
is bounded by:

$$\begin{aligned} & K((\max\{h(T_i)\}_{\mathcal{T}_m} - \min\{h(T_i)\}_{\mathcal{T}_m}) + 1) + 3K(m - 2) + 2K = \\ = & K(\max\{h(T_i)\}_{\mathcal{T}_m} - \min\{h(T_i)\}_{\mathcal{T}_m}) + 3K(m - 1) \end{aligned}$$

□

In Figure 3.4 we can see a balanced tree, after a join.

The table below lists the costs of balanced join and leave events. This shows that AVL trees do not cost significantly more than standard key-graphs. We shall use them henceforth.

Figure 3.4: **Balancing the join case.**

Construction	Leave	Join
$2Kn$	$2K \log_2 n$	$K \log_2 n$

Note that the technique described so far does not maintain backward confidentiality (BCY). For example, in the join case, member p_9 learns the set of keys used before by members $\{p_1, \dots, p_8\}$. To guaranty PBS, we must replace all keys on the path from the joiner to the root with fresh keys.

Currently, we do not address the issue of BCY in full. However, we allow the application to determine a time period after which the whole tree is discarded and built from scratch. By default this timer is set to 24 hours. Hence, BCY is guaranteed up to 24 hours.

3.3 The basic (\mathcal{B}) distributed protocol

The centralized solution \mathcal{C} does not satisfy our objectives because it relies on a centralized server which has knowledge of the complete key-graph. We require a completely distributed solution, without a single point of failure. While our algorithm is based on \mathcal{C} , each member keeps only $\log_2 n$, keys and members play symmetric roles. The algorithm presented here is called Basic (\mathcal{B}) since it is rather simple and inefficient. We optimize it in the next sections.

We shall say that a group of members G has key-graph T if the set of leaf-nodes that T contains is equal to G . Furthermore, each member of G must possess exactly the set of keys on the route from itself to the root of T . The leader of T , denoted C_T , is the leaf that is the left-most in the tree. Since members agree on the tree structure, each member knows if it is a leader or not. We denote by $M(T)$ the members of T , and by K_T the top key in T (if one exists).

We can see that members in G must have routes in the tree that “match”, i.e., when pulled together they make a tree that is the same as a tree obtained by the centralized algorithm. To perform this ‘magic’ in a distributed environment we rely on Virtual Synchrony (see more below 3.3.1).

To motivate the general algorithm, we shall describe a naive protocol that establishes key-trees for groups of size 2^n . We use the notion of subtrees *agreeing* on a mutual key. Informally, this means that the members of two subtrees L and R , securely agree on a mutual encryption key. The protocol used to agree on a mutual key is as follows:

Agree(L, R):

1. C_L chooses a new key K_{LR} , and sends it to C_R using a secure channel.
 2. C_L encrypts K_{LR} with K_L and multicasts it to L ; C_R encrypts K_{LR} with K_R and multicasts to R .
 3. All members of $L \cup R$ securely receive the new key.
-

The *agree* primitive costs one point-to-point and two multicast messages. Its effect is to create the key-graph $Br(L, K_{LR}, R)$ for the members of $M(L) \cup M(R)$.

Below is an example for the creation of a key-tree for a group of 8 members (see Figure 3.5):

1. Members 1 and 2 agree on mutual key K_{12}
 Members 3 and 4 agree on mutual key K_{34}
 Members 5 and 6 agree on mutual key K_{56}
 Members 7 and 8 agree on mutual key K_{78}
2. Members 1,2 and 3,4 agree on mutual key K_{14}
 Members 5,6 and 7,8 agree on mutual key K_{58}
3. Members 1,2,3,4 and 5,6,7,8 agree on mutual key K_{18}

Each round’s steps occur concurrently. In this case, the algorithm takes 3 rounds, and each member stores 3 keys. This protocol can be generalized to any number of members.

Base case: If the group contains 0 or 1 members, then we are done.

Recursive step ($n = 2^{k+1}$): Split the group into two subgroups, M_L and M_R , containing each 2^k members. Apply the algorithm recursively to M_L and M_R . Now, subgroup M_L possesses key-tree L , and subgroup M_R possesses key-tree R . Apply the agreement primitive to L and R such that they agree on a group key.

This protocol takes $\log_2 n$ rounds to complete. Each member stores $\log_2 n$ keys.

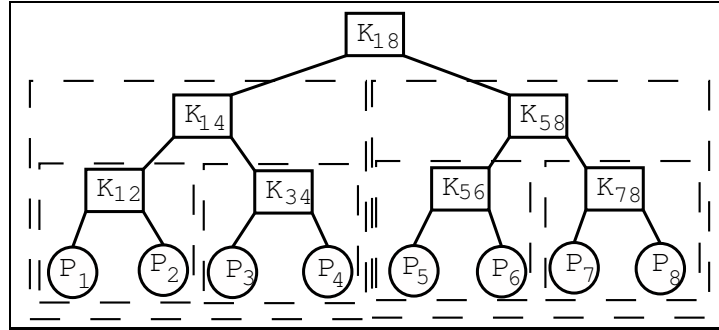


Figure 3.5: **The naive protocol for a group of eight members. Groups that agree on a key are surrounded by a dashed rectangle.**

Clearly, the naive protocol builds key-graphs for groups of members of size 2^n . Note that no member holds more than $\log_2 n$ keys, and the total number of keys is n .

In the lifetime of the group members may join, leave, the group may partition and merge with other group components. For example, when member p_1 leaves, the key-graph is split into three pieces. When p_9 joins, the new key-tree is a merging of the key-graphs containing $\{p_1, \dots, p_8\}$, and $\{p_9\}$. To handle these events efficiently, the computation of the new key-tree, as a function of existing subtrees, is performed by the group leader. After a view-change, the leader requests subtree *layouts* from subleaders. A tree *layout* is a symbolic description of a key-tree. The leader combines the subtree layouts and multicasts a new complete layout, coupled with a schedule describing how to merge subtrees together.

Note that the leader is not allowed to learn the actual keys used by other members. This is the reason for using symbolic representations.

We now extend *agree* to conform to the merge step described in the previous section. If L and R are two subtrees that should be merged together then, assuming $h(L) \geq h(R)$, the leader finds a subtree L' of L such that $1 \geq h(R) - h(L') \geq 0$. The protocol is as follows. $C_{L'}$ denotes the leader within L' , C_R the leader within R , C_T the tree leader. The algorithm as a whole is initiated by C_T .

Extended Agree(L, R):

1. C_T initiates the protocol by multicasting the new layout and schedule.
 2. $C_{L'}$ chooses a new key $K_{L'R}$, and sends it to C_R using a secure channel.
 3. $C_{L'}$ encrypts $K_{L'R}$ with $K_{L'}$ and multicasts it.
 $C_{L'}$ encrypts all keys above $K_{L'R}$ with $K_{L'}$ and multicasts them.
 C_R encrypts $K_{L'R}$ with K_R and multicasts it.
 4. All members of $L \cup R$ securely receive the new key.
-

Below are two examples, the first is a leave scenario, the second is a join scenario. Both reference Figure 3.2.

In figures (1) and (2), member p_1 fails. All members receive a view-change notification, and locally erase from their key-trees all keys of which p_1 had knowledge. The key-graph has now been split into three pieces T_1, T_2, T_3 containing: $\{p_2\}, \{p_3, p_4\}, \{p_5, p_6, p_7, p_8\}$ respectively. The leader collects the structures of T_1, T_2 , and T_3 from the subleaders p_2, p_3 , and p_5 . It decides that the merged tree will have the layout described in Figure 3.6(2). It multicasts this structure to the group, with instructions for T_1 to merge with T_2 into T_{12} , and later for T_{12} to merge with T_3 .

In figures (3) and (4) member p_9 joins the group. All members receive a view-change that includes member p_9 . The key-graph is now comprised of two disjoint components: T_1 containing: $\{p_1, \dots, p_8\}$, and T_2 containing $\{p_9\}$. The leader collects the structures of T_1 and T_2 from the subleaders p_1 and p_9 . It decides that the merge tree will have the layout described in Figure 3.6(4). It multicasts this structure to the group with instructions for T_1 to merge with T_2 .

Protocol \mathcal{B} takes 2 stages in case of join, $\log_2 n$ stages in case of leave, and $\log_2 n$ in case of tree construction. In general, the number of rounds required is the maximal depth of the set of new levels in the tree that must be constructed. The optimized protocol improves this result to two communication rounds, regardless of the number of levels that must be constructed.

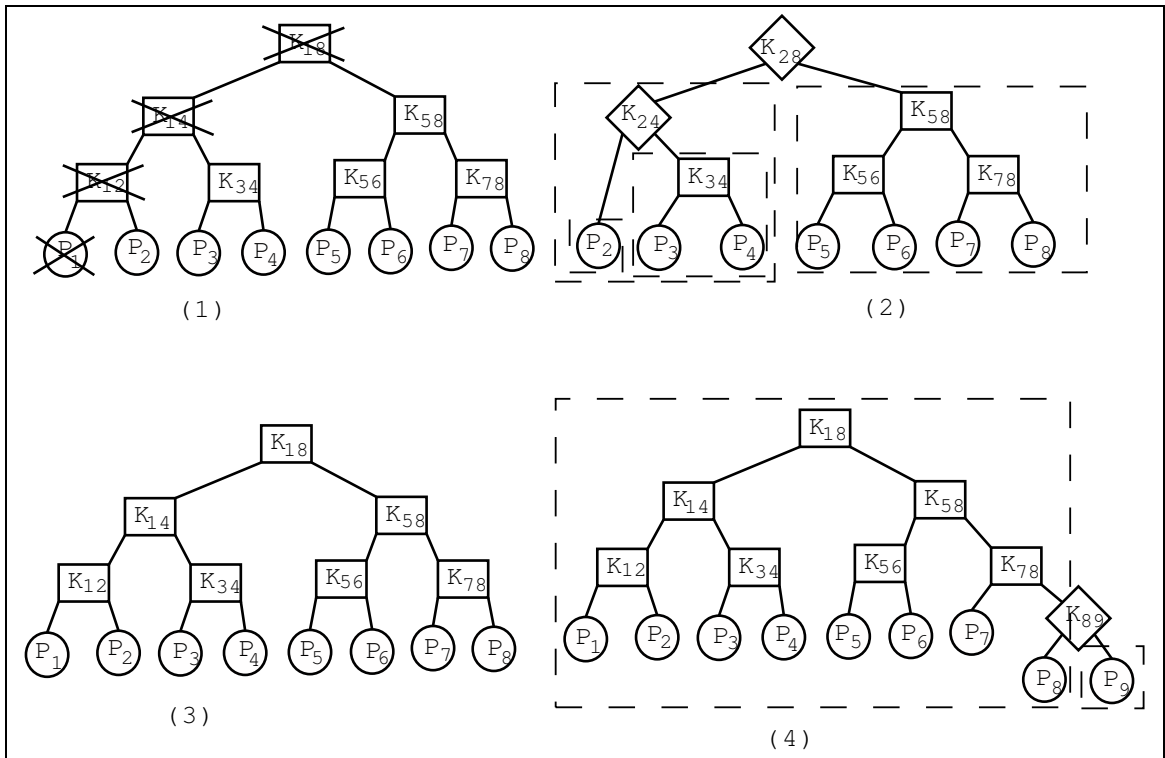


Figure 3.6: **Single member join/leave cases.** Examples (1) and (2) show the actions performed when member p_1 leaves the group. Examples (3) and (4) show the actions performed when member p_9 joins.

3.3.1 Virtual Synchrony

Protocol \mathcal{B} makes use of the GCS guarantees on message delivery, and view agreement. All point-to-point and multicast messages are reliable, and sender-ordered. Furthermore, they arrive in the view they are sent, hence, an instance of the protocol is run in a single view. If the view changes, due to members leaving or joining, then the protocol is restarted.

The guarantee that members agree on the view is crucial. It allows members to know what role they play in the protocol, be it leader, sub-leader, or no-role.

Although the protocol is multi-phased, we can guarantee an *all-or-nothing* property. Either all members receive the new key-tree, and the new group-key, or the protocol fails (and all members agree on this). To see this, examine the last step of the protocol, where \mathcal{B} has been completed, and acknowledgments are collected by the leader. Once acknowledgment collection is complete, a *ProtoDone* message is multicast by the leader.

Members that receive *ProtoDone* know that the protocol is over, and that other members also possess the same key-tree.

Should failures occur prior to the last multicast, then the protocol is restarted with the previous sub-trees. If a failure occurs afterwards, then remaining members agree on the key-tree and group-key. Failed members will be removed in the next view, and the protocol will be restarted.

This description is also valid for the optimized solution described in the next section.

3.4 Optimized solution (\mathcal{O})

First, we describe an example showing that \mathcal{B} can be substantially improved. Then we describe the optimized algorithm \mathcal{O} .

Examine the case where member p_1 leaves the group. There are three components to merge, T_1, T_2, T_3 and two tree levels to reconstruct. This requires two *agree* rounds. However, we can transform the protocol and improve it substantially by having p_2 choose all required keys at the same time. See Figure 3.7(1).

The protocol used is as follows:

- Round I: p_2 chooses K_{24}, K_{28} . It sends K_{24} to the leader of T_2 , K_{28} to the leader of T_3 . Finally, it multicasts $\{K_{28}\}_{K_{24}}$ so that members of T_2 can retrieve K_{28} .
- Round II: The leader of T_2 multicasts $\{K_{24}\}_{K_{34}}$.
The leader of T_3 multicasts $\{K_{28}\}_{K_{58}}$.

A larger example is depicted in Figure 3.7(2). Here, in a 16 member group, p_1 leaves. The regular algorithm requires three *agree* rounds, costing six communication rounds. This can be optimized to cost only two communication rounds, similarly to the above example.

- Round I: p_2 chooses $K_{24}, K_{28}, K_{2,16}$. It sends K_{24} to the leader of T_2 , K_{28} to the leader of T_3 , and $K_{2,16}$ to the leader of T_4 . Finally, it multicasts $\{K_{28}\}_{K_{24}}, \{K_{2,16}\}_{K_{28}}$, so that members of T_2 and T_3 can retrieve K_{28} and $K_{2,16}$.
- Round II: The leader of T_2 multicasts $\{K_{24}\}_{K_{34}}$.
The leader of T_3 multicasts $\{K_{28}\}_{K_{58}}$.
The leader of T_4 multicasts $\{K_{2,16}\}_{K_{9,16}}$.

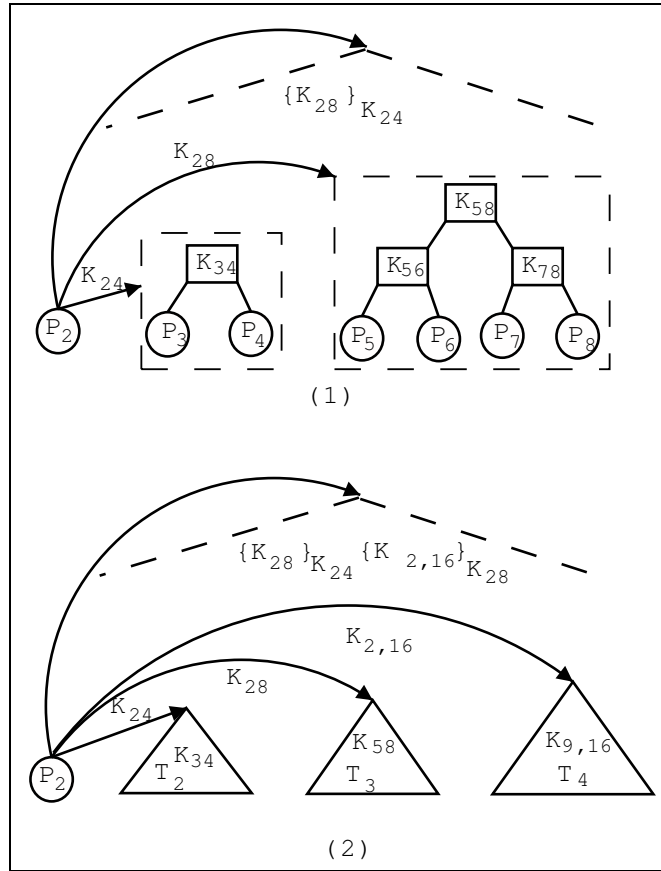


Figure 3.7: **Optimizing the leave case.**

Prior to presenting the general solution, we provide an example for the creation of a complete key-tree of a group of eight members (see Figure 3.8).

In the first stage (Figure 3.8(1)), members engage in a protocol similar to the leave case. It is performed in parallel by all leaders in the group. New keys are denoted by full arrows. Multicast keys are written underneath member nodes. In the second stage, members pass received keys along, using multicast.

Round 1:

- p_1 chooses K_{12}, K_{14}, K_{18} .
- $m_1 \rightarrow m_2 : K_{12}$
- $m_1 \rightarrow m_3 : K_{14}$
- $m_1 \rightarrow m_5 : K_{18}$
- $p_1 \rightarrow G : \{K_{14}\}_{K_{12}}, \{K_{18}\}_{K_{14}}$

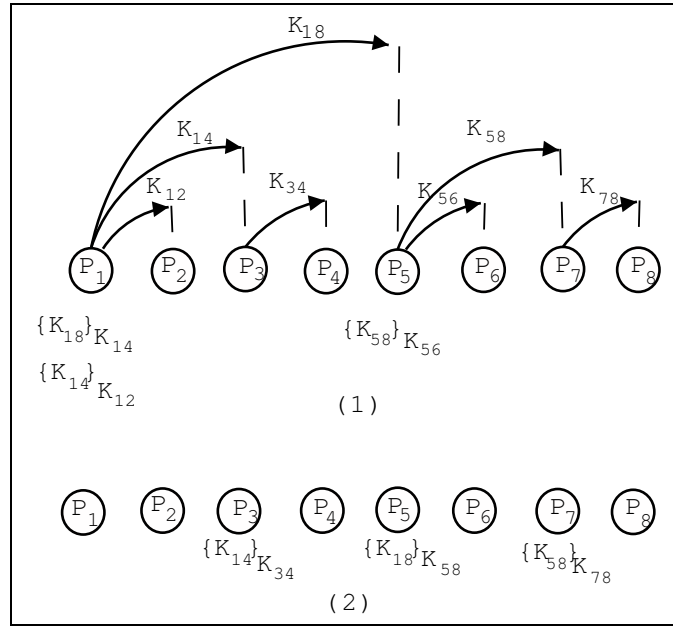


Figure 3.8: **Tree construction for an eight member group.**

p_3 chooses K_{34} .
 $m_3 \rightarrow m_4 : K_{34}$
 p_5 chooses K_{56}, K_{58} .
 $m_5 \rightarrow m_6 : K_{56}$
 $m_5 \rightarrow m_7 : K_{58}$
 $p_1 \rightarrow G : \{K_{58}\}_{K_{56}}$
 p_7 chooses K_{78} .
 $m_7 \rightarrow m_8 : K_{78}$

Round 2:

$p_3 \rightarrow G : \{K_{14}\}_{K_{34}}$
 $p_5 \rightarrow G : \{K_{18}\}_{K_{58}}$
 $p_7 \rightarrow G : \{K_{58}\}_{K_{78}}$

At the end of the protocol, all members have all the keys. For example, member p_1 has all the keys at the end of the first round. It learns no other keys as the protocol progresses. Member p_8 receives in round one K_{78} . In round two it receives: $\{K_{18}\}_{K_{58}}$ and $\{K_{58}\}_{K_{78}}$. Thus, it can retrieve K_{58} , and K_{18} .

We now generalize the protocol. We shall describe how the leader, after retrieving subtree layouts $\{T_1, \dots, T_m\}$, creates an optimal schedule describing how to merge the

subtrees together. Since members simply follow the leader's schedule, we shall focus on the scheduling algorithm (**Sched**).

Sched works recursively. First, we use the centralized algorithm, and merge T_1, \dots, T_m together. There are $m - 1$ instances where merge operations occur, each such operation is translated using a simple template.

Assuming subtrees L and R are to be merged, and $h(L) \geq h(R)$, then there is a subtree L' of L with which to merge R . Mark the list of keys on the route from $K_{L'}$ to the root of L by L_{path} .

The template is:

Stage 1:

$C_{L'}$ choose a new key $K_{L'R}$ and:

$$C_{L'} \rightarrow C_R : K_{L'R}$$

$$C_{L'} \rightarrow G : \{K_{L'R}\}_{K_{L'}}, \{L'_{path}\}_{K_{L'R}}$$

Stage 2:

$$C_R \rightarrow G : \{K_{L'R}\}_{K_R}$$

Building a complete schedule requires performing **Sched** iteratively for $\{T_1, \dots, T_k\}$ and storing the set of instructions in a data structure with two fields: one for the first stage, and another for the second stage. Each application of **Sched** adds two multicasts and one point-to-point message to the total. The number of rounds remains two.

An example for a more general case of **Sched** can be seen in Figure 3.9. The figure shows a set of three subtrees T_1, T_2, T_3 that need to be merged.

The resulting schedule will be:

Stage1:

C_{T_1} chooses K_1 .

$$C_{T_1} \rightarrow C_{T_2} : K_1$$

$$C_{T_1} \rightarrow G : \{K_1\}_{K_{T_1}}$$

$C_{T'_2}$ chooses K' .

$$C_{T'_2} \rightarrow C_{T_3} : K'$$

$$C_{T'_2} \rightarrow G : \{K'\}_{K_{T'_2}}$$

$$C_{T'_2} \rightarrow G : \{K_2\}_{K'}$$

Stage2:

$$C_{T_2} \rightarrow G : \{K_1\}_{K_{T_2}}$$

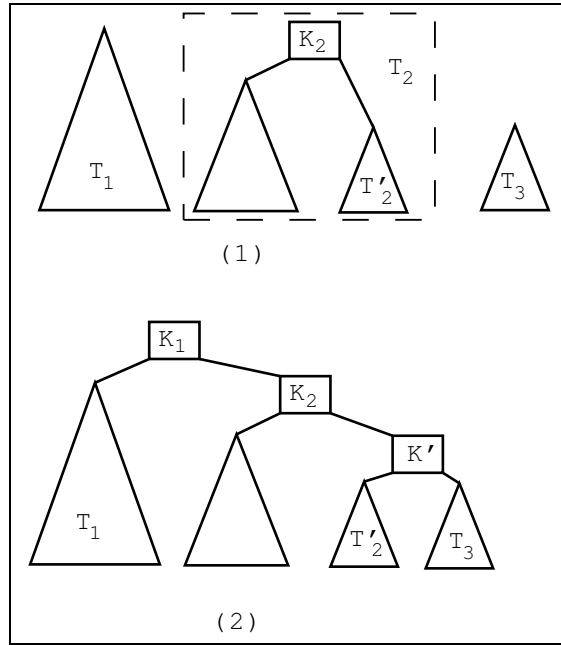


Figure 3.9: **The leader of T'_2 must send K_2 to T_3 .**

$$C_{T_3} \rightarrow G : \{K'\}_{K_{T_3}}$$

For example, all members of T_3 will receive: $\{K'\}_{K_{T_3}}$ from C_{T_3} , $\{K_2\}_{K'}$ from $C_{T'_2}$, and $\{K_1\}_{K_2}$ from C_{T_2} . Using K_{T_3} they can decrypt K' , K_2 , and finally K_1 . Hence, they can retrieve all the keys on the route to the root. Notice that $K_{T'_2}$ must send the path inside T_2 to all members of T_3 . In this case, the path included only K_2 .

Claim 3 *Algorithm Sched produces a schedule that guarantees all members receive exactly all the keys on the path to the root.*

Proof: The proof is performed by induction on the number of merge steps. If m trees need to be merged, then there are $m - 1$ steps to perform. The induction invariant is that the leader of a subtree knows all keys on the path to the root of its subtree in stage 1.

Base case:

If m trees need to be merged, then the base case is merging the first two.

For two trees L, R , we assume w.l.o.g that $h(L) \geq h(R)$. At the end of the protocol, C_R learns $K_{L'R}$, and the set of multicasts: $\{K_{L'R}\}_{K_{L'}}$, $\{L_{path}\}_{K_{L'R}}$, $\{K_{L'R}\}_{K_R}$ is sent. Members of R learn (exactly) $K_{L'R}$ and L_{path} . Members of L learn $K_{L'R}$, and the rest of the members learn nothing.

Note that the leader of L' chooses the new key $K_{L'R}$, hence the invariant holds.

Induction step:

Assume $k - 1$ steps can be performed in two stages. Now we need to show for the k 'th step. We perform the first $k - 1$ steps, and get a tree L coupled with a two step schedule. The k 'th step consists of scheduling a merge between the smallest tree in the heap R and L . Assume w.l.o.g that $h(L) \geq h(R)$, and that $h(R) = h(L')$, where L' is a subtree of L . The final tree will be L with L' replaced with $Br(L', K_{L'R}, R)$.

$C_{L'}$ chooses $K_{L'R}$ and passes it to C_R . It knows $K_{L'}$, hence it multicasts $\{K_{L'R}\}_{K_{L'}}$. All members of L' will know $K_{L'}$ at the end of stage two, hence they will also know $K_{L'R}$.

Leader C_R knows K_R in the first stage, hence, it multicasts $\{K_{L'R}\}_{K_R}$ in the second stage. By induction, all members of R will know K_R at the end of the second stage, hence they will also be able to retrieve $K_{L'R}$.

Furthermore, In the first stage, $C_{L'}$ multicasts $\{L_{path}\}_{K_{L'R}}$. This guarantees that members of R will know all the keys on the path to the root at the end of the second stage.

Note that the leader of L' chooses $K_{L'R}$, hence the invariant holds.

□

3.4.1 Three round solution (\mathcal{O}_3)

The optimized solution \mathcal{O} is optimized for rounds, i.e., it reduces the number of communication rounds to a minimum. However, it is not optimal with respect to the number of multicast messages. Each subtree-leader sends $\log_2 n$ multicast messages, potentially one for each level of recursion. Since there are $n/2$ such members, we may have up to $O(n \times \log_2 n)$ multicast messages sent in a protocol run.

Here we improve \mathcal{O} and create protocol \mathcal{O}_3 . Protocol \mathcal{O}_3 is equivalent to \mathcal{O} except for one detail, in each view, a member p_x is chosen. All subtree-leaders, in stage 2, send their multicasts messages point-to-point to p_x . Member p_x concatenates these messages and sends them as one (large) multicast. The other members will unpack this multicast and use the relevant part. This scheme reduces costs to $n/2$ point-to-point messages from

subtree leaders to p_x , and one multicast message by p_x . Hence, we add another round to the protocol but reduce multicast traffic.

3.4.2 Costs

Here, we compare the three-round solution with the regular centralized solution. Such a comparison is inherently unfair, since the distributed algorithm must overcome many obstacles that do not exist for the centralized version. To even out the playing ground somewhat, we do not take into account (1) collection of acknowledgments (2) sending tree-layouts. We compare three scenarios — building the key-tree, the join algorithm and the leave algorithm. We use tables to compare the solutions and we use the following notations:

pt-2-pt: The number of point-to-point messages sent.

multicast: The number of multicast messages sent.

bytes: The total number of bytes sent

rounds: The number of rounds the algorithm takes to complete

First, we compare the case of building a key-tree for a group of size 2^n where there are no preexisting subtrees. The following table summarizes the costs for each algorithm:

	# pt-2-pt	# multicast	# bytes	# rounds
\mathcal{C}	0	1	$2Kn$	1
\mathcal{O}_3	$1.5n - 1$	1	$3Kn$	3

Since the group contains n members, there are $n - 1$ keys to create. A single secure point-to-point message is used to create each key. There are $n/2$ members that act as subleaders. These all send multicast messages in the first and second stages. These messages are converted into point-to-point messages sent to the leader. All counted, there are $1.5n - 1$ point-to-point messages. The number of bytes is broken down as follows: (1) the $n - 1$ secure point-to-point messages cost Kn . (2) The messages to the leader cost a total of Kn (3) The final multicast bundles together all point-to-point messages, and it therefore costs an additional Kn . The cost is in fact Kn for the second and third items because the final multicast contains essentially all the group key-tree, except for the lowest level.

The leave algorithm costs:

	# pt-2-pt	# multicast	# bytes	# rounds
\mathcal{C}	0	1	$2K \log_2 n$	1
\mathcal{O}_3	$\log_2 n$	1	$3K \log_2 n$	3

The join algorithm costs:

	# pt-2-pt	# multicast	# bytes	# rounds
\mathcal{C}	0	1	$\log_2 n$	1
\mathcal{O}_3	2	1	$\log_2 n$	3

3.5 Performance

This section describes the performance of our protocol. As described previously (Section 2.4), our test-bed was a set of 20 PentiumIII 500Mhz Linux2.2 machines, connected by a switched 10Mbit/sec Ethernet. Machines were lightly loaded during testing. A group of n members was created, a set of member join/leave operations was performed, and measurements were taken. To simulate real conditions, we flushed the cache once every 30 rekey operations, and discarded “cold-start” results. To simulate more than 20 members, we ran a couple of processes on the same machine.

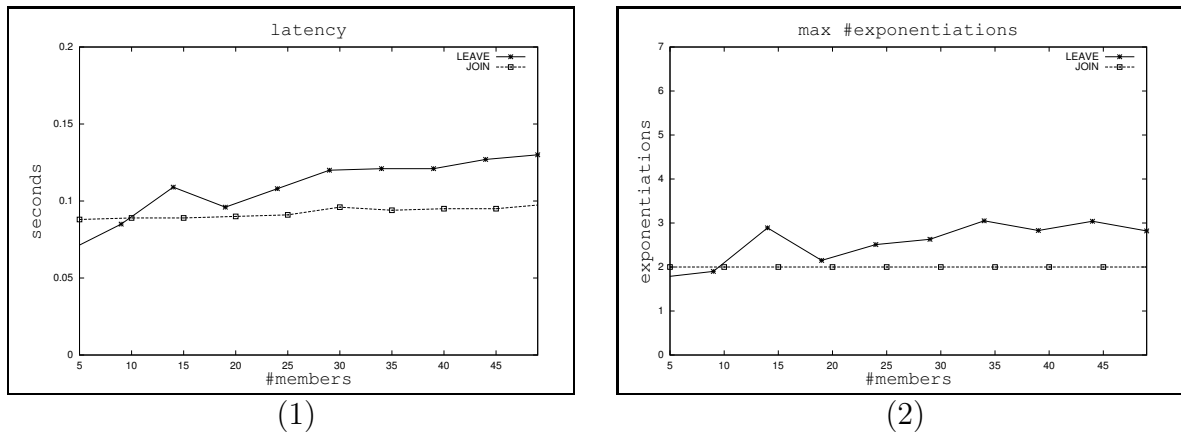


Figure 3.10: (1) Latency of the dLKH algorithm for join/leave operations. (2) The maximal number of exponentiations on the critical path.

Figure 3.10(1) describes the latency of join/leave operations. Performance in the join case is optimal. There are exactly two integer exponentiations on the critical path. This

is optimal. For example, if a new member is added on the left hand side of the tree, then that member must create connections to $\log_2 n$ components. The improvement comes from using a *tree-skewing* technique. When a single member joins, we attempt to add it as far to the right as possible. For example, if it becomes the right-most member, then only one connection will be required to connect it to the tree. Since we must also keep our tree AVL, we can keep skewing the tree until it reaches a state depicted in figure 3.11.

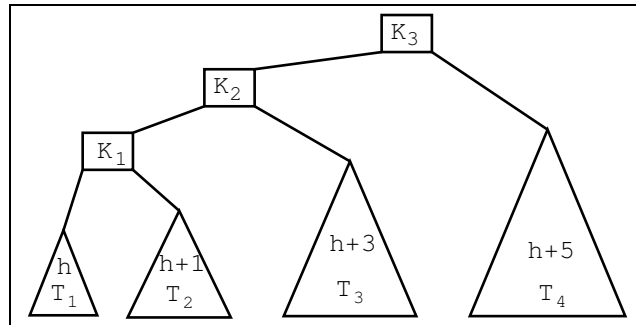


Figure 3.11: **A skewed tree. The tree is comprised of four subtrees: T_1, T_2, T_3 , and T_4 with heights $h, h + 1, h + 3$, and $h + 5$.**

When a member leaves, naively, $\log_2 n$ components should be merged together. This should cost $2 \log_2 n$ exponentiations. Examining the latency graph, and Figure 3.10(2), we can see that very few exponentiations take place. This is due to the caching optimization, and due to tree-skewing. To see how tree-skewing is beneficial, examine Figure 3.11. Assume a member of T_1 leaves, and keys K_1, K_2, K_3 must be reconstructed. The leader of T_1 needs three secure connections in order to disseminate new keys. Had the tree been fully balanced, the depth of T_1 would have been substantially larger than three, incurring the creation of additional channels.

Latency grows logarithmically as a function of group size (n). It does not grow monotonically because of the specific algorithm used to build the key-graphs. It can happen that the depth of 16-member key-tree is larger than an 18-member key-tree. This is cause of the non-monotonic behaviour of the latency graph.

Figure 3.12(1) describes the size of the multicast message describing the layout of the group-tree. Figure 3.12(2) depicts the size of the round three multicast message containing the set of encrypted keys. Figure 3.12(2) depicts the (average) total number of exponentiations. For the join case, this is a constant two, whereas for the leave case, this slowly rises.

To summarize, the protocol does not incur significant communication overhead, and the number of exponentiations is kept to a bare minimum. In some cases, the number of

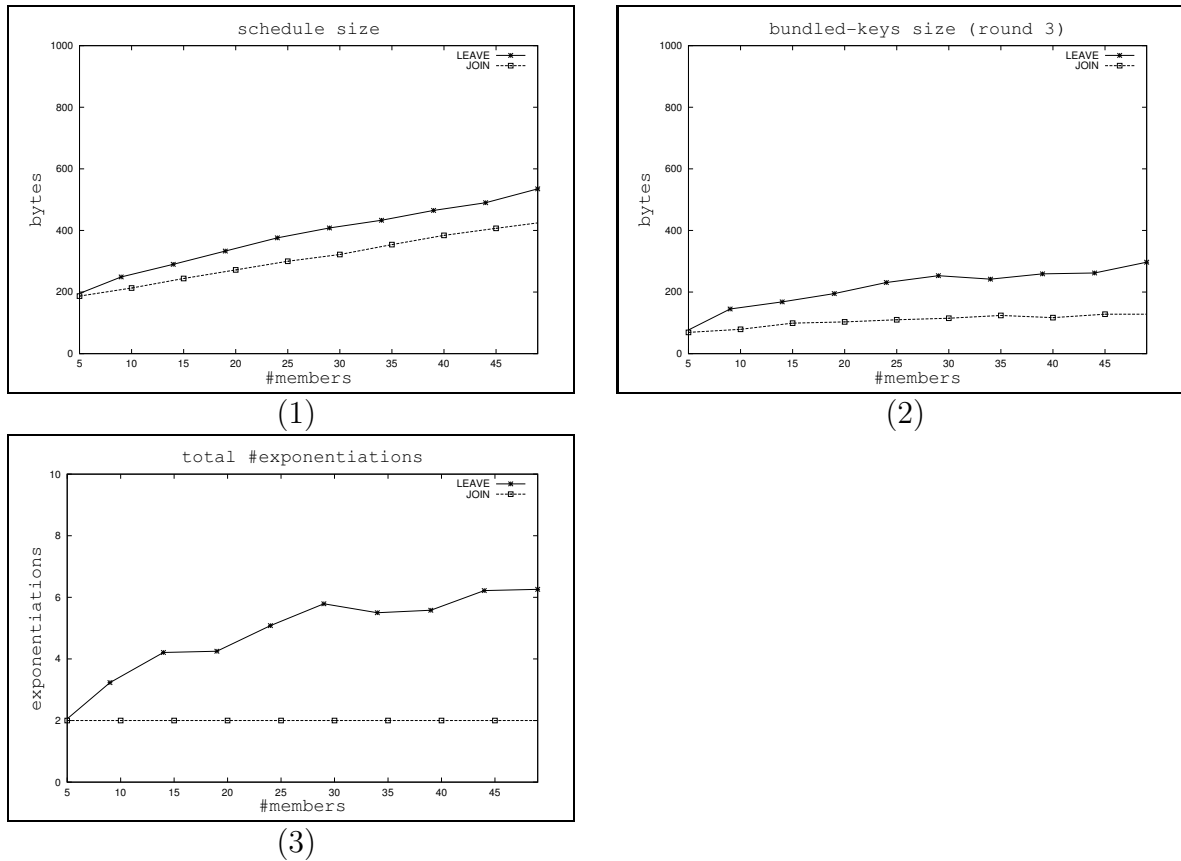


Figure 3.12: **Additional performance measurements.** (1) Size of the schedule sent by the leader. (2) Size of the bundled keys sent in the third round. (3) Total number of integer exponentiations.

exponentiations is in fact optimal.

3.6 Conclusions

We have described an efficient protocol for the management of group-keys for Group Communication Systems. Our protocol is based on the notion of key-graphs, or Logical Key Hierarchy, as originally suggested by Wong et al. and Wallner et al. We adapt and extend this work, making the protocol completely decentralized and fault-tolerant (to the extent possible), and employing a highly efficient tree balancing scheme.

In contrast, keygraphs were previously used in centralized settings, where a large

group of clients is managed by a central server. The server builds a keygraph encompassing all the clients, facilitating group-key management. In a GCS, where no single point of failure is allowed, we could not afford to delegate the server-task to any single member.

Therefore, our protocol differs substantially from the centralized approach. Rather, members enlist in a collaborative effort to create the group key-graph. The surprising result is that the completely distributed solution has comparative performance to the centralized one.

An issue for future work is supporting Backward Confidentiality efficiently. Currently, we support it in a weak sense.

Another problematic area is scalability to thousands of nodes. Our protocol relies on a Group Communication system, that can operate up to a hundred nodes. The protocol strongly relies on the agreed membership, and virtual synchrony properties of the GCS. It remains to be seen whether or not the protocol can be rewritten to use a weaker, yet more scalable type of infrastructure. For example, recent work on probabilistic failure tracking [GRB00] has yielded a scalable membership mechanism but with properties weaker than virtual synchrony. Applying our algorithm to such a scalable system is an interesting open question.

Chapter 4

Virtual Private Networks

4.1 Preface

In this chapter we describe an application relying strongly on a secure GCS: a Dynamic Virtual Private Network.

We say that a Virtual Private Network (VPN) exists between a set of machines if they can communicate over a public infrastructure with the illusion that the communications network has been dedicated for their exclusive use. VPNs provide an attractive solution to network problems¹.

At the time this work was undertaken, the VPN area was sparsely populated, with few interested companies, and few published academic papers. Today, there are various commercial solutions, and many companies offer firewall and VPN integrated solutions, Checkpoint [Che], Nortel [Rad], Cisco [Cis], Microsoft [Mic] to name a few. The networking community has also become interested [GLHA98, MM98, JJWB98]. Follow up work [DGG⁺99, RvCL98] has explored issues of shared resource management in a VPN.

VPNs are transparent (mostly) to the applications running inside them and they offer robust protection. However, they are limited in certain ways:

- Secure point-to-point connections are used to connect firewalls. This would be problematic if the set of firewalls were dynamic. For example, assume two sets of firewalls S_1 of size n and S_2 of size m ; $O(n^2)$ connections connect S_1 and $O(m^2)$ connect S_2 . To fully connect the sets together (for example, after a network partition) $n * m$

¹Currently, most commercial VPNs do not address QoS or denial service, thus, full network dedication is not provided.

connections must be set up. Similarly, If m firewalls are removed from the VPN then $(n - m) * m$ connections must be torn down.

- It is impossible to create several VPNs, and manage their interaction. Namely, multilevel security cannot be implemented.
- Autonomous management: when a VPN partitions into several disjoint components due to network fault, only a single component can continue to manage itself.

We propose *Dynamic Virtual Private Networks* as the next step in system security. A DVPN has the following properties, extending traditional VPNs:

- Decentralized management(fault tolerance): The DVPN does not have a single point of management. Should the DVPN split into several connected components each component is able to manage itself as well as retain capability of remerging with other components when network connectivity is reestablished.
- Scalable dynamic membership: machines may dynamically join and leave the DVPN. When trusted machines join the DVPN they automatically receive up-to-date security keys and other relevant information.
- Lightweight: A DVPN does not consume much machine resources, this allows new uses for DVPNs: running several DVPNs over a single machine, implementing multilevel security and more.
- Technology: We propose using a shared secret key to encrypt communication throughout the DVPN. We use a rigorous mechanism for securing the key's freshness and security such that only trusted and authenticated members receive it. The advantage over secure point-to-point connections is easier handling of dynamic addition and removal of firewalls. When machines are added to the DVPN, only the key need be passed. when a machine ceases to be trusted the DVPN key is simply switched — effectively removing it from the trusted network.

Since a DVPN incurs little overhead we propose using multiple overlapping DVPNs to implement fine grained security. For example, in Figure 4.1 we depict a company in which the CEO (C) is in the center, the manager group includes C , M_s , and M_p , the workers are W_s , W_{p1} , and W_{p2} . The managers form a group M in which they handle their discussions. There are also the Production, and Sales groups which use their private DVPNs for discussions. The CEO is naturally a member of all these groups. There is also a group that encompasses the whole company. DVPNs are used to confine information to authorized machines. Information should not flow from the managers to the workers; to

capture such restrictions we propose using an IP filter to enforce a policy for intra-DVPN communication. Such a filter, installed on all the machines in the company, will enforce a policy such as in Table 4.1.

DVPN	M	S	P
M	T	F	F
S	T	T	F
P	T	F	T

Table 4.1: **Example of a policy. Information may flow from M to M , from S to M and S , and from P to P and M .**

Such filtering can be performed efficiently, and it will prevent information from leaking from machines in M to less secure machines. When can a machine m be a member of DVPNs $D1$ and $D2$? When it is secure enough for both. To make this formal we attach a security level to each DVPN and each machine. Security levels are elements of a grid where the maximum and minimum operations are defined. Thus, a m may be a member of $D1$ and $D2$ only if its security level is greater or equal to $\max(D1, D2)$.

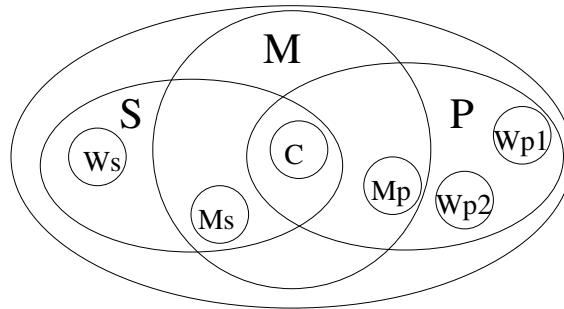


Figure 4.1: **The DVPN constellation in the company. C is the CEO; M_p and M_s are the lower level managers; W_s , W_{p1} , and W_{p2} are the workers. The groups are P for Production, S for Sales, and M for Management.**

Processes in a DVPN may need to use insecure services outside the protection scope of the DVPN. For example, the NFS server may not reside inside the DVPN. We propose encrypting all data stored on the NFS with a key known to all members of the DVPN. We do not explore this further in the paper except to note that using such unprotected services securely is formidable challenge.

We have built a system implementing a single DVPN, and have also dabbled with multiple DVPNs (though this has not been made freely available yet). Our work was performed on Linux, but should easily port to NT (on which our software could be used

to control the built-in NT VPN technology), Solaris, or other systems. Although we made some modifications to the operating system, these are so standard (a new device driver, some changes to the routing table, a packet filter) that they should be possible on any OS.

This chapter is organized as follows: Section 4.2 discusses today's VPN architectures, Section 4.3 describes the model and the assumptions that we make. Section 4.4 describes the security architecture that we are proposing, sections 4.5 — 4.9 describe its different components, scalability, and performance. Section 4.10 discusses future work and Section 4.11 concludes.

4.2 A standard VPN

This section describes a standard VPN. Normally, multiple VPNs are not used to secure an enterprise. Rather, a company's machines are encapsulated within a single VPN to protect them from malicious outsiders.

To build a VPN, the system administrator divides the user group into trusted and untrusted users. A machine is trusted if only trusted users can access it; otherwise, it is untrusted. An IP network is said to be secure if all the machines it contains are trusted. Thus, the global set of machines to be protected is partitioned into a disjoint set of secure IP networks, and distant clients.

The private IP subnets are sealed from the world by *firewalls* or *gateways*. These machines interpose on all communication paths between the trusted IP network and the external, untrusted, environment. The firewall employs a mixture of proxy mechanisms and packet filtering to protect the inside machines from intruders and also restricts access to the outside. For simplicity, henceforth we focus on VPNs running all their services on the firewalls. This is important, as we shall later see, since a DVPN essentially runs a lightweight firewall on all its members.

In Figure 4.2 an example of a set of machines in a VPN is shown. There are three trusted IP subnets, at Cornell, Berkeley, and the Hebrew universities, with firewalls at machines $F1$, $F2$, $F3$ and a trusted distant client $C1$. The adversary A is trying, illegally, to enter the VPN not through the firewall — his access is denied.

The firewalls are connected by a mesh of secure point-to-point links. To avoid the cost of leased lines, such connections are run over the Internet, using techniques such as PPTP [HPV⁺97] and IP_tunneling. Messages are encrypted and MAC-ed to ensure privacy and authenticity. Distant clients are connected to the VPN through one of the firewalls, using a secure connection.

VPNs are useful in many settings:

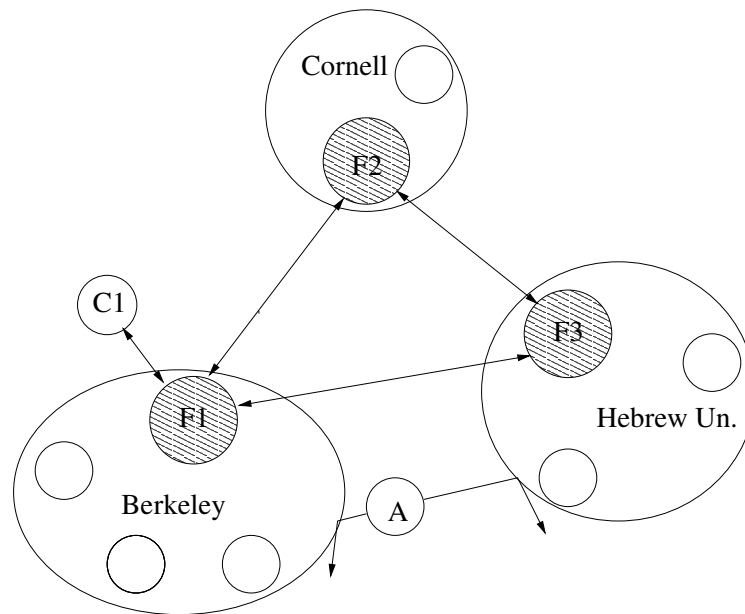


Figure 4.2: A typical VPN configuration.

- Virtual Organizations. The VPN connects hosts belonging to a single institution/company which is geographically spread over several distinct subnets. The organization thus avoids the need to lease dedicated lines, but has protection against Internet cyber-espionage.
- Connecting distant clients to a company's network. Increasingly, companies provide goods and services to their clients over the Internet, granting privileged access denied to non-clients. VPN solutions permit the company to define client networks such that only legitimate users can access network resources, and protecting valuable data against attack.
- Military and intelligence applications. A VPN creates a security containment zone, in which security policies limit information flow both into and out of the zone.
- So-called Community Health Information Networks (CHINs) are being developed in many states to link primary-care physicians and Health Management Organizations (HMOs) with local hospitals and specialized care providers. These networks may be connected directly to medical devices such as IV drips, and will carry sensitive data for which isolation and security are clearly needed.

The usual technology for ensuring the security of a point-to-point connection involves symmetric digital encryption and keyed-MAC. Data is encrypted to ensure its privacy; it

is MAC-ed to ensure its authenticity. Standard encryption schemes include DES [US 77] and IDEA [LMM91]; MAC schemes include MD5 [Riv92] and SHA [US 95]. All encryption and MAC operations require a shared secret key, such keys are agreed upon by two entities by using standard IKE [HC98] protocols.

Distribution of these shared keys is central to the correct functioning of a VPN. One must address key expiration, and periodically switch encryption keys. Also, the membership of the VPN is important so that only legitimate users receive up-to-date keys. Key revocation must be possible — when a user ceases to be trusted, his VPN access must be revoked. For example, if a computer is stolen from a laboratory, access from it to the CHIN should subsequently be denied.

A VPN must keep track of the set of up and running gateways. This can be performed by running an autonomous routing protocol between participating gateways, however, it is unclear whether this is currently implemented, or if this will be implemented in the future. A set of gateways, that can number 10s or 100s of machines, requires centralized management. Hence, gateways work with the Simple Network Management Protocol (SNMP) [HPW99], and can be managed remotely. The set of IP addresses owned by a VPN is hidden from the outside, this allows uninterrupted communication between members of the VPN, and hinders potential attacks. It does require translation between inner and outer hosts using NAT [SE01].

The big advantage of a VPN is transparency. Applications are typically oblivious to the fact that encryption and MAC are performed under the covers. This permits use of unmodified legacy software, and also follows good software engineering practices by separating security functionality from other aspects of the application.

4.3 Model

A VPN is said to exist between a set of machines if they can communicate without outside interference, or eavesdropping, and if they possess network services indistinguishable from private, dedicated network services. We distinguish between machines that are authorized to join a DVPN \mathcal{V} which we will refer to as the honest machines (in context of \mathcal{V}), and the remainder, potentially dishonest machines. We assume no Byzantine faults; Thus, all machines in a DVPN trust each other and mistrust all machines outside the DVPN. Once a machine is allowed into a DVPN, it is trusted until it leaves. Furthermore, a machine is trusted not to leak information received within the confines of a DVPN to the outside.

Our goal is to protect all message content sent between machines in a DVPN. We do not attempt to eliminate covert channels (such as timing channels), nor do we provide

protection against retransmission or denial of service attacks.

Security is provided on a machine granularity (since most off the shelf operating systems could readily be compromised by a knowledgeable user). We are thus vulnerable to any security problems inherent to the OS.

We consider only IP communication, and work with an existing IP stack, hence, we are exposed to attacks at the level of the stack itself (although our solution could be strengthened in this respect). For example, a “poison pill” attack might be able to crash a node on which our DVPN runs, and a flooding attack could saturate its interfaces. We assume the existence of a public key infrastructure and authentication mechanism that allows any machine to authenticate any other machine with complete certainty. This infrastructure, comprising of highly secure authentication and authorization servers, may also be used to pass authentic and confidential information between any pair of machines.

We assume that the adversary may listen to, modify, and retransmit any packet on the wire, it also has control over dishonest machines. However, the adversary does not gain access to honest machines. The adversary may try to crack (discover) encryption keys via search of the key space. With today’s technology a 56-bit DES key can be assumed intact for several hours under a very aggressive attack. A 128-bit IDEA key is virtually unbreakable at the time of this writing. The security concern is therefore key disclosure by some leakage.

Each machine p is assumed to have an IP address p_r in the regular IP address space. Virtual addresses for p are denoted by p_v for DVPN \mathcal{V} , p'_v for \mathcal{V}' , etc.

4.4 Our Solution

Our solution consists of four software modules: a loadable device driver and a packet filter, which reside in the kernel, two modified library procedures (**gethostname** and **gethostbyname**, that map machine names to IP addresses) and a management **mng** server that runs with super-user permissions on the host machine. Not all parts of the prototype have been built, yet enough has been accomplished to justify reporting on it. At present, we instantiate all components on every machine where the DVPN is used. We begin with a brief overview of the role of these components, then discuss some of them in greater detail. A schematic diagram is shown in Figure 4.3; The IP protocol stack is shown sliced according to the ISO layer model. The shaded areas are those which we have added. In the link layer we show the regular Ethernet driver and the DVPN driver. The network layer is composed of the IP layer and the IP filter; with a modified routing table. The transport and session layers have not been modified. Above these are the standard library functions of which we have modified **gethostname** and

gethostbyname. At the top are the applications, these remain unchanged, except for the addition of the **mngr** server.

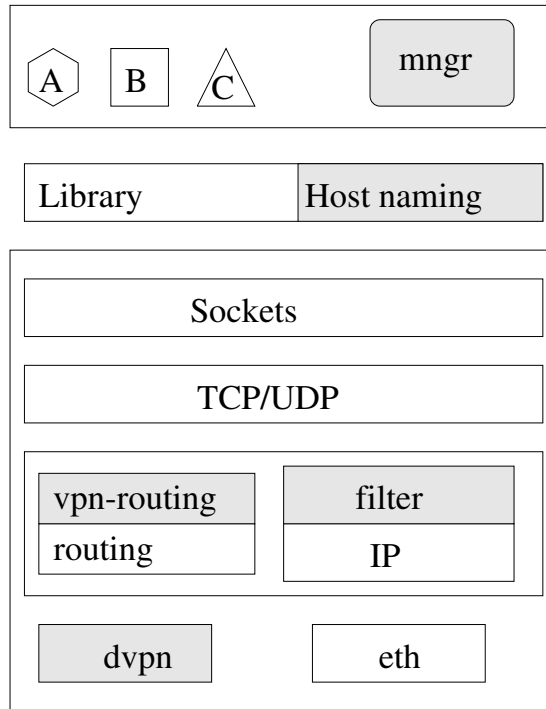


Figure 4.3: **Our security architecture, the shaded areas are those which we have added.**

Our DVPNs use a global service which we call the Available Address Service (AAS). This service is provided by a set of servers responsible for allocating virtual IP addresses within the global pool of available virtual IP addresses (see Section 4.7).

The life of a process can be summarized as follows. The process is first created on some machine, inheriting environmental attributes from the process that launched it. The process may look up the textual name of the machine on which it is running (by calling **gethostname**), and then map this name to its own IP address by calling **gethostbyname**. The process can also obtain the names of other processes from its runtime environment, the network information service, or (later) through interaction with other processes and services. These, too, it maps to IP addresses using **gethostbyname**. The process then communicates using IP services that operate using IP addresses, not plain-text names.

Our DVPN approach starts by arranging that each process is created within² some

²Work in Progress.

DVPN. More precisely, the login program on the machine is modified to join a DVPN at the time the user logs in. If the login is successful, a DVPN identifier is stored into the process context of the user’s command shell, and will be inherited by its child processes, which will pass it to their own children, and so forth.

We virtualize IP addresses and machine names. A machine may have multiple names — its regular DNS name and one for each DVPN to which it belongs. An IP address is associated with each name, a regular IP address for the real name and virtual addresses for the virtual names. For example: host `swift.cs.cornell.edu`, a member of DVPNs `sec1` and `sec2` will have names: `swift.cs.cornell.edu`, `sec1.swift.cs.cornell.edu`, `sec2.swift.cs.cornell.edu`. And corresponding IP addresses: 128.84.248.55, 200.200.200.1, and 200.200.250.3. The `gethostname` function has been modified so that if a process within DVPN \mathcal{V} (identifiable by the DVPN id associated with p in the process table) requests its name, then it gets the name associated with \mathcal{V} . For example, a process in DVPN `sec2` would get `sec2.swift.cs.cornell.edu`. The `gethostbyname` function has been extended to handle DNS names in the virtual networks, such names are not recognizable by the regular resolver.

Our packet filter intervenes on all IP communication pathways (functionality of this sort is standard in kernel-level packet filters). For packets sent by a user-level application in DVPN \mathcal{V} , the filter checks the source IP address and makes sure that it is correct. It must be the machine’s IP address in \mathcal{V} . The filter next examines the source and destination IP addresses as a pair. Messages sent within a single DVPN are passed to the IP routing layer. Currently, messages cannot be sent across DVPNs; thus, a process is only aware of other processes within its DVPN. This will be relaxed in the future: we plan to introduce mechanisms for controlled communication between a DVPN and the outside world, or between different DVPNs (see Section 4.10).

When we install a machine in a DVPN, we modify its IP routing table so that our DVPN driver will be shown as the interface to use for all outgoing/incoming IP packets in the DVPNs.

The role of our `mng` server is to manage a distributed database of information associated with each DVPN that the machine has joined. It is also responsible for acquiring an IP address in each DVPN it joins from the AAS. As membership of a DVPN changes, the routing table must be adjusted accordingly and the new membership data exported for use by `gethostbyname`. Additionally, the `mng` runs a protocol by which symmetric keys are generated, and periodically refreshed, for each DVPN. As keys are refreshed, the server installs them in our encapsulating driver by means of a system call.

The sections that follow give additional details for some of these components.

4.5 The driver

Our driver, D , is placed below the IP networking layer in the IP stack. We describe its functioning for a single DVPN \mathcal{V} . Each packet that travels through D is encapsulated within another IP packet. The new packet is then handed back to the IP layer which forwards it to its (possibly) new destination. The destination receives the packet, passes it to the IP layer which decapsulates it and hands to D . The driver then strips its own header and passes the inner packet back to the IP layer. It proceeds to deliver it to the application. Our driver never buffers packets. Packet size is limited according to the link technology used, for example, Ethernet allows no more than 1536 bytes per packet. The driver enlarges IP packets by attaching a new header, thus, it publishes the maximal packet size it can handle without fragmentation. The IP layer automatically fragments packets that are over that limit before handing them to D .

We configure the machine to use our driver by setting up the IP routing table so that \mathcal{V} is accessible through D only. Any IP packet sent to a virtual address will be forwarded through the driver. The driver uses a translation table by which it reroutes packets. A packet from p_v to q_v is sent with a header $p_r \rightarrow q_r$. The encapsulated packet specifies (in the protocol field of the IP header) a new protocol type notifying the receiver that this packet uses a special protocol. The receiver's IP networking layer will demultiplex the IP packet to D , as opposed to the UDP/TCP layers, or the ICMP daemon. Addresses in \mathcal{V} are not visible to the outside for two reasons: we do not publish them through regular IP services, and because they are accessible only via our special protocol and driver.

The packet is encrypted using IDEA, protecting its content from disclosure. Encryption does not guarantee authenticity, thus we also sign the packet with keyed-MD5. This flavor of MD5 starts the MD5 state with a secret key known only to DVPN members. Thus, unauthorized members cannot tamper with the packet signature. Currently, the same key is used for encryption and signature. In order for the receiving driver to know which key was used by the sender, all keys are uniquely numbered and the key number used to sign a packet is added to its header.

The header size used is 40 bytes long: 20 giving a normal IP address, and 20 split between 16 bytes for the MD5 hash, and 4 bytes giving the key number.

4.5.1 Key security

If an encryption key is used frequently over a long period of time, the potential of the key to be cracked or disclosed rises. Therefore, our DVPN switches keys rather frequently. However, this creates a technical challenge, because the network is asynchronous and we do not wish to stop communication while rekeying the DVPN. Suppose, for example, that

machines p and q are communicating using key x and it becomes necessary to change to key y . A situation could easily arise in which p possesses the new key and switches to it, but q has not yet received the new key and expects packets to be sealed with x . Machine q would now drop all packets from p , potentially disrupting communication for an extended period of time. Alternatively, p could delay use of key y until it is sure that q has received and switched to y , but now communication from p to q would need to be delayed during the period of uncertainty. For modern protocols, which are typically optimized to assume a low rate of IP packet loss, such disruptions can have serious performance impact. Even with a near simultaneous rekeying, a related issue would arise because packets may be arbitrarily delayed in the network. The TCP protocol, for example, assumes a delay bound of 2 minutes and long delays are often observed in wide-area settings. Ideally, we would prefer that the DVPN continue to accept such messages for a brief period. To address both issues, our driver keeps a backlog of n keys. We can choose n to be as large or as small as we wish. In order to switch keys smoothly the backlog must include at least the last key. When a key switch from x to y occurs, a driver starting to seal messages using y still keeps x in order to unseal messages sent by other drivers still using x . Currently we set n to 2.

4.6 The mngr

The **mngr** is the management daemon running on each machine in the DVPN. It is an application written using our Group Communication System (GCS) Ensemble. This section describes the role and inner workings of the **mngr**.

We can now describe the implementation of the server component of our DVPN architecture. At each machine in the DVPN, a server called **mngr** is started up as part of the initialization procedure. The servers join a secure process group that manages \mathcal{V} . This includes supplying the drivers throughout \mathcal{V} with identical translation and key tables. The translation table simply keeps a mapping between a group member its virtual IP address, real IP address, and real DNS name. This mapping is updated every view change. The key table simply keeps the current keys. A specific concern occurs upon a view change. To avoid communication disruptions we'd like to transition smoothly to a single key. We begin by merging all the key tables into a single table. After two minutes we switch to a new key that the (possibly new) leader distributes and erase the others.

The **mngr** also schedules the following events periodically:

Key Distribution: We rekey the DVPN by having the leader select a new (random) key which it securely broadcasts in the **mngr** group. In order to ensure that a key x will not be used until all members obtain it, we rely upon a two-phase-commit

protocol for each key update. The leader chooses x and broadcasts it to the group. All the members send a message acknowledging receipt. The leader then broadcasts a *Commit* message informing all members that they can begin sealing packets using x . If a membership change occurs during this protocol then the key is committed through the view-change/state-transfer protocol.

DVPN Sync: This operation erases all old keys, leaving only the newest key in place. When this process is done, only members in the DVPN know the current key. Thus, no old member can eavesdrop on current communication.

Mngr group rekey: Every few hours we refresh the Ensemble group key (in distinction to the DVPN key). We simply trigger the Ensemble Rekey protocol to perform this.

Normally, the above three operations are performed periodically by the **mngr**. However, the user may also request them at his discretion.

4.6.1 Key Lifetimes

The security of our solution depends upon a hierarchy of keys and assumptions. The least sensitive keys are also the ones used most heavily, namely the keys employed in the DVPN drivers. These are currently 128-bit symmetric IDEA keys. We rekey at this level once per minute, measured by the leader's clock. If a key is somehow compromised or cracked — the exposure is limited to data transmitted while that key was still in use. More sensitive are the keys used within the **mngr** group and managed using the Ensemble security protocol. These are also 128-bit IDEA keys; if one were broken, the intruder would be able to decrypt the driver-level keys distributed with it. If the current group key is broken the intruder would also be able to break the GCS by forging bogus messages and breaking communication protocols. Since these keys are used much less frequently and in a very restricted context, their lifetime can be longer, currently, several hours. Of course, if a machine leaves the DVPN or appears to fail, but may actually be at risk of compromise, it would be desirable to rekey promptly. Finally, we depend upon the public key infrastructure. Were a private RSA key to be compromised, this would compromise the past history of all groups that the corresponding machine was able to join. Future communication in those groups would be at risk until the next rekey event.

4.7 Available Address Service (AAS)

We create a global service to administer virtual IP addresses. This service is similar to the DNS in that it binds Internet names to IP addresses. However, it does not answer

naming queries. The local `gethostbyname` function receives such information through its local `mng` server.

We instantiate a server at all LANs in which machines use some DVPN. The AAS servers split amongst themselves the global pool of available IP addresses per DVPN. Each server has authority over its slice of the addresses. Each machine must receive an IP address from its local AAS for each DVPN it wishes to join. The AAS keeps track of the virtual addresses it has given out and the machines which have received them.

Currently, the mapping from text name to virtual address is static, but we intend to make them dynamic in the future. This could be useful in alleviating IP address shortage on virtual networks. Such an approach is already wide-spread in organizational networks, using DHCP [Dro97].

4.8 Performance

We performed several performance tests including continuous runs of the system for several days. The tests were run between two 200MHz PentiumPro machines connected through 100Mbit/sec Ethernet, the machines were situated on the same Ethernet segment. To perform the tests we used a *Virtual Network* driver (vn-driver); it is identical to the vpn-driver except that it creates virtual addresses without the added cost of encryption and signature. We also used an MD5 driver (md5-driver) which is the same as the VPN driver except that it does not perform encryption.

First we checked our machine's raw capacity for encryption and signature. The cost per-byte of IDEA encryption is $0.55\mu\text{sec}/\text{byte}$, for MD5 hashing it is $0.05\mu\text{sec}/\text{byte}$.

Next we executed ping between two machines to measure the latency achievable using our driver (Figure 4.4). We used varying packet sizes in multiplies of 1460 bytes (maximal packet size for our driver). The x -axis shows packet size in bytes, the y -axis shows latency in milliseconds. We see that the MD5 and vn-driver are fairly close to regular IP communication. On the other hand, the driver that uses IDEA is fairly costly. The difference in performance can be attributed solely to encryption cost. The cost per packet of size 1460 is the sum of encryption (sender) and decryption (receiver) costs:

$$2 * 0.55(\mu\text{sec}/\text{byte}) * 1460\text{byte} = 1606\mu\text{sec}$$

The measured difference is asymptotic to this number. We conclude that adding encapsulation and virtual addresses incurs a very low performance overhead. This makes our design suitable to provide message integrity; one can make a complete IP network strongly secure from tampering using our scheme. Since the cost of sending a packet

through the VPN driver is the cost of sending it through the regular IP stack, adding encryption, then the cost of using a DVPN is the cost of encryption. Using a high performance CPU or dedicated hardware should alleviate this cost, allowing applications to use DVPNs at nearly no cost.

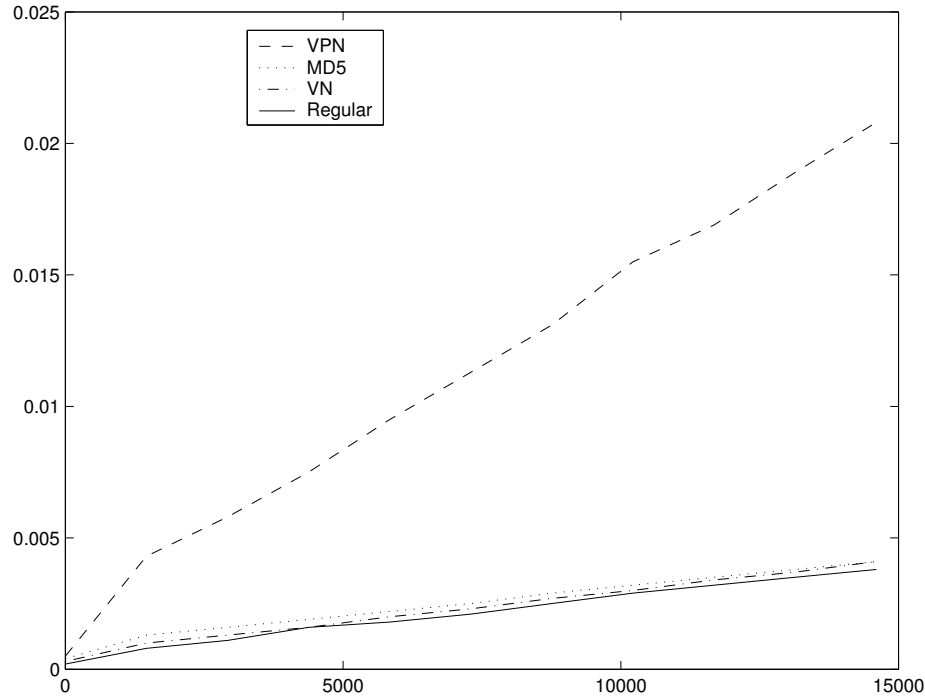


Figure 4.4: **Latency using: no driver, vn-driver, md5-driver, and a vpn-driver. The x -axis shows packet size in bytes, the y -axis shows latency in milliseconds.**

We went on to check achievable throughput. A TCP connection was opened between two machines and 10 megabytes of data were transmitted through it. We ran this several times and averaged. Table 4.2 shows throughput in *Mbyte/sec* in four cases: using the regular IP stack, vn-driver, md5-driver, and vpn-driver. We performed this test over a 10Mbit/sec Ethernet as well. In the 10Mbit/sec case³, using the vn-driver introduces insignificant cost while the vpn-driver reduced throughput to about 90%. On a 100Mbit/sec Ethernet the picture is very different. Using the vn-driver reduces throughput to 97%, the md5-driver to 81% and the vpn-driver to 17%. The bottleneck in this case is the CPU. Adding MD5 and IDEA adds to the CPU overhead and prevents it from processing incoming packets. There is no idle CPU time to use for encryption and signature.

³Our department switched to using 100Mbit/sec Ethernet while work was in progress; hence, we do not have complete measurements.

	Maximum	regular	vn-driver	md5-driver	vpn-driver
100Mbit/sec	12	8.823	8.568	7.114	1.486
10Mbit/sec	1.2	0.734	0.731	NA	0.666

Table 4.2: **TCP throughput in megabyte per second for 10Mbit/sec and 100Mbit/sec Ethernet. The columns show, from left to right, maximum achievable throughput, and then throughput using: no encapsulation, vn-driver, md5-driver, and vpn-driver.**

Next, we used a heterogenous network to check the performance of our protocols. The following measurement was performed with 11 machines, 3 200Mhz Pentium2 Linux machines, 5 Sparc10/20 Solaris 2.5, 1 SparcLX Solaris 2.5, and 2 SparcLX SunOS 4.1.3..

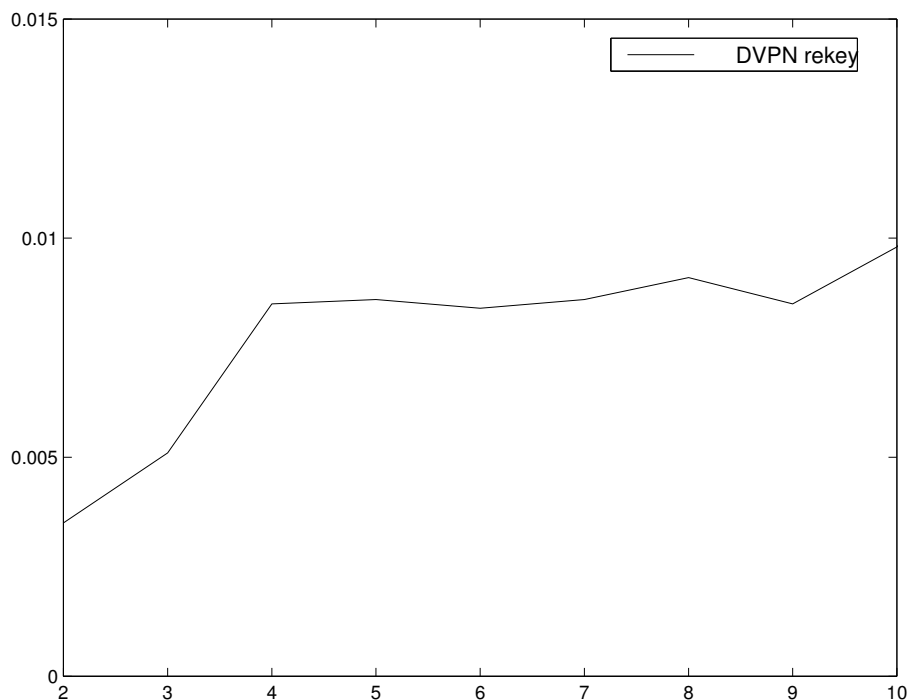


Figure 4.5: **DVPN rekey cost. The x -axis depicts the number of members in the group. The y -axis depicts latency in seconds.**

No attempt was made to optimize the Ensemble protocol stack, since the results obtained using a non-optimal version were sufficient for our purposes. In Figure 4.5 we see the cost of rekeying the DVPN. Recalling the protocol: the leader broadcasts a new key, the members acknowledge by a point-to-point message and the leader commits

the key via another broadcast. We measured the latency at the leader machine, the member machines still need another broadcast to complete the protocol. By multiplying the measured latency by a factor of 2 we can bound their latency. The latency for groups of up to 10 machines is less than 10 milliseconds which we believe to be more than adequate for our purposes. These experiments were performed just to show that Ensemble rekeying is not the bottleneck of the DVPN implementation, it was left for further work to establish this (chapters II and III).

4.9 Scalability

The Ensemble infrastructure on which we are building our solution currently scales to approximately 100 machines spread across a local area network. This allows **mngrs** on up to 100 machines to communicate securely and reliably. The question then is whether the *protocols* that we use are scalable to networks of this size.

The DVPN rekey protocol costs two broadcasts and n point-to-point messages. It's latency is 3 rounds of Ensemble communication. This protocol is the fastest and most heavily used. It takes less than 10 milliseconds to complete for 10 machines and can scale well.

In the state transfer protocol we merge the key and routing tables. All members send their state to the leader which merges and broadcasts the new state. This poses a total communication load of n point-to-point messages and one broadcast. The latency is 2 rounds of Ensemble communication. It is executed when a view change takes place; Ensemble takes less than a second to perform a view change, we add a fairly light overhead.

As the number of machines grows, we may have to make this protocol more scalable, this could be done using a tree structure to send the acknowledgments back to the leader.

We believe that our protocols scale, at least to a 100 machines in a LAN. We do believe that they can scale to much more than that, using a client/server architecture for Ensemble and some work on the data structure used to store the routing table. We will pursue this course in the future.

4.10 Future Work

IPsec includes all the encryption and signature options implemented by our driver. When it becomes available our encryption and signature code will no longer be of use. Instead, the driver will set for each outgoing packet the appropriate IPsec options.

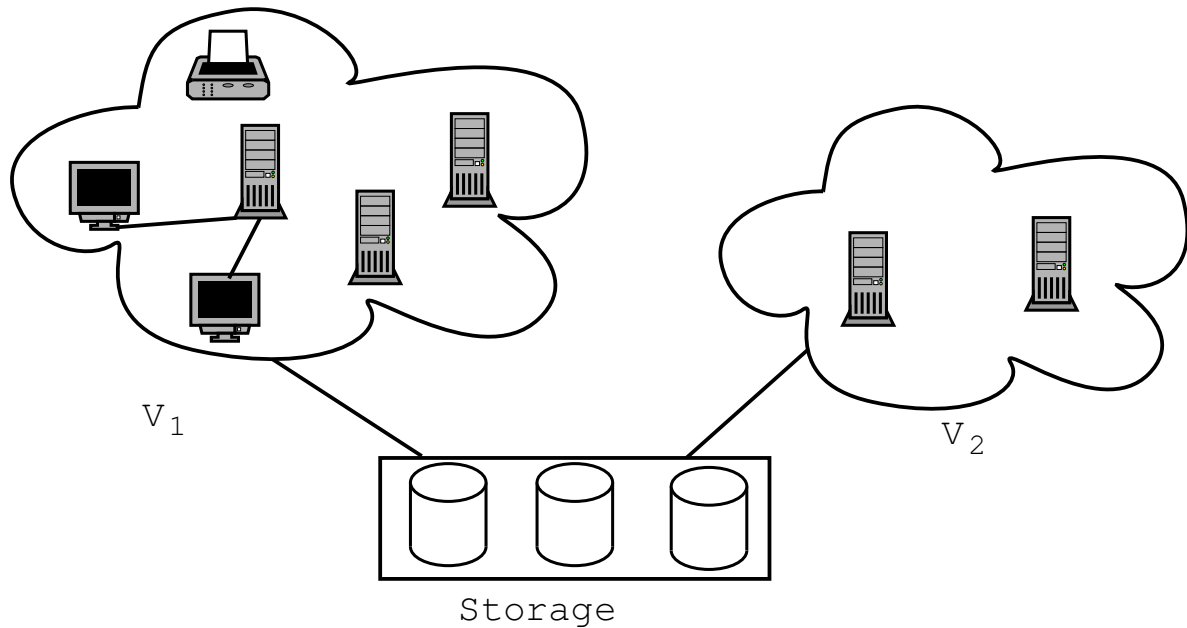


Figure 4.6: **A shared storage configuration: two DVPNs, V_1 and V_2 , share storage services.**

We have restricted applications in a DVPN to communicate only with other applications on the same DVPN. We intend to alleviate this restriction using a policy inside the packet filter. For example, a (weak approximation of) multilevel security architecture could be built using a concentric set of DVPNs using policies that allow applications to receive information from all less secure DVPNs, and send information up to any higher security DVPN.

Supporting access to insecure services from inside a DVPN requires care.

If a host uses a file system located in its DVPN, then stored data enjoys the same protection level as the rest of system. However, the file-system may be located on a remote storage server accessed by protocols such as NFS or CIFS. Assuming the storage server cannot be compromised by an intruder, it may be that more than one DVPN shares the same server. An example with two networks V_1, V_2 and shared storage is depicted in Figure 4.6. The problem is such a configuration is that the file system itself becomes a tunnel through which information can leak undetected. For example, if a database storing employee personal data is used by V_1 , then V_2 may also access the database and learn employee salaries etc. A DVPN is only as secure as its weakest link, here, storage becomes the Achilles heal.

If the storage server is highly trusted then it can be programmed to recognize traffic

arriving from different DVPNs, and tag files according to their origin. Consequentially, access to files can be restricted to their original DVPN only, or, more ambitiously, filtering rules could be applied, where V_1 may read files from V_2 , but not write them. Such rules should mimic the security levels governing the relationship between DVPNs. Work on a High-Assurance database that implements such features has been performed in [IARH98].

A different alternative is to encrypt all file content prior to storage. However, this technique is limited because it does not protect the directory structure itself, hence, V_2 can "peek" into V_1 's directory structure although it may not be able to read the actual files. The structure can reveal user login names, and access patterns according to access times and file length.

If a single storage server is not trusted enough, several can be used in concert. A file can be split into several pieces, where the combination of the pieces recreates the file. This may be achieved using secret sharing, or Error Correction Codes (ECC). Such techniques can be used to create $n - k$ schemes, where the file is split into n pieces, such that the combination of any k recreates the original file. Seminal work on storing files in several Object-Disks has been performed in the NASD project [WBS⁺00].

To use FTP or HTTP between hosts on two different DVPNs V_1 and V_2 , assuming V_2 is less secure than V_1 , requires an application level filter that allows protocol messages to flow in both directions, but content to move only from V_2 to V_1 . Later, configurations such as extranets can also be supported. An extranet is configuration used by a company, when it wishes to allow restricted access by business associates to some of its machines. The shared machines are isolated on a subnet separated from the world through a gateway (one or more) and separated from the company's core network by another gateway. The core network is protected and used exclusively by the organizations employees. In such situations, information is allowed to flow from the core network into the outer network and vice-versa but in carefully controlled ways.

4.11 Conclusions

We have presented a DVPN architecture which we believe to be significantly more flexible than traditional VPN systems. Unlike more standard VPN approaches, our solution dynamically rekeys the DVPN rapidly (as often as once every minute), and creates a self managing DVPN that can survive network faults and partitions. An initial implementation of our system already demonstrates that the approach is feasible, has low performance overhead, and can be carried out with minimal, rather standard, OS modifications.

Bibliography

- [AAH⁺00] Amir, Y., Ateniese, G., Hasse, D., Kim, Y., Nita-Rotaru, C., Schlossnagle, T., Schultz, J., Stanton, J., and Tsudik, G. Secure group communication in asynchronous networks with failures: Integration and experiments. In *International Conference on Distributed Computing Systems*, USA, April 2000. IEEE Computer Society Press.
- [ADKM92] Amir, Y., Dolev, D., Kramer, S., and Malki, D. Transis: A Communication Sub-System for High Availability. In *FTCS conference*, pages 76–84, USA, July 1992. IEEE Computer Society Press. <http://www.cs.huji.ac.il/~transis>.
- [AL62] Adel'son-Vel'skii, G. and Landis, E. An algorithm for the organization of information, 1962.
- [Bal96] Ballardie, A. Scalable multicast key distribution. Technical Report 1949, IETF, May 1996.
- [BCH97] Bruck, J., Cypher, R., and Ho, C.T. Fault-tolerant meshes with small degree. *SIAM Journal on Computing*, 26(6):1764–1784, 1997.
- [Bel90] Bellovin, S. M. Pseudo-network driver. In *Usenix*, January 1990.
- [BHO⁺99] Birman, K. P., Hayden, M., Ozkasap, O., Xiao, Z., Budiu, M., and Minsky, Y. Bimodal multicast. *ACM Transactions on Computer Systems*, 17(2):41–88, May 1999.
- [Bir99] Birman, K. P. A review of experiences with reliable multicast. *Software, Practice and Experience*, 29(9):741–774, Sept 1999.
- [BJ87] Birman, K. P. and Joseph, T. Exploiting virtual synchrony in distributed systems. In *11th ACM Symposium on Operating Systems Principles*, February 1987.

- [BMS99] Balenson, D., McGrew, D., and Sherman, A. Key management for large dynamic groups: One-way function trees and amortized initialization. Technical report, IETF, February 1999. draft-balenson-groupkeymgmt-oft-00.txt.
- [BR93] Bellare, M. and Rogaway, P. Entity authentication and key distribution. In *Crypto 93*, pages 232–249, USA, 1993. IEEE Computer Society Press.
- [BR94] Birman, K. and Renesse, R. V. *Reliable Distributed Computing with the Isis Toolkit*. IEEE Computer Society Press, USA, 1994.
- [CDB⁺98] Callas, J., Donnerhacke, L., Bellare, M., Finney, H., and Thayer, R. *OpenPGP Message Format*, July 1998. Internet Engineering Task Force, An Open Specification for Pretty Good Privacy (openpgp) Working Group.
- [CEH⁺00] Cox, M. J., Engelschall, R. S., Henson, S., Laurie, B., Young, E. A., and Hudson, T. J. Open SSL, 2000. <http://www.openssl.org>.
- [CGI⁺99] Canetti, R., Garay, J., Itkis, G., Micciancio, D., M. Naor, and B. Pinkas. Multicast security: A taxonomy and some efficient constructions. In *INFOCOM*, volume 2, pages 708–716, USA, March 1999. IEEE Computer Society Press.
- [Che] Checkpoint inc. <http://www.checkpoint.com/>.
- [Chi96] Chiakpo, E. *RS/6000 SP High Availability Infrastructure*. IBM, International Technical Support Organization, Poughkeepsie, USA, November 1996.
- [Cis] Cisco inc. <http://www.cisco.com/>.
- [CMN99] Canetti, R., Malkin, T., and Nissim, K. Efficient communication-storage tradeoffs for multicast encryption. In *Theory and Application of Cryptographic Techniques*, pages 459–474, Berlin, 1999. Springer-Verlag.
- [Dep85] Departement of Defence. Trusted computing base, 1985. Trusted Computer System Evaluation Criteria, DOD 5200.28-STD.
- [DGG⁺99] Duffield, N., Goyal, P., Greenberg, A., Ramakrishnan, K., and Merwe, J. V. D. A flexible model for resource management in virtual private networks. In *SIGCOMM*, August 1999.
- [DH76] Diffie, W. and Hellman, M. New directions in cryptography. *IEEE Transactions on information Theory*, IT-22:644–654, November 1976.

- [DMS95] Dolev, D., Malki, D., and Strong, R. A framework for partitionable membership service. Technical Report 95-4, Institute of Computer Science, The Hebrew University of Jerusalem, March 1995.
- [DPPU88] Dwork, C., Peleg, D., Pippinger, N., and Upfal, E. Fault tolerance in networks of bounded degree. *SIAM Journal on Computing*, pages 17:975–988, 1988.
- [Dro97] Droms, R. Dynamic host configuration protocol. RFC 2131, Bucknell University, 1997. IETF, Network Working Group.
- [EFL⁺99] Ellison, C., Frantz, B., Lampson, B., Rivest, R., Thomas, B., and Ylonen, T. Spki certificate theory. RFC 2693, IETF, Network Working Group, September 1999.
- [Ell99] Ellison, C. Spki requirements. RFC 2692, IETF, Network Working Group, Intel, September 1999.
- [FMG99] Friedman, R., Manor, S., and Guo, and K. Scalable stability detection using logical hypercube. October 1999.
- [GL99] Goft, G. and Lotem, E. Y. The AS/400 Cluster Engine: A case Study. In *International Workshop on Group Communication (IWGC'99)*, USA, September 1999. IEEE Computer Society Press.
- [GLHA98] Gleeson, B., Lin, A., Heinanen, J., and Armitage, G. A framework for ip based virtual private networks. internet-draft draft-gleeson-vpn-framework-00.txt, IETF, September 1998. Work in progress.
- [Gon97] Gong, L. Enclaves: Enabling secure collaboration over the internet. *IEEE Journal on Selected Areas in Communications*, 15(3):567–575, April 1997.
- [GRB00] Gupta, I., Renesse, R. V., and Birman, K. P. A pobabilistically correct leader election protocol for large groups. Tr, Department of Computer Science, University of Cornell, 2000.
- [GSSC96] Greenwald, M.B., Singhal, S.K., Stone, J.R., and Cheriton, D.R. Designing an Academic Firewall: Policy, Practice, and Experience With SURF. In *Symposium on Network and Distributed Systems Security*, February 1996.
- [Har62] Harary, F. The maximum connectivity of a graph, 1962.
- [Hay98] Hayden, M. The Ensemble System. Phd Thesis TR98-1662, Cornell University, Computer Science, 1998.

- [HC98] Harkins, D. and Carrel, D. The internet key exchange (ike). RFC 2409, IETF, november 1998.
- [HCBD99] Hardjono, T., Canetti, R., Baugher, M., and Dinsmore, P. Secure ip multicast: Problem areas, framework, and building blocks. Technical report, IRTF, October 1999. draft-irtf-smug-framework-00.txt.
- [HCBD00] Hardjono, T., Canetti, R., Baugher, M., and Dinsmore, P. Secure ip multicast: Problem areas, framework, and building blocks. Technical report, IRTF, September 2000. draft-irtf-smug-framework-01.txt.
- [HCD99a] Hardjono, T., Cain, B., and Doraswamy, N. A framework for group key management for multicast security. Technical Report draft-ietf-ipsec-gkmframework-01.txt, IETF, Network security subworking Group, 1999.
- [HCD99b] Hardjono, T., Cain, B., and Doraswamy, N. Intra-domain group key management protocol. Technical Report draft-ietf-ipsec-intragkm-01.txt, IETF, Network security subworking Group, 1999.
- [HCM01] Harney, H., Colegrove, A., and McDaniel, P. Principles of Policy in Secure Groups. In *Proceedings of Network and Distributed Systems Security*, Reston VA, USA, February 2001. Internet Society, Internet Society. San Diego.
- [HJSU00] Hiltunen, M. A., Jaiprakash, S., Schlichting, R.D., and Ugarte, C. A. Fine-grain configurability for secure communication. Technical Report TR00-05, Department of Computer Science, University of Arizona, June 2000.
- [HM97a] Harney, H. and Muckenhirn, C. Group key management protocol architecture. RFC 2094, IETF, 1997.
- [HM97b] Harney, H. and Muckenhirn, C. Group key management protocol specification. RFC 2093, IETF, 1997.
- [HPV⁺97] Hamzeh, K., Pall, G., Verthein, W., Taarud, J., and Little, W. Point-to-Point Tunneling Protocol. Internet draft, IETF, 1997. <http://www.ietf.org/internetdrafts/draft-ietf-pppext-pptp-02.txt>.
- [HPW99] Harrington, D., Presuhn, R., and Wijnen, B. An architecture for describing snmp management frameworks. RFC 2571, IETF, April 1999.
- [HS96] Hiltunen, M. A. and Schlichting, R. D. Adaptive distributed and fault-tolerant systems. *International Journal of Computer Systems Science and Engineering*, 11(5):125–133, September 1996.

- [HT99] Hardjono, T. and Tsudik, G. Ip multicast security: Issues and directions, 1999.
- [IARH98] Irvine, C.E., Anderson, J.P., Robb, D.A., and Hackerson, J. High assurance multilevel services for off-the-shelf workstation applications. In *National Information Systems Security Conference*, pages 421–431, October 1998.
- [IB93] Ioaniddis, J. and Blaze, M. The architecture and implementation of network layer security in unix. In *4th Usenix UNIX Security Symposium*, pages 29–39, Berkely, USA, 1993. Usenix association.
- [Jen96] Jenkins, R. J. Isaac. In *Fast Software Encryption, Third International Workshop*, pages 41–49, Berlin, 1996. Springer-Verlag. 1039 Lecture Notes in Computer Science (D. Gollman, ed.).
- [JJWB98] Jamieson, D., Jamoussi, B., Wright, G., and Beaubien, P. Mpls vpn architecture. Technical Report draft-jamieson-mpls-vpn00.txt, IETF, August 1998. Work in progress.
- [JMT98] Johnston, W., Mudumbai, S., and Thompson, M. Authorization and attribute certificates for widely distributed access control. In *IEEE 7th International Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises*, USA, 1998. IEEE Computer Society Press.
- [KBC97] Krawczyk, H., Bellare, M., and Canetti, R. Hmac: Keyed-hashing for message authentication. RFC 2104, IETF, February 1997.
- [KMM98] Kihlstrom, K.P., Moser, L.E., and Melliar-Smith, P.M. The securering protocols for securing group communication. In *Proceedings of the 31st Annual Hawaii International Conference on System Sciences (HICSS)*, volume 3, pages 317–326, USA, 1998. IEEE Computer Society Press.
- [KPT00] Kim, Y., Perrig, A., and Tsudik, G. Simple and fault-tolerant key agreement for dynamic collaborative groups. In *7th ACM Conference on Computer and Communication Security*, New York, USA, November 2000. ACM press.
- [KS95] Karn, P. and Simpson, W. The photuris session key management protocol. Technical Report 2522, IETF, 1995.
- [LMM91] Lai, X., Massey, J.L., and Murphy, S. Markov ciphers and differential cryptanalysis. In *Advances in Cryptology – EUROCRYPT*, Berlin, 1991. Springer.
- [Mic] Microsoft inc. <http://www.microsoft.com/>.

- [Mit97] Mittra, S. Iolus: A framework for scalable secure multicasting. In *SIGCOMM*, New York, USA, September 1997. ACM press.
- [MJ96a] McCanne, S. and Jacobson, V. The vat audio conferencing tool, 1996. <http://www-nrg.ee.lbl.gov/vat>.
- [MJ96b] McCanne, S. and Jacobson, V. The vic video conferencing system, 1996. <http://www-nrg.ee.lbl.gov/vic>.
- [MK99] Mukkamalla, S. and Katz, R. H. A scalable framework for secure multicast. Technical Report CSD-99-1049, Berkely university California USA, June 1999.
- [MM98] Muthukrishnan, K. and Malis, A. Core ip vpn architecture. internet-draft draft-muthukrishnan-corevpn-arch-00.txt, IETF, October 1998. Work in progress.
- [MMA⁺96] Moser, L. E., Melliar-Smith, P. M., Agarwal, D. A., Budhia, R. K., and Lingley-Papadopoulos, C. A. Totem: A fault-tolerant multicast group communication system. *Communications of the ACM*, 39(4):54–63, April 1996. homepage: <http://beta.ece.ucsb.edu/totem.html>.
- [MP00] McDaniel, P. and Prakash, A. Ismene: Provisioning and Policy Reconciliation in Secure Group Communication. Technical Report CSE-TR-438-00, Electrical Engineering and Computer Science, University of Michigan, December 2000.
- [MPH99] McDaniel, P. D., Prakash, A., and Honeyman, P. Antigone: A Flexible Framework for Secure Group Communication. In *Proceedings of the 8th USENIX Security Symposium*, Berkely USA, August 1999. Usenix society.
- [MRR99] Moyer, M. J., Rao, J.R., and Rohatgi, P. Maintaining balanced key trees for secure multicast. Technical report, IETF, June 1999. draft-irtf-smug-key-tree-balance-00.txt.
- [NT94] Neuman, B. C. and Ts'o, T. Kerberos: An authentication service for computer networks. *IEEE Communications*, 32(9):33–38, September 1994.
- [PB99] Poovendran, R. and Baras, J. S. An information theoretic analysis of rooted-tree based secure multicast key distribution schemes. In *CRYPTO*, pages 624–638, Berlin, 1999. Springer-Verlag.
- [Rad] Nortel inc. <http://www.nortel.com/>.

- [RBD00a] Rodeh, O., Birman, K. P., and Dolev, D. Optimized group rekey for group communication systems. In *Symposium on Network and Distributed System Security*, USA, February 2000. Internet Society.
- [RBD00b] Rodeh, O., Birman, K. P., and Dolev, D. A study of group rekeying. Technical Report TR2000-1791, Cornell University Computer Science, March 2000.
- [RBH⁺97] Renesse, R.V., Birman, K. P., Hayden, M., Vaysburd, A., and Karr, D. Building adaptive systems using ensemble. TR 97-1638, Cornell University, July 1997.
- [RBM96] Renesse, R.V., Birman, K.P., and Maffeis, S. Horus, a flexible group communication system. *Communications of the ACM*, 39(4):76–83, April 1996.
- [RBR94] Reiter, M.K., Birman, K.P., and Renesse, R.V. A security architecture for fault-tolerant systems. *ACM Transactions on Computer Systems*, 16(3):986–1009, November 1994.
- [Rei94] Reiter, M.K. Secure agreement protocols: Reliable and atomic group multicast in rampart. In *ACM Conference on Computer and Communication Security*, pages 68–80, New York, USA, November 1994. ACM press.
- [Riv92] Rivest, R. The md5 message digest algorithm. RFC 1321, SRI Network Information Center, April 1992.
- [RL96] Rivest, R. and Lampson, B. Sdsi—a simple distributed security infrastructure, 1996. <http://theory.lcs.mit.edu/~cis/sdsi.html>.
- [RSA78] Rivest, R., Shamir, A., and Adleman, L.M. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21(2):120–126, 1978.
- [RvCL98] Rooney, S., van, J. der Merwe, Crosby, S., and Leslie, I. The tempest, a framework for safe, resource assured, programmable networks. In *IEEE Communications Magazine*, volume 36, USA, 1998. IEEE Computer Society Press.
- [SE01] Srisuresh, P. and Egevang, K. Traditional ip network address translator (traditional nat). RFC 3022, IETF, January 2001.
- [SKJ00] Setia, S., Koussih, S., and Jajodia, S. Kronos: A scalable group re-keying approach for secure multicast. In *21th Symposium on Research in Security and Privacy*, USA, 2000. IEEE Computer Society Press.

- [STW98] Steiner, M., Tsudik, G., and Waidner, M. Cliques: A new approach to group key agreement. In *IEEE International Conference on Distributed Computing Systems (ICDCS'98)*, pages 380–387, USA, May 1998. IEEE Computer Society Press.
- [SYJS00] Shands, D., Yee, R., Jacobs, J., and Sebes, E. Secure virtual enclaves: Supporting coalition use of distributed application technologies. In *Network and Distributed System Security*, 2000.
- [TK97] Thayer, R. and Kaukonen, K. A stream cipher encryption algorithm. Internet draft, IETF, July 1997.
- [US 77] US Government. Data encryption standard. Technical Report 46, National Bureau of Standards, Federal, 1977.
- [US 95] US Department of Commerce, National Institute of Standards and Technology (NIST). Secure hash standard (shs), 1995.
- [VDB⁺98] Vogels, W., Dumitriu, D., Birman, K., Gamache, R., Short, R., Vert, J., Massa, M., Barrera, J., and Gray, J. The design and architecture of the microsoft cluster service – a practical approach to high-availability and scalability. In *28th Symposium on Fault-Tolerant Computing*, USA, June 1998. IEEE Computer Society Press.
- [WBS⁺00] Wylie, J.J., Bigrigg, M. W., Strunk, J. D., Ganger, G. R., Kilite, H., and Khosla, P.K. Survivable information storage systems. *IEEE Computer*, 33(8):61–68, 2000.
- [WGL98] Wong, C.K., Gouda, M., and Lam, S.S. Secure group communication using key graphs. In *SIGCOM*, New York, USA, September 1998. ACM press.
- [WHA98] Wallner, D., Harder, E., and Agee, R. Key management for multicast: Issues and architectures. Internet Draft draft-wallner-key-arch-01.txt, IETF, Network Working Group, September 1998. Work in progress.
- [WHA99] Wallner, D., Harder, E., and Agee, R. Key management for multicast: Issues and architectures. Technical Report 2627, IETF, Network security subworking Group, 1999.
- [Zim00] Zimmermann, P. Pretty good privacy, 2000. <http://www.pgp.com>.

Chapter 5

Appendix: K-diamonds

5.1 Introduction

This chapter describes the construction of graphs that can remain connected despite the loss of up to $k - 1$ nodes. This is a special case of Harary graphs [Har62]. Our graph construction supports efficient node addition, node deletion, and graph merge. There are two efficiency measures: complexity of computation, and edge complexity. The second measure quantifies the number of new edges that are constructed in each operation.

For the case of a single failure, our work from chapter 2 has shown that *diamond* graphs are an efficient construction. Here, we extend that construction to *k-Diamonds*. Throughout, we focus on the case of $k = 3$, since the construction is basically the same for larger values of k . We abbreviate k-Diamonds as simply diamonds.

Our construction uses a recursive data structure with two variants:

- A single member
- A diamond such that:
 - There are three root nodes
 - A diamond of depth > 2 must have between 3 and 5 sub-diamonds. A diamond of depth two may have between 1 and 5 children.
 - All sub-diamonds are connected to all roots (with different links).
 - AVL : All sub-diamonds have heights between h and $h + 1$

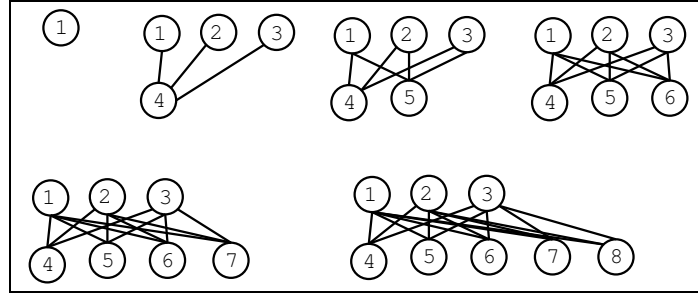


Figure 5.1: **The basic 3-diamond structures. These are examples for up to 8 members.**

A 3-diamond, with at least 6 members, is resilient to two faults. In what follows A and B will stand for Diamonds, and $h(X)$ will denote the height of diamond X .

First, we show some simple examples of 3-diamonds, see Figure 5.1 for diamonds of sizes one through eight. Larger diamonds can be constructed by the *node-addition* operation. See examples in Figure 5.2. Boxes denote diamonds of size up to eight, the top three members are the roots, and lower members are connected to all roots. Boxes are used throughout to reduce the number of drawn edges.

The node addition procedure takes an existing diamond A and a new member m . It first attempts to add m in an empty spot in A . For example, Figure 5.2 shows how members are added one at a time to a diamond of size four. At a certain point, the diamond must be rebalanced and increased in height to accommodate new members. Diagram (1a), depicts a depth two diamond with six children that is rebalanced to a depth three diamond. Similarly, diagrams (2a) and (3a) depict a depth three diamond with 6 immediate children, rebalanced to spread the children in a more balanced fashion. We use a single transformation for rebalancing.

The general rebalancing scheme is that when a diamond D has three roots and more than five children, then three nodes are “stolen” and assigned the task of being the roots of a new, larger diamond. Of the remaining children (sub-diamonds), the two largest ones are moved to be children of the top-level roots, to make the diamond more balanced.

The *steal* primitive locates and strips *free* members from D . A free member is one whose removal will leave a diamond intact. Examples of free members are, in Figure 5.1 members five, six, seven, and eight. If there are no free members, then a more drastic measure is taken. Since D now has only sub-diamonds of depth two with four members, the deepest one, C , is decomposed. C has three root nodes, and a single child. The root nodes are striped away, and the child is left in place of C . Since C is the deepest sub-diamond, the AVL constraint as well as the rest of the constraints, can be maintained.

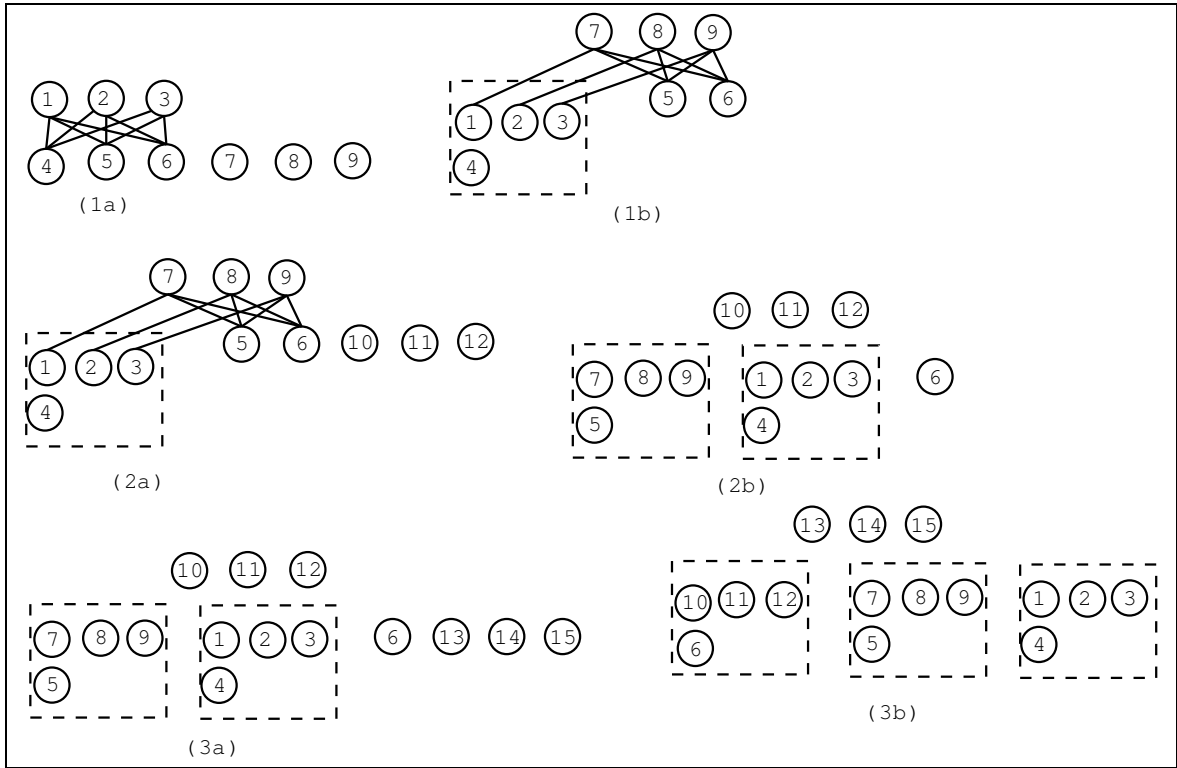


Figure 5.2: **3-diamond structures for up to 15 members. Boxes are used to describe diamonds of sizes less than 8. The top three members are the roots, and lower members are all connected by three links to all roots. Diagrams (1a,2a,3a) shows an unbalanced diamond, that must be balanced, forming the corresponding diagrams (1b,2b,3b).**

Two of the root nodes are attached to D as free members, and the last member is returned. An example for stealing three members is shown in Figure 5.3.

Two cases of a more general nature are presented in Figure 5.4. There are other cases, where sub-diamonds are shrunk because of stolen members etc., such cases are similar to those depicted.

The computational complexity of the rebalance step is $O(n)$, because the tree is searched for free members. The number of new edges to construct is split between: 1) stealing three members: the maximal cost is that of breaking a sub-diamond of depth two, and the return of two members to D . This amounts to $3(k + 1)$ new edges. 2) The rebalance step itself: connecting the three new root nodes to the three new children costs $3(k + 1)$ edges. All in all — $O(k)$ new edges.

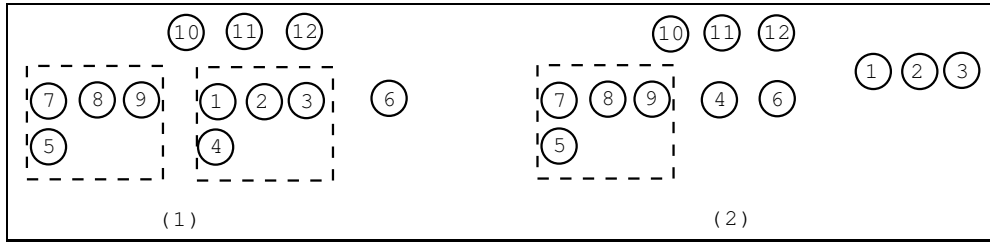


Figure 5.3: **Stealing three members.** Members one, two, and three are stolen from the diamond.

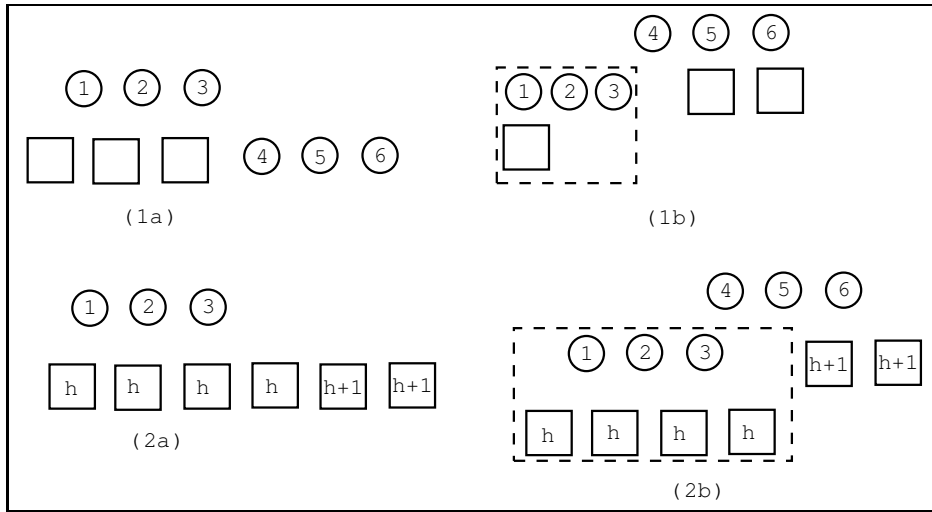


Figure 5.4: **Two cases of rebalancing.** (1a) There are three free members that can easily be “stolen”. (2a) Members four, five, and six are stolen, moved to the root position, and the larger sub-diamonds are moved one level up.

The computational complexity of node-addition is in the normal case $O(n)$ (to search for an empty spot), and $k + 1$ edges to connect the new member to D . If a rebalance step is required, then its cost is added to the normal case cost. This sums up to $O(n)$ time and $O(k)$ edges.

5.2 Merge

This section describes how two 3-diamonds, A and B , are merged together at low cost.

There are two cases:

- $h(A) = h(B)$: we mark the height of both A and B as h . The children of A and B may be of heights $h - 1$ and $h - 2$. First, we exchange the children, such that A holds all children of height h , and B holds all children of height $h - 1$. Now there are two cases:
 - $h(A) = h$ and $h(B) = h - 1$, in which case we make B a child of A . Since A may now have 6 children, a rebalance step may be required.
 - $h(A) = h$ and $h(B) = h$, but all of A 's children are of height $h - 1$. Hence, B can be a child of A , and as above, a rebalance step may be required.
- W.l.o.g $h(A) > h(B)$: we now find a sub-diamond A' of A such that $1 \geq h(A') - h(B) \geq 0$. A' is chosen such that it is in the smallest possible sub-diamond, so that adding B will not mess up the AVL requirement. We now proceed to add B as a child of A' . If this creates more than 6 children for A' , then we use the rebalance step described above.

To compute the cost of merging two diamonds A and B , we split it into two cases:

- $h(A) = h(B)$: exchanging children ($O(k)$ time) and a re-balance step — $O(k)$ edges and $O(n)$ time.
- $h(A) < h(B)$: search for a sub-diamond A' + re-balance step.

In both cases, the cost is bounded by $O(k)$ edges and $O(n)$ time.

5.3 Leave

If a single member m leaves diamond D , then several cases are possible:

- The diamond now has less than six nodes, in which case there is nothing to do.
- If there is a free member in D , then strip it and place it in m 's place.
- If there are no free members in D , then decompose the deepest sub-diamond, leave a single node in its place, and use the free members to fill in m 's place. Spread the rest of the members throughout D .

This scheme clearly has computational complexity $O(n)$, and edge complexity $O(k)$.