High Performance, Robust and Secure Group Communication

Yair Amir
Department of Computer Science,
Johns Hopkins University

Final Report

Prepared for DARPA
under contract F30602-00-2-0526

February 2004

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. REPORT DATE 15-02-2004

2. REPORT TYPE Final Report

3. DATES COVERED

4. TITLE AND SUBTITLE

High Performance, Robust and Secure Group Communication

5a. CONTRACT NUMBER
F30602-00-2-0526

5b. GRANT NUMBER

5c. PROGRAM ELEMENT NUMBER

6. AUTHOR(S)

Yair Amir

5d. PROJECT NUMBER

5e. TASK NUMBER

5f. WORK UNIT NUMBER

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

Johns Hopkins University
3400 N. Charles Street
Computer Science Department
224 NEB
Baltimore, MD 21218

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)

Col. Tim Gibson
DARPA/ATO
3701 North Fairfax Drive
Arlington, VA 22203

Mr. Jon Valente
AFRL/IFGA
525 Brooks Rd.
Rome, NY 13441

10. SPONSOR/MONITOR'S ACRONYM(S)

11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION / AVAILABILITY STATEMENT

13. SUPPLEMENTARY NOTES

14. ABSTRACT

We demonstrate how security techniques can be integrated into group communication systems, while maintaining a high level of performance. We propose an architecture for secure group communication, relying on a group key agreement protocol that is efficient, robust to process crashes and network partitions and merges and protects confidentiality of past data even when long term keys of the participants are compromised. We show how different group communication semantics can be supported in the proposed architecture, discuss the accompanying trust issues and present experimental results that offer insight into its scalability and practicality.

15. SUBJECT TERMS

Secure Group Communication, robust key agreement, secure middleware

16. SECURITY CLASSIFICATION OF:			**17. LIMITATION OF ABSTRACT**	**18. NUMBER OF PAGES**	**19a. NAME OF RESPONSIBLE PERSON**
					Yair Amir
a. REPORT	**b. ABSTRACT**	**c. THIS PAGE**		135	**19b. TELEPHONE NUMBER** 410-516-4803

Standard Form 298 (Rev. 8-98)
Prescribed by ANSI Std. 239.18

Abstract

Distributed applications increasingly rely on messaging systems to provide secure, uninterrupted service within acceptable throughput and latency parameters. This is difficult to guarantee in a complex network environment that is susceptible to a multitude of human or electronic threats. Security is a critical component of the survivability of such distributed messaging systems that operate in a dynamic network environment and communicate over insecure networks such as the Internet.

We demonstrate how security techniques can be integrated into group communication systems, while maintaining a high level of performance. We propose an architecture for secure group communication, relying on a group key agreement protocol that is efficient, robust to process crashes and network partitions and merges and protects confidentiality of the data even when long term keys of the participants are compromised. We show how different group communication semantics can be supported in the proposed architecture, discuss the accompanying trust issues and present experimental results that offer insight into its scalability and practicality.

Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1

Summary

Distributed applications increasingly rely on messaging systems to provide secure, uninterrupted service within acceptable throughput and latency parameters. This is difficult to guarantee in a complex network environment that is susceptible to a multitude of human or electronic threats. Security is a critical component of the survivability of such distributed messaging systems that operate in a dynamic network environment and communicate over insecure networks such as the Internet.

We demonstrate how security techniques can be integrated into group communication systems, a particular case of distributed messaging systems, while maintaining a high level of performance. Many security services (data secrecy, data integrity, entity authentication, etc) can be bootstrapped if members of the group share a common secret, which makes key management a critical building block. However, designing key management protocols that are robust and efficient in the presence of network and process faults is a big challenge. We propose an architecture for secure group communication, relying on a group key management protocol that is efficient, robust to process crashes and network partitions and merges and protects confidentiality of the data even when long term keys of the participants are compromised. Our technical approach builds on our work with the Spread group communication system (http://www.spread.org) and the CLIQUES key agreement protocols suite (http://sconce.ics.uci.edu/cliques/) and resulted in Secure Spread (http://www.cnds.jhu.edu/securespread), a secure group communication system. Current key agreement protocols are not designed to tolerate failures and changes in the membership during their execution. In contrast, Secure Spread uses as building block our protocols, which are completely resilient to any sequence of such events. We believe this is the first

robust implementation of distributed key agreement protocols that provide perfect forward secrecy, group membership authentication, non-repudiation, and resilience to known-key attacks.

The current state of the art in secure group communication implements security as a layer, separate from the reliability, ordering, and membership services. Although this structure has much merit, there is a high performance cost attached. We provide two basic approaches to integrate security into a group communication system. The Layered Architecture version places the security services on top of the reliability, ordering and membership services. The Integrated Architecture version tailors the security protocols into the core reliability, ordering and membership services, drastically cutting the latency and bandwidth cost associated with group membership changes.

We show how different group communication semantics can be supported in the proposed architecture, discuss the accompanying trust issues and present experimental results that offer insight into its scalability and practicality.

Chapter 2

Introduction

Ubiquitous information access and communication have become essential to everyday life, global business, and national security. Activities including personal and multinational financial transactions, studying and teaching, shopping for goods (such as books, cars, software and even groceries), or managing modern battlefields have fundamentally changed over the last decade as a result of the expanding capabilities of computers and networks. Most such activities are in fact supported by distributed applications which in turn increasingly rely on messaging systems to provide secure, uninterrupted service within acceptable throughput and latency parameters. This is difficult to guarantee in a complex network environment that is susceptible to a multitude of human or electronic threats, especially as network attacks have become more sophisticated and harder to contain.

A distributed messaging system is an abstraction layer that is built on top of an underlying network and provides distributed applications with services not available from the native network (for example built-in security, ordered message delivery, etc) or with improved services (for example higher availability, improved reliable delivery, etc). Group communication systems, overlay networks, and middleware are examples of messaging systems serving as infrastructure for applications such as web clusters, replicated databases, scalable chat services and streaming video.

In the context when many applications are expected to run over the Internet, the need for security in computing and communication became a necessity. We note that also for applications running in local area networks, particularly in commercial environments, security is required to ensure restricted access to data and to protect communication according to regulations and hierarchical structures specific to a company. Although not an

independent service, security is an enabling feature without which the actual end-services cannot be trusted or relied upon. To this end, the research community has invested a lot of effort in investigating and developing effective and efficient security services. Numerous algorithms, protocols, frameworks and policy languages have been developed to provide security services in point-to-point or group-based communication models. However, there has not been much research into the integration of security techniques into distributed systems, while maintaining a reasonable level of performance.

This work focuses on providing security services for a particular type of distributed messaging systems, group communications systems. More precisely, it presents how security services can be integrated into group communication systems without sacrificing high-availability and performance. The challenge lies in understanding the advantages and limitations of security protocols when integrated with group communication systems and the effect this interaction has on the overall fault-tolerance and performance of the system.

The rest of the chapter is organized as follows. We first introduce group communication systems and briefly overview basic cryptographic mechanisms we used in designing the security services for our secure group communication system. Since a critical building block of a secure group communication system is group key management, we concentrate on key management protocols. We then specify the focus and the contributions of this work. We continue with overviewing related work in the areas of group key management protocols and secure group communications systems and describing the roadmap of the report. We end by presenting in detail Spread, the group communication system which is the focus of this work and Cliques, the key management cryptographic library that was used as developing tool in designing our secure group communication system.

2.1 Group Communication Systems

Group communication systems are distributed messaging systems that enable efficient communication between a set of processes logically organized in groups and communicating via multicast in an asynchronous environment. More specifically they provide two services: group membership and reliable and ordered message delivery. The membership service provides all members of a group with information about the list of current connected and alive group members (also referred as a view) and notifies the members about every group change. A group can potentially change because of several reasons. In a fault-free

network, the group change can be caused by members voluntarily joining or leaving the group. However, faults can happen, for example processes can get disconnected or crash, or network partitions can prevent members from communicating. When faults are healed, group members can communicate again. All the above events can also trigger changes in the group membership.

The reliable and ordered message delivery includes several services. The basic ordering service is FIFO that guarantees that messages are delivered to recipients in the order they were sent by the sender. Stronger ordering services are CAUSAL that ensures that messages are delivered to destination in causal order and AGREED that delivers message in total order. The strongest service is SAFE delivery, that provides both ordering and reliability guarantees, messages are delivered in total order and delivered to recipients unless they crash. More details about these services are provided in Section 3.2.

Group communication systems are strongly connected with fault-tolerance. The strong services provided by group communication systems made them appealing to be used as developing tool or infrastructure for numerous fault-tolerant applications ranging from the classical replications applications to the more recent fault-tolerant CORBA. Examples of applications that can take advantage of group communication systems include:

- replication using a variant of the state machine/active replication approach (1; 2), such as (3; 4; 5; 6; 7);
- primary-backup replication (8);
- distributed transactions and database replication (9; 10; 11; 12; 13);
- distributed and clustered operating systems (14; 15; 16);
- collaborative applications such as collaborative computing (17; 18) (19), distance learning (20), video and audio conferences (21), application sharing (22);
- resource allocation (23; 24) and load balancing (25; 26);
- system management (27), cluster management (28) monitoring (29) and distributed logging (30);
- highly available servers: management (31), (32) video on demand servers (33);
- real-time applications (34);

- provision of object group services within CORBA (Electra (35), Eternal (36) and the Object Group Service (37).

The core mechanism of group communication systems is achieving agreement about group membership views and about the order of delivering messages, between multiple participants, communicating in an asynchronous environment with failures. However, many agreement protocols were proved to have no solution in asynchronous systems with failures (38). Practical group communication systems overcome the problem by using time-out based failure detection to detect network connectivity and process faults. The risk of such an approach is that alive and connected members communicating on links that suffer from high delay, can be excluded from the group membership. If the network is stable, group communication systems reflect the current list of connected and alive group members.

The membership and reliable and ordered message delivery services were formalized in two main group communication models: Virtual Synchrony (39) and Extended Virtual Synchrony (40). The main difference between the models comes from the relation between the views in which messages are sent and delivered. We discuss this aspect in details in Chapter 3.

Group communication systems have been built around a number of different architectural models, such as peer-to-peer libraries, 2- or 3-level middle-ware hierarchies, modular protocol stacks, and client-server. To improve performance, modern group communication system use a client server architecture where the expensive distributed protocols are run between a set of servers, providing services to numerous clients. In this architecture the client membership service is implemented as a "light-weight" layer that communicates with a "heavy-weigh" layer asynchronously using a FIFO buffer. In such a model, an application using the group communication system as infrastructure, will link with the group communication client library in order to get access to the membership and ordering and reliable message delivery provided by the servers.

Security is crucial for distributed and collaborative applications that operate in a dynamic network environment and communicate over insecure networks such as the Internet. Basic security services needed in such a dynamic peer group setting are largely the same as in point-to-point communication. The minimal set of security services that should be provided by any group communication system include: client authentication and access control as well as group key management, data integrity and confidentiality. More specifically:

- *Client authentication.* Any client should be authenticated when it requests access to the group communication system, e.g., when it connects to a group communication system server.

- *Access control.* It is not enough that a client was authenticated, he also must be authorized to access system resources. Typical group communication resources are: joining or leaving a group and sending messages to a group.

- *Group key management.* Group key management protocols specify how members of a group can compute a shared group key and how that key is refreshed. For example the key can change when the list of group members changes, or it can change based on a time-out or data-out indicators. The shared key can be used to bootstrap other group services, i.e., data integrity and confidentiality. These services cannot be attained without secure, efficient and fault-tolerant group key management.

- *Integrity and confidentiality.* All the communication between group members should be protected both from eavesdropping and undetected modification.

2.2 Cryptographic Mechanisms

Security services or goals are in general achieved by employing a set of cryptographic mechanisms. The set of cryptographic primitives used to provide security services are referred in cryptography as *cryptosystems* (41). There are two main ways of designing cryptosystems. One approach, is based on the assumption that the participants share a common secret (key). In this case, the secret key is used both for encryption and decryption. This is referred as *symmetric cryptography* and cryptosystems using it are known as *symmetric cryptosystems.* Another approach uses two different keys: one for encryption and the other decryption, one of the key is public and the other is secret. The two keys are mathematically related. This approach is referred as *public cryptography* and cryptosystems design based on this concept are referred as *asymmetric cryptosystems.*

In general public keys are bind to their owners by using a Public Key Infrastructure (PKI). A PKI not only that provides means to link a specific public key with its owner, but also allows distribution of these keys in large networks.

The main components of a PKI are:

- *Public Key Certificate*: It represents an electronic record that binds a public key to the owner of the public key. It also digitally binds the certificate to the entity that issued it and specifies limits to the validity of the certificate.

- *Certification Authority (CA)*: It is an entity that issues and revokes public key certificates.

- *Certificate Revocation List (CRL)*: It represent a list of certificates that have been revoked.

Algorithms employing symmetric cryptography are usually about 3-4 orders of magnitude faster than algorithms employing public cryptography, which makes them more appealing for encryptions data-streams. Although more expensive, public cryptography techniques have the advantage that they do not require any secret to be shared between parties. A common application for public encryption is to distribute secret keys that can then be used to encrypt data using algorithms based on symmetric cryptography.

In this section we give a high-level description of the main cryptographic techniques we used to design secure group communication systems: group key management, encryption algorithms, message authentication codes and digital signatures.

2.2.1 Group Key Management

As discussed above many efficient security services rely of the fact that participants share a common secret. The process whereby a shared secret becomes available to two or more parties is known as key establishment. An important aspect with respect to the shared key, is how it is replaced, when necessary. This is achieved by key management protocols that in addition to key establishment, provide maintenance of ongoing keying relationships between parties, including replacing older keys with newer keys. In the context of groups, the shared secret is referred as *group key* and the protocols that define how such a key can be computed and how is it periodically refreshed, is referred in the literature as *group key management protocols*.

There are several desirable security properties for key management protocols.

1. *Group Key Secrecy*. This property guarantees that it is computationally infeasible for a passive adversary to discover any group key.

2. *Forward Secrecy.* This property guarantees that a passive adversary who knows a contiguous subset of old group keys cannot discover subsequent group keys.

3. *Backward Secrecy.* This property guarantees that a passive adversary who knows a contiguous subset group keys cannot discover preceding group keys.

4. *Key Independence.* This is a strong property that guarantees that a passive adversary who knows a proper subset of group keys cannot discover any other group key. It includes the previous three properties.

5. *Perfect Forward Secrecy.* This is a strong property that specifies that even when long-term key of participant get compromised, the secrecy of past group keys is preserved.

There are two important aspects of any key management protocol. One defines how the key is selected or computes (key selection), the other one specifies how the participants obtain the key (key distribution). Traditional centralized key management relies on a single fixed key server to generates and distributes keys to the group. This approach does not work correctly for group communication systems that guarantee continuous operation in any possible group subset and any arbitrary number of partitions in the event of network partitions of faults. Although a server can be made constantly available and attack-resistant with the aid of various fault-tolerance and replication techniques, it is very difficult (in a scalable and efficient manner) to make a centralized server present in every possible group subset. We note that centralized approaches work well in one-to-many multicast scenarios since a key server (or a set thereof) placed at, or very near, the source of communication can support continued operation within an arbitrary partition as long as it includes the source.

The requirement of providing continued operation in an arbitrary partition can be overcome by dynamically selecting a group member to act as a group key server. However, most centralized key distribution protocols do not provide strong security properties such as key independence and perfect forward secrecy because they encrypt new group keys using old group keys using symmetric encryption, or they encrypt group keys with long term group keys in which case by breaking the long term key an attacker can get access to all session group keys.

The above strong properties can only be provided if the key server maintains pair-wise secure channels with each group member in order to distribute group keys and refreshes

them every time it needs to distribute a new group key. Although this approach seems appealing, each time a new key server comes into play, significant costs must be incurred to set up pairwise secure channels. In addition, this method has a minor disadvantage (common with the centralized fixed server case) in that it relies on a single entity to generate good (i.e., cryptographically strong, random) keys.

Another approach is to use a fully distributed contributory group key management where a group key is not selected by one entity, but, instead, is a function of each group member's contribution. This avoids the issues with centralized trust, single point of failure (and attack) and the requirement to establish pairwise secret channels, and provides strong security properties such as forward and backward secrecy, key independence and perfect forward secrecy (41).

2.2.2 Encryption Algorithms

One important security concern in network communication is confidentiality or secrecy of the data. Encryption algorithms were design to address this problem. As mentioned before, there are two main ways of designing encryption algorithms: one using symmetric cryptography and the other using public cryptography. For efficiency reasons, the preferred method is symmetric cryptography which in turn requires the involved parties to share a common secret.

There are two main classes of encryption algorithms: block ciphers and stream ciphers. A block cipher is an encryption scheme which breaks up the plaintext messages to be sent over the network into strings of fixed length, and encrypts one block at a time. Examples of block ciphers include DES (42), Triple-DES (42), Skipjack (43), Blowfish (44), AES (45). Because block cipher algorithms define how to map a block of a fixed length, when used to encrypt messages of variable and longer size, block cipher algorithms are used in encryption modes of operation. Encryption modes specify what is the relation between the encrypted blocks and subsequent data and encrypted data. The four most common modes are ECB, CBC, CFB and OFB, (46; 47). For a detailed description of the modes and discussion of the advantages and disadvantages of each of them, the reader is referred to (41).

In addition to providing confidentiality, block cipher algorithms can be used as fundamental building block for construction of other cryptographic primitives such as pseudo-

random number generators, stream ciphers, message authentication codes and hash functions.

Another important class of encryption algorithms are stream ciphers. They encrypt the characters of a message one at a time, using an encryption transformation which varies with time. The concept behind stream ciphers is to simulate the one-time pad which is unconditionally secure regardless of the statistical distribution of the plaintext. In such a scheme the symmetric key used to encrypt must have a length equal to the length of the data being encrypted and used only once. Examples of stream ciphers include RC4 (48) and SEAL (49). As oppose to block cipher protocols, stream ciphers are faster. However, they were less used in practice because of proprietary claims, while many block cipher implementations were in the public domain.

2.2.3 Message Authentication Codes and Digital Signatures

An important security service besides confidentiality is data integrity which ensures that a message was not modified between source and destination. The simplest method to achieve this is by having the source to compute a one-way function (hash) on the message and send it together with the message to the destination. Hash function are not invertible and therefore it is very difficult for an attacker to be able to recover the message from the result of applying the hash function. When the destination receives the message, it computes the hash on the received message and compares it with the received hash. If the two hash value are different then the message was modified by a potential attacker and should be rejected. Otherwise, the integrity of the message from source to destination was preserved. Examples of hash functions include: MD5 (50) and SHA1 (51).

When receiving a message, it is very important for the recipient to be able to authenticate the source of the message: is it coming from a trusted party or not. The standard way to do this is by using a shared key, which implies that whoever had the key was able to generate the result. A largely used scheme is called HMAC (52). It provides both message integrity by using a one-way hash function and source authentication by relying on a secret key between the communicating parties.

A different method to provide data source authentication is by using digital signatures. Digital signatures are a direct application of public encryption cryptography: the sender encrypts the message with the private secret key ("signs the message"), and the

destination decrypts the message by using the public key ("verifies the message"). When combined with a hash method(instead of signing the message, the hash of the message is signed), digital signatures provide both data integrity and authentication.

In addition to source authentication, digital signatures provide non-repudiation of the source of a message because the secret key used to generate the digital signature is associated with only one entity. However, because of their increased cost (about 3 to 4 orders of magnitude more expensive than symmetric cryptography based schemes), if non-repudiation is not a required service, it is desirable to use a method like HMAC, instead of digital signatures. Examples of digital signatures schemes include RSA (53) and DSS (54). We note that digital signatures schemes that require a small time to verify the signature (for example RSA) are more appropriate for systems where communication is achieved via multicast.

Digital signatures are also used to bind a public key certificate to the CA that issued it. A CA digitally signs the content of any public key certificate it issued. Another entity can accept a public key certificate, if it was signed by a CA it trusts and the signature verifies.

2.3 Focus and Contribution

This work focuses on designing high-performance and efficient secure architectures for group communication systems. Since many group communication systems are built around a client-server architecture where a relatively small number of servers provide group communication services to numerous clients, for performance and scalability reasons, we focus on a system using this architecture. More precisely, we focus on securing Spread (55), a local and wide area group communication system.

Secure, robust and efficient key management is a critical building block for secure group communication. However, designing key management protocols – that are robust and efficient in the presence of network and process faults – is a big challenge. An important contribution of this work is showing how multi-round group key agreement protocols can be made fault-tolerant, by using the membership and reliable and ordered message delivery services of group communication systems.

Key management protocols have a big impact not only on the fault-tolerance of the system, but also on the performance, since they can become a bottleneck of the system.

As part of this work we analyze both theoretically and experimentally four well-known key agreement protocols and compare them with a centralized key distribution protocol adapted to provide the same security properties: key independence and perfect forward secrecy. We believe that the contributions of this work are valuable to both group communication and security research communities. The actual costs associated with group key management have been poorly understood in the past. Consequently, there has been a dual undesirable tendency: on the one hand, adopting suboptimal security for reliable group communication, while, on the other hand, constructing excessively costly group key management protocols.

We distinguish among two basic approaches to integrate security services into a client-server group communication system. The first approach, referred to as the *layered architecture*, places security services in a client library layered on top of the group communication system client library. The second approach entails housing some (or all) security services at the servers in order to obtain a more scalable design, referred to as the *integrated architecture*. We show how both such architectures can be achieved for Spread and provide a comparison and experimental results that offer insights into their scalability and practicality.

More specifically, the contributions of this work are:

1. We design a secure and robust group communication service by combining contributory group key agreement protocols with a reliable group communication service supporting Virtual Synchrony semantics. Specifically , we present two robust contributory key agreement protocols (basic and optimized) that are resilient to any *finite*, potentially cascaded, sequence of events. The protocols are based on the Group Diffie-Hellman (GDH) (56) key agreement. We prove that our protocols preserve the Virtual Synchrony group communication properties and does not break the security properties of the Group Diffie-Hellman key agreement protocol.

2. We provide an insight into the costs of adding security services to group communication systems, focusing on group key management, by presenting a theoretical analysis of four notable group key agreement methods with respect to communication and computation costs and comparing them with a centralized protocol modified to provide similar security properties (e.g. key independence and perfect forward secrecy). In addition, we provide an in-depth experimental evaluation of these protocols implemented in the same framework. We present results obtained from live experiments

with various types of group membership changes (join, leave, partition and merges) over both local and wide area networks. These results provide valuable insights into the protocols' scalability and practicality.

3. We present a high-performance security architecture for Spread, under two well-known group communication semantics: Virtual Synchrony (39) and Extended Virtual Synchrony (40). Both models support network partitions and merges. Our approach entails using contributory group key management in a light-weight/heavy-weight (5) group architecture such that the cost of key management is amortized over many groups, while each group has its own unique key. We propose three variants of an integrated architecture that trade off encryption cost for complexity and group communication model support. We discuss their performance and security guarantees and compare them to a layered approach, demonstrating the increased scalability.

2.4 Organization of the Report

Chapter 2 introduces group communications systems and cryptographic mechanisms used to provide security services. Then it layouts the focus and contributions of our work and presents a survey of notable prior work in the areas of key management protocols and secure group communication systems. Finally, it describes Spread, the group communication system that is the focus of this work and Cliques, the key management cryptographic library used in the implementation of our secure group communication system.

Chapter 3 describes the failure model and the group communication semantics assumed for this work.

Chapter 4 focuses on fault-tolerant group key agreement, by demonstrating how provably secure, multi-round group key agreement protocols can be combined with reliable group communication services to obtain provably **fault-tolerant** group key agreement solutions. More precisely, it presents two robust contributory key agreement protocols which are resilient to any sequence (even cascaded) of events while preserving group communications membership and ordering guarantees. Both protocols are based on the Group Diffie-Hellman contributory key agreement protocol that generalizes on the two-party Diffie-Hellman (57)

key exchange and uses the services of a group communication system supporting Virtual Synchrony semantics.

Chapter 5 analyses five popular key management protocols for collaborative peer groups, integrated with a reliable group communication system They are: Centralized Group Key Distribution (CKD), Burmester-Desmedt (BD), Steer et al. (STR), Group Diffie-Hellman (GDH) and Tree-Based Group Diffie-Hellman (TGDH). The chapter describes the framework used to experiment with the protocols, a layered architecture for Spread. It then provides an in-depth comparison and analysis of the five protocols based on experimental results obtained in real-life local and wide area networks, for all possible groups events, including partitions and merges. The analysis of the protocols' experimentally measured performance offers insights into their scalability and practicality.

Chapter 6 presents several integrated security architectures for Spread, a client-server group communication systems. The chapter discusses the benefits and drawbacks of each proposed architecture and compares them with a layered architecture. It then presents experimental results that demonstrate the superior scalability of an integrated architecture.

Chapter 7 concludes this work and summarizes the contributions of this work.

2.5 Related Work

There are two major directions in secure group communication research. The first one aims to provide security services for IP-Multicast and reliable IP-Multicast like groups. These type of groups are very large and are managed centrally; they assume one sender and many receivers. Typically, one-to-many settings only aim to offer continued operation within a single partition that includes the source, and is less concerned with providing a strong group membership service. Since the presence of a shared secret can be used as a foundation of efficiently providing data confidentiality and data integrity, a lot of work has been done in designing very scalable key management protocols.

The second major direction in secure group communication research is securing application level multicast systems, also known as group communication systems. These systems assume a many-to-many communication model where each member of the group

can be both a receiver and a sender, and provide reliability, strong message ordering and group membership guarantees, with moderate scalability. These systems typically provide continuous operation in the presence of faults, network partitions and merges, making the group key management more difficult that in single-source groups because of the availability and robustness requirements. Work done in securing group communication systems goes beyond key management, since these systems aim to provide security services such as confidentiality, integrity and resilience to different type of attacks on the membership protocol.

In this chapter we consider related work in two areas: group key management and secure group communication systems.

2.5.1 Group Key Management

Cryptographic techniques for securing all types of multicast or group-based protocols require all parties to share a common key. Group key management protocols generally fall into two classes:

- Protocols designed for large-scale (e.g., IP Multicast) applications with a one-to-many communication paradigm and relatively weak security requirements (58; 59; 60). Most of these protocols are centralized key distribution schemes, one or a few centralized authorities distribute keys to all members of the group.

- Protocols designed to support tightly-coupled dynamic peer groups with medium scalability requirements, a many-to-many communication paradigm and strong security requirements (61; 56). Both distributed group key distribution and contributory group key agreement methods were designed in such settings.

Many protocols of the first type are being developed in the context of the GSEC IRTF research group and MSEC IETF working group. Examples of such protocols are: the Group Key Management Protocol (GKMP) (59), Multicast Key Management Protocol (MKMP) (62), the Scalable Multicast Key Distribution (SMKD) (63) approach based on the Core Based Trees (58), Intra-domain Group Key Management Protocol (IGKMP) (60), the VersaKey Framework (64), Logical Key Hierarchy Protocol (LKH) (65), One-way Function Trees (66), Group Secure Association Key Management Protocol (GSAKMP)(67), GSAKMP-light (68), Group Domain of Interpretation (GDOI) (69), while (70) defines an

architecture for large scale group key management. Since the focus of our work is on dynamic peer groups key management, we discuss only distributed group key distribution and contributory key agreement protocols.

Most of the group key agreement schemes (56; 71; 61; 72; 73; 74) extend the well-known Diffie-Hellman key exchange (57) method to groups of n parties.

Steer et al. proposed a group key agreement protocol (71) for static conferencing. While the protocol is well-suited for adding new group members as it takes only two rounds and two modular exponentiations, is relatively expensive when excluding members.

In 1994, Burmester and Desmedt (61) proposed an efficient key management protocol. The goal was to minimize the amount of computation that each group member must perform. It takes three modular exponentiations per member to generate a group key allows all members to re-compute the group key for any membership change with a constant small CPU cost. However, this distribution of the computation effort is achieved by using $2n$ simultaneous broadcast messages. The communication overhead can become significant on wide area networks.

More recently Tzeng and Tzeng designed an authenticated key agreement scheme based on secure multi-party computation (73). Their protocol is optimized in terms of the number of the communication rounds, but still uses $2n$ simultaneous broadcast messages. Although its cryptographic mechanism is elegant, the resulting group key does not provide perfect forward secrecy (PFS) which represents a major drawback.

Steiner et al. address dynamic membership issues (56) in group key agreement and propose a family of Group Diffie-Hellman (GDH) protocols based on straight-forward extensions of the two-party Diffie-Hellman protocol. Their protocol suite is fairly efficient in leave and partition operation, but the merge protocol requires as many rounds as the number of new members to complete the key refresh protocol. All of the protocols in the suite scale linearly with the group size in the number of serial exponentiations required to change the group key. The entire protocol suite has been proven secure with respect to both passive and active attacks (75; 76).

Follow-on work focused on providing more efficient protocols by using tree structures. In (72), the efficiency is achieved in computation, the protocol scaling logarithmically with the group size in the number of exponentiations, while in (74) the focus shifts on minimizing the communication.

Dynamic group key distribution methods are also amenable to dynamic peer groups. CKD is a very simple example of distributed key distribution, where the oldest group member acts as a key distribution center and in the event of a partition or a leave of the center, the role shifts to the oldest remaining member. Rodeh et al. proposed more advanced key distribution protocols, combining a key tree structure with dynamic key server election (77) or taking advantage of efficient data structures such as AVL trees (78). Although the communication and computation costs are appreciably lower than those in CKD, the protocol does not provide forward secrecy, key independence and perfect forward secrecy.

2.5.2 Secure Group Communication Systems

Research in group communication systems operating in a local area network (LAN) environment has been quite active in the last 15-20 years. Initially, high availability and fault tolerance were the main goals. This resulted in systems like ISIS (79), Transis (80), Horus (81), Totem (82), and RMP (83). These systems explored several different models of group communication such as Virtual Synchrony (39) and Extended Virtual Synchrony (40). More recent work in this area focuses on scaling group membership to wide area networks (WAN) (84; 85).

With the increased use of group communication systems over insecure open networks, some research interest shifted to securing group communication systems. Research on securing group communication is fairly new. The only implementations of group communication systems that focus on security (in addition to ours) are the SecureRing (86) project at UCSB, the Horus/Ensemble work at Cornell (78; 87) and the Rampart system at AT&T (88).

The core of any group communication system is a membership protocol. Some of the work in securing group communication focused on protecting the membership protocol in the presence of Byzantine faults. This includes systems such as Rampart (88) and SecureRing (86). Rampart builds its group multicast over a secure group membership protocol achieved by the means of two-party secure channels. The SecureRing system protects the low-level ring protocol by using digital signatures to authenticate each transmission of the token and each data message received. Both systems exhibit limited performance since they use costly protocols and make use intensively of public key cryptography.

In addition to the membership service, group communication systems provide reliable ordered message delivery within a group. To make this secure, group members (senders) must be authenticated and confidentiality and integrity of client data must be guaranteed. One notable result in this area is the Horus/Ensemble work at Cornell (89; 78; 87). Ensemble achieves data confidentiality by using a shared group key obtained by means of group key distribution protocols. Although efficient, the scheme does not provide forward secrecy, key independence and perfect forward secrecy. Ensemble uses as authentication the popular PGP (90) method. In addition, the system allows application-dependent trust models in the form of access control lists which are treated as replicated data within a group. Recent research on Bimodal-Multicast, Gossip-based protocols (91) and the Spinglass system has largely focused on increasing the scalability and stability of reliable group communication services in more hostile environments such as wide-area and lossy networks by providing probabilistic guarantees about delivery, reliability, and membership.

Some other approaches focus on building highly configurable dynamic distributed protocols. Cactus (92) is a framework that allows the implementation of configurable protocols as composition of micro-protocols. Survivability of the security services is enhanced by using redundancy for specific security services. For example, in (93), redundancy of data confidentiality is obtained by encrypting data multiple times, each time using a different encryption algorithm. This approach is not appropriate for data-stream applications where throughput is a concern.

Another toolkit that can be used to build secure group oriented applications is Enclaves (94). It provides group control and communication (both point-to-point and multicast) and data confidentiality using a shared key. The group utilizes a centralized key distribution scheme where a member of the group (group leader) selects a new key every time the group changes and securely distributes it to all members of the group. The main drawback of the system is that it does not address failure recovery when the leader of the group fails.

A collaborative application can have its own specific security requirements and its own security policy. The Antigone policy (95) framework allows flexible application-level group security policies in a more relaxed model than the one usually provided by group communication systems. Policy flavors addressed by Antigone include: re-keying, membership awareness, process failure and access control. The system implements group rekeying mechanisms in two flavors: session rekeying - all group members receive a new

key, and session key distribution - the session leader transmits an existing session key. Both schemes present some problems: distributing the same key when the group changes breaks perfect-forward-secrecy, while the session rekeying mechanism although can detect the leader's failure, can not recover from it.

Unlike the aforementioned systems, our approach focuses on the use of contributory key agreement as a building block for other security services in Spread (55). Contributory key agreement protocols provide strong security properties, they guarantees that the compromise of any subset of group keys does not lead to the compromise of a group key and even that the compromise of a long-term secret key will not lead to the compromise of any group key. Our work investigates the inter-relation between key agreement and group communication system. We designed and implemented the first robust contributory group key agreement protocol. Moreover, our secure group communication system supports strong group communication semantics: Virtual Synchrony and Extended Virtual Synchrony.

2.6 Spread Group Communication System

Spread (55) is a general-purpose group communication system for wide- and local-area networks. It supports a many-to-many communication paradigm where any group member can be both a sender and a receiver. Although designed to support small- to medium-size groups, it can accommodate a large number of collaboration sessions, each spanning the Internet. Spread scales well with the number of groups used by the application without imposing any overhead on network routers.

The main services provided by the system are reliable and ordered delivery of messages (FIFO, causal, total order, safe) and a membership service in a model that considers benign network and computer faults (crashes, recoveries, partitions and merges).

The system consists of a server and a client library that an application using the group communication systems as a communication infrastructure must link with. The client library provides an API interface that allows a client to connect/disconnect to a server, to join and leave a group, and to send and receive messages.

The client and server memberships follow the model of light-weight and heavy-weight groups (96). This architecture amortizes the cost of expensive distributed protocols, since these protocols are executed by a relatively small number of servers, as opposed to having all clients participating. Another advantage of this architecture is that a simple join

or a leave of a client process translates into a single message, instead of a full-fledged membership change. Only network partitions [1] incur the heavy cost of a full-fledged membership change.

The servers maintain the groups membership, data flow control and the reliability and ordering of messages. Because all the above services are managed by the servers and not at the individual process group level, they are more efficient.

A client can be a member of more than one group. The global orderings and reliability (Causal, Agreed, Safe) are provided across all groups in the system. If two messages are sent by different clients to different groups, any client who has joined both groups will receive the two messages according to the requested ordering service, even though they are received in different groups.

The toolkit is highly configurable, allowing the user to tailor it to their needs. Spread can be configured to use just one server in the world or to use one server on every machine running group communication applications. The best performance when there are no faults is achieved when a daemon is available on every machine, while using a smaller number of servers decreases the cost of recovery.

Spread supports two well-known group communication semantics, Virtual Synchrony (VS) (5; 97) and Extended Virtual Synchrony (EVS) (40; 4) (see (98) for a comprehensive survey of group communication models). The VS service is provided by a client library implemented on top of the library providing EVS semantics.

Both group semantics guarantee that group members define that group members will see the same set of messages between two sequential group membership events and that the order of messages requested by the application is preserved. They also guarantee that all messages are delivered in the same view. However, there is a major difference in this last aspect: while VS guarantees that messages are delivered to all recipients in the same view as the sending application thought it was a member of at the time it sent the message (also known as Sending View Delivery), EVS guarantees that messages will be delivered in the same group view to connected members (also known as the Same View Delivery property). Note that, in the EVS case, the delivery view can be different from the sending view.

The VS service is easier to program and understand, while the EVS service is more general and has better performance. The VS service is slower, since it requires application-

[1] By a network partition we mean connectivity changes due to networking hardware, routing, or a machine crash.

level acknowledgments for every group change. Moreover, it requires closed groups semantics, allowing only current members of the group to send messages to the group. EVS, in contrast, allows open groups where non-member clients can send to a group.

When securing a group communication system providing VS semantics, it is both natural and efficient to use a shared group key associated with a group view (securely refreshed upon each membership change) for data confidentiality. A message is guaranteed to be encrypted, delivered and decrypted in the same group view and, hence, with the same current key. This property does not hold in EVS since a message can be sent in one view and delivered in another view, and also because of the open groups support. Therefore, a natural solution for EVS is to use two kinds of shared keys: one key is shared between the client and the server it connects to, and the other one is shared among the group of servers. The former is used to protect client-server communication, while the latter is used to protect server-server communication.

The Spread toolkit is publicly available (see `http://www.spread.org`) and is being used by several organizations in both research, educational and production settings. It supports cross-platform applications and has been ported to several Unix platforms as well as to Windows and Java environments.

2.7 Cliques Library

Cliques is a cryptographic library that provides support for the implementation of a number of key agreement protocols for dynamic peer groups. It performs all computations required to achieve a shared key in a group and is built using the popular OpenSSL (see `http://www.openssl.org`) library. The toolkit assumes the existence of a reliable communication platform that transports protocol messages, provides ordering of messages and information about group membership, as well as synchronization rounds between group members when required by the protocols. All messages are authenticated by means of digital signatures (both RSA and DSA signatures are supported).

Currently, Cliques includes five group key management protocols. Four of them are contributory key agreement protocols: Group Diffie-Hellman (GDH), Tree Group Diffie-Hellman (TGDH), Steer et al. (STR) and Burmester-Desmedt (BD), and one of them, Centralized Key Distribution (CKD), is a centralized key distribution protocol modified to modified to provide key independence and perfect forward secrecy. Each is briefly mentioned

below and discussed in more detail in Appendix A.

GDH is a protocol based on group extensions of the two-party Diffie-Hellman key exchange (56) and provides fully contributory key agreement, authenticated my means of digital signatures. GDH is fairly computation-intensive requiring $O(n)$ cryptographic operations upon each key change. It is, bandwidth-efficient, especially in handling member leave and group partition.

CKD is a centralized key distribution (modified to provide key independence and perfect forward secrecy) where the key server is dynamically chosen from among the group members (99). The key server uses pairwise Diffie-Hellman key exchange to distribute keys. CKD is comparable to GDH in terms of both computation and bandwidth costs.

TGDH combines a binary key tree structure with the group Diffie-Hellman technique (72). In a tree built by TGDH every node has associated a key. The leaves represent contributions from group members, the root represents the group key and any other nodes represent Diffie-Hellman between the children. TGDH is efficient in terms of computation as most membership changes require $O(\log n)$ cryptographic operations.

STR (74) is a form of TGDH, using a so-called "skinny" or imbalanced tree. The main difference between STR and TGDH consists of the method use to maintain the tree used to compute the group key. The scheme is based on the protocol by Steer et al. (71). STR is more efficient than the previously mentioned protocols in terms of communication; whereas, its computation costs for subtractive group events are comparable to those of GDH and CKD.

BD is another group variation of group Diffie-Hellman proposed by Burmester-Desmedt (61). It is efficient in computation requiring a constant number of exponentiations upon any membership (group key) change. However, communication costs are significant.

All protocols supported by the Cliques library provide key independence and perfect forward secrecy. Only outside intruders (both passive and active) are considered in Cliques. In this model, any entity who is not a current group member is considered an outsider. Based on this definition any former or future member is also considered an outsider. Attacks coming from the inside of the group are not considered, as the focus is on the secrecy of group keys and the integrity of the group membership. Consequently, insider attacks are not relevant in this context since a malicious insider can always reveal the group key and/or its own private key(s) thus allowing for fraudulent membership authentication.

All the above protocols were proven secure with respect to passive outside (eaves-

dropping) attacks (56; 72; 74; 61). Active outsider attacks consist of injecting, deleting, delaying and modifying protocol messages. Some of these attacks aim to cause denial of service and we do not address them. Attacks with the goal of impersonating a group member are prevented by the use of public key signatures, since every protocol message is signed by its sender and verified by all receivers. Other, more subtle, active attacks aim to introduce a known (to the attacker) or old key. These are prevented by the combined use of: timestamps, unique protocol message identifiers and sequence numbers which identify the particular protocol run.

Chapter 3

Model

In this chapter we define the failure model under which the group communication system operates in Section 3.1 and present the group communication semantics considered for this work in Section 3.2

3.1 Failure Model

We consider a *distributed system* that is composed of a group of processes executing on one or more CPUs and coordinating their actions by exchanging messages (100). The message exchange is conducted via asynchronous multicast and unicast messages. While messages can be lost, we assume that message corruption is masked by a lower layer.

The system is subject to process crashes and recoveries. If a process has several components, a crash of any component of a process is considered a process crash. We assume that the crash of any of these components is detected by all the other components and is treated as a process crash.

Due to network events (e.g., congestion or outright failures) the network can be split into two or more disconnected subnetwork fragments. We refer to such an event as a *network partition*. While processes are in separate disconnected components they cannot exchange messages. However, processes in the same network component can communicate with each other.

When a network partition is repaired, the disconnected components merge into a larger connected component. We refer to such an event as a*network merge*. After a network merge, processes previously disconnected can exchange messages.

We consider that the network eventually stabilizes if from some point onward no processes crash or recover, communication is symmetric and transitive, and no changes occur in the network connectivity.

Byzantine process failures are not considered in this work.

3.2 Group Communication Model

A group communication system provides two fundamental and interrelated services: group membership, and reliable and ordered multicast message dissemination. The membership service maintains a list of the currently active and connected processes, logically organized in groups. The output of the membership service is called a group view. The list of connected processes members of a group, can potentially change over time for several reasons. In a fault-free network, the group change can be caused by members voluntarily joining or leaving the group. However, faults can happen, for example processes can get disconnected or crash, or network partitions can prevent members from communicating. When faults are healed, group members can communicate again. All the above events can also trigger changes in the group membership. When the list changes, the membership service reports the change to the group members by installing a new view. The membership service strives to install the same view (e.g. with the same identifier and list of members) at mutually connected members.

The reliable ordered multicast services deliver messages to the current view members, according to some ordering services. Most group communication systems provide FIFO, causal, agreed (or total order) and safe reliable ordering services. More than one group can exist in the system, and a process can be a member of more than one group.

The interrelation between the membership service and the multicast service is given by the fact that the delivery of messages takes place in some view delivered by the membership service. Group communication systems operating in a network partition failure model, can not guarantee total order across disconnected components, but they guarantee total order for a connected component. However, this order is provided with respect to views, and no guarantee is provided when the views are changing. The reliability property prevents having holes in the ordering of messages. A particular case is the safe delivery service that is a “best-effort” with respect to ”all-or-nothing” semantics that specifies that either all the processes deliver a message or none of them do this. We also note that

"all-or-nothing" semantics can not be guaranteed in an asynchronous network subject to partitions, where messages can be lost. The safe delivery service specifies that all members of the current view have received this message from the network and they will deliver the message unless they crash, even if the network partitions at that point.

In order to specify when the ordering and safe delivery properties are met inside a view, group communication systems deliver to the application an additional notification referred as a*transitional signal*.

The membership and ordered message delivery services were formalized in several different group communication models (40; 5), each providing a different set of semantics to the application. Many models are called Virtual Synchrony or some variant on the name based on a property, *Virtual Synchrony*, that states that processes moving together from one view to another subsequent view deliver the same set of messages in the former view. However, not all of these models include the same set of properties and to the best of our knowledge, a canonical "Virtual Synchrony model" has not been defined in the literature. A good survey describing many of the variations of virtual synchrony semantics can be found in (98).

Virtual Synchrony is a very useful service for applications that implement data replication using the state machine approach [Lamport 78; Schneider 1990]. These type of applications rely on the agreed ordering service to maintain consistency of the replicas. In a model where network partitions can occur, replicas might diverge and reach different states. When a network merge occurs, a state transfer is performed to achieve consistency among all replicas. The Virtual Synchrony property allows processes that "continue together" from one view to another to avoid state transfer. Determining the set of processes that continue together can not be done exclusively using the list of current members (see (98) for a detailed explanation). Therefore, an additional information provided with the view by a group communication system is what is referred as the *transitional set*. This set represents the set of processes that continue together with the process to which the membership notification was delivered and allows processes to locally determine if a state transfer is required. Different transitional sets may be delivered with the same view at different processes.

3.2.1 Virtual Synchrony Semantics

One property of the Virtual Synchrony model is the *Sending View Delivery* (98) property, which requires messages to be delivered in the same view they were sent in. To satisfy *Sending View Delivery* without discarding messages from live and connected members, a group communication system must block the sending of messages before the new view is installed (101). This is achieved in the following manner. When a group membership change occurs, the group communication system sends a message to the application, referred as *flush request*, asking for permission from all current group members to install a new view. The application must respond with a *flush acknowledgment* message agreeing with the group membership change. Before sending this acknowledgement, the application should sent all the messages that would like to be delivered in the old view. After sending the acknowledgment, the application is not allowed to send any more messages until the new view is delivered.

The properties of the flush mechanism we described above are:

1. *Flush Acknowledgment*

   For processes already part of a group, a new membership is preceded by a *flush request* message delivery. The new membership is delivered only after all the processes part of that group, sent a *flush acknowledgment*. In case the group membership change is caused by the join of a new process, no *flush request* message will be delivered to the joining process and the membership notification is the first message delivered to it.

2. *Block Messages*

   The group communication system does not allow a process to send messages between the time the *flush acknowledgment* was sent by that process, till the time the new membership is delivered to it. In case the process attempts to send message during the restriction time, an error will be returned to the client.

The group communication service interacts with the application and the network through a set of events. The group communication service receives notifications about the network connectivity and information about process faults, and based on them adjusts the new group view. It also uses the network as means of communication. An application can generate events to the group communication service by expressing the desire to join or leave a specific group, or to send messages. In turn, the group communication service delivers to

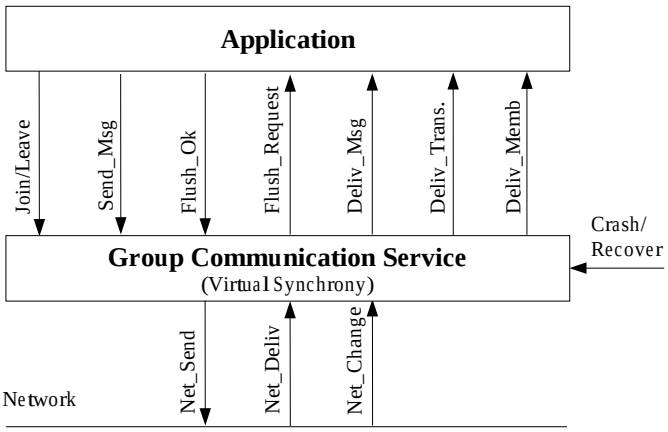


Figure 3.1: Group Communication Service: Virtual Synchrony Semantics

the application messages received from the network and membership and transitional signal notifications.

Figure 3.1 describes the set of events between the group communication service and the application and between the group communication service and network. The group communication service receives notifications about the network connectivity through *Net_change* primitive, while send and receive operation are achieved via network using the *Net_Send* and *Net_Deliv* operations.

The interaction between the group communication system and an application is defined by the following set of events:

- deliver a flush request message using the *Flush_Request* primitive.
- deliver a regular message using the *Deliver_Msg* primitive.
- deliver a transition signal using *Deliver_Trans* primitive.
- deliver a group membership change via *Deliver_Memb*. The output of this notification is a view which specifies the list of the group members. Each view has an associated transitional set and an identifier.
- send a message (unicast or multicast) by using the *Send_Msg* primitive.
- send a flush acknowledgment message via the *Flush_Ok* primitive.
- join or leave the group via the *Join* or *Leave* primitives.

We define now formally, the virtual synchrony semantics properties for the group communication system. This set of properties is largely based on the popular survey in (98) and the definition of related semantics in (40) and (97).

We consider that some event e occurred in view v at process p if the most recent view installed by process p before the event e was v.

1. *Self Inclusion*

   If process p installs a view v then p is a member of v.

2. *Local Monotonicity*

   If process p installs a view v_2 after installing a view v_1 then its identifier id_2 is greater than v_1's identifier id_1.

3. *Sending View Delivery*

   A message is delivered in the view that it was sent in.

4. *Delivery Integrity*

   If process p delivers a message m in a view v, then there exists a process q that sent m in v causally before p delivered m.

5. *No Duplication*

   A message is sent only once. A message is delivered only once to the same process.

6. *Self Delivery*

   If process p sends a message m, then p delivers m unless it crashes.

7. *Transitional Set*

   1) Every process is part of its transitional set for a view v.

   2) If two processes p and q install the same view and q is included in p's transitional set for this view, then p's previous view was identical to q's previous view.

   3) If two processes p and q install the same view v and q is included in p's transitional set for v, then p and q have the same transitional set for v.

8. *Virtual Synchrony*

   Two processes p and q that move together[1] through two consecutive views v_1 and v_2 deliver the same set of messages in v_1.

[1] If process p installs a view v with process q in its transitional set and process q installs v as well, then p and q are said to move together.

9. FIFO Delivery

If message m_1 is sent before message m_2 by the same process in the same view, then any process that delivers m_2 delivers m_1 before m_2.

10. Causal Delivery

If message m_1 causally precedes message m_2 and both are sent in the same view, then any process that delivers m_2 delivers m_1 before m_2.

11. Agreed Delivery

1) Agreed delivery maintains all causal delivery guarantees.

2) If agreed messages m_1, and later, m_2 are delivered by process p, and m_1 and m_2 are also delivered by process q, then q delivered m_1 before m_2.

3) If agreed messages m_1, and later, m_2 are delivered by process p in view v, and $m2$ is delivered by process q in v before a transitional signal, then q delivers m_1. If messages m_1, and later, m_2 are delivered by process p in view v, and $m2$ is delivered by process q in v after a transitional signal, then q delivers m_1 if r, the sender of m_1, belongs to q's transitional set.

12. Safe Delivery

1) Safe delivery maintains all agreed delivery guarantees.

2) If process p delivers a safe message m in view v before the transitional signal, then every process q of view v delivers m unless it crashes. If process p delivers a safe message m in view v after the transitional signal, then every process q that belongs to p's transitional set delivers m after the transitional signal unless it crashes.

13. Transitional Signal

Each process delivers exactly one transitional signal per view.

3.2.2 Extended Virtual Synchrony Semantics

The Virtual Synchrony Model (due to the Sending View Delivery property) is slow, since it requires application-level acknowledgments for every group change. Moreover, it requires closed groups semantics, allowing only current members of the group to send messages to the group.

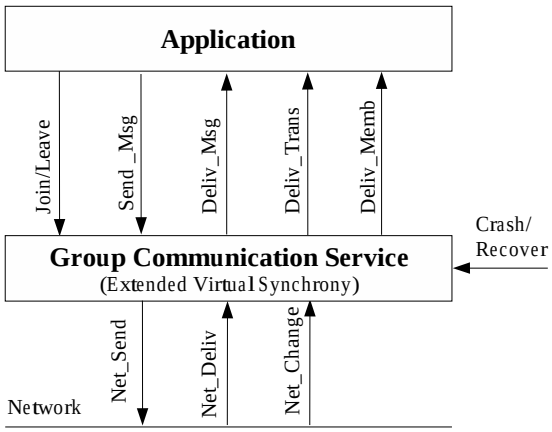


Figure 3.2: Group Communication Service: Extended Virtual Synchrony Semantics

Extended Virtual Synchrony Model avoids blocking the application by weakening the Sending View Delivery property and requiring only that a message be delivered at the same view at every process that delivers it (referred as Same View Delivery property). Although Same View Delivery is strictly weaker than Sending View Delivery, it is sufficient for applications that do not require knowing in which views messages were multicast.

In terms of the interaction between the group communication service and the application, the flush request mechanism is not required anymore. The interaction between the group communication service and the application and the environment is presented in Figure 3.2.

The Extended Virtual Synchrony Model is given by all the properties described in Section 3.2.1, with the modification that Property 3 (Sending View Delivery) is replaced by the Same View Delivery property defined below.

Same View Delivery

If processes p and q both receive message m, they receive m in the same view.

We note that typically group communication systems that provide only Same View Delivery without providing Sending View Delivery (Transis, Spread) implement a "heavy-weight" service that provides Sending View Delivery and the corresponding reliability property between a set of servers, and compose this service with an asynchronous FIFO buffer thus yielding weaker semantics (satisfying only Same View Delivery), as a "light-weight" membership service for clients.

Both VS and EVS semantics guarantee that group members see the same set of messages between two sequential group membership events and that the order of messages requested by the application is preserved. They also guarantee that all messages are delivered in the same view. However, there is a major difference in this last aspect: while VS guarantees that messages are delivered to all recipients in the same view as the sending application thought it was a member of at the time it sent the message (also known as Sending View Delivery), EVS guarantees that messages will be delivered in the same group view to connected members (also known as the Same View Delivery property). Note that in the EVS case the delivery view can be different from the sending view.

The VS service is easier to program and understand, while the EVS service is more general and has better performance. The VS service is slower, since it requires application-level acknowledgments for every group change. Moreover, it requires closed groups semantics, allowing only current members of the group to send messages to the group. In contract, EVS, allows open groups where non-member clients can send to a group.

Chapter 4

Fault-Tolerant Key Agreement

Secure, robust and efficient key management is critical for secure group communication. Designing key management protocols that are robust and efficient in the presence of network partitions and process faults is a challenging task. Most multi-round cryptographic protocols do not offer built-in robustness with the notable exception of protocols for fair exchange (102).

In this chapter we focus on fault-tolerant group key agreement, by demonstrating how provably secure, multi-round group key agreement protocols can be combined with reliable group communication services to obtain provably **fault-tolerant** group key agreement solutions. More precisely, we present two robust contributory key agreement protocols which are resilient to any sequence (even cascaded) of group events while preserving group communication membership and ordering guarantees. Both protocols are based on the Group Diffie-Hellman (GDH) contributory key agreement method that generalizes on the two-party Diffie-Hellman key exchange (57) and uses the services of a group communication system supporting Virtual Synchrony semantics.

Our first protocol utilizes membership information provided by the group communication system in order to appropriately re-start the GDH scheme, in an agreed-upon manner, every time the group changes. The second protocol optimizes the performance of common group changes at the cost of a more sophisticated protocol state machine. We prove our protocols preserve membership and ordering guarantees of the group communication system and show that the security properties of the GDH key agreement protocol are preserved.

The rest of the chapter is organized as follows. We first define more precisely the

problem we set up to solve in Section 4.1, and layout our threat and thrust model in Section 4.2. Since our protocols are based on the GDH key agreement scheme, we give a high level overview of GDH (56) in Section 4.3, focusing on the GDH merge protocol that is used as building block for our algorithms. We then present our two robust protocols and prove their correctness in Section 4.4 and 4.5. Section 4.6 concludes this chapter.

4.1 Problem Definition

Most of the key agreement cryptographic protocols specified in the literature are not atomic, they require more than one communication round. At the same time, they make unrealistic assumptions about the network. In particular, they assume that once the key agreement protocol starts, all group members are reachable and alive. This is not true in asynchronous distributed systems supporting network partitions and process crashes. Therefore, these protocols do not operate correctly especially if subtractive events change the group membership, while the key agreement protocol is in progress.

Since the focus of this work is on providing security services for group communication systems that are very powerful tools for developing fault-tolerant applications, we need a key agreement protocol able to handle changes in the group occurring while the key agreement protocol is performing.

A possible approach is to integrate the key agreement protocol with a failure detector in order to detect processes that are not connected anymore. However, this solution is not straight-forward , since processes need to reach also agreement about how to handle the failures, for example some of the failures will actually require the restart of the key agreement protocol.

Our approach is to layer the key agreement protocol on top of a group communication service providing Virtual Synchrony semantics. The output of our algorithm is a secure group communication service, therefore the Virtual Synchrony (VS) semantics as defined in Section 3.2.1 must be preserved.

The Virtual Synchrony model enables the use of a shared group key per view, securely refreshed upon each membership change. This key can be used for example to provide data confidentiality by encrypting all client data. Because of the Sending View Delivery property of the Virtual Synchrony model described in Section 3.2.1, any message is guaranteed to be encrypted, delivered and decrypted in the same group view and, hence,

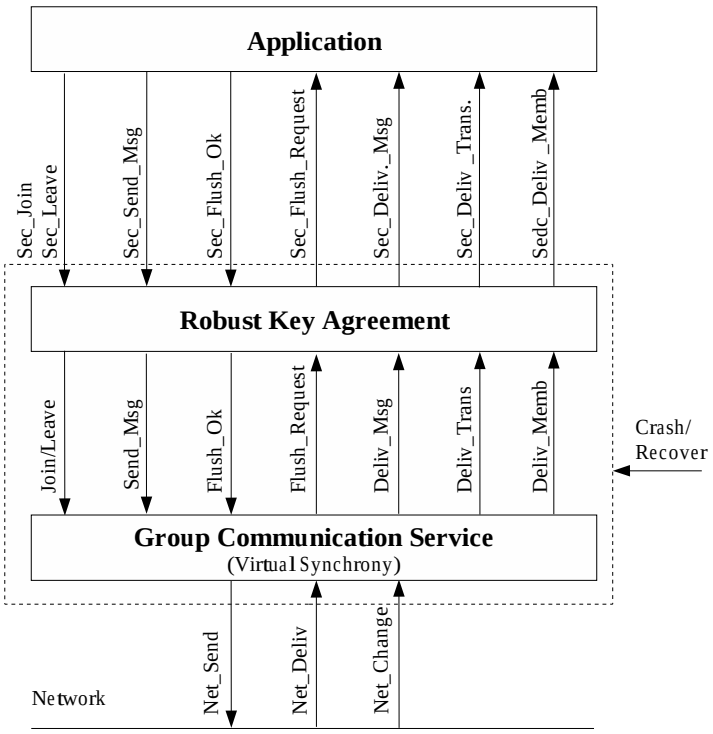


Figure 4.1: Secure Group Communication Service

with the same current key.

Figure 4.1 presents our secure group communication service. The group key agreement (GKA) interacts with both the application and the GCS. In order to provide Virtual Synchrony, the GKA must implement the flush request mechanism described in Section 3.2.1 as follows. When a *flush request* message is received from the GCS, GKA delivers it to the application. When the application acknowledgment message is received, the GKA sends down to the GCS.

Throughout the remainder of this section, we mean by GCS, a group communication service providing Virtual Synchrony semantics.

The model under which this work is conducted is the failure and group communication semantics from Chapter 3, and the threat model is presented in the following section.

4.2 Trust and Threat Model

We consider an adversarial model where attacks can come only from outside. An outsider is anyone who is not a current group member. Any former or future member is an outsider according to this definition. We do not consider insider attacks as our focus is on the secrecy of group keys and the integrity of the group membership. The latter means the inability to spoof authenticated membership. Consequently, insider attacks are not relevant in this context since a malicious insider can always reveal the group key or its own private key, thus allowing for fraudulent membership.

An attacker can eavesdrop the network with the aim of discovering the group key(s). Also it can inject, delete, delay and modify protocol messages. An attacker can aim to impersonate a group member. We assume the existence of a PKI infrastructure that allows the prevention of impersonation by using of digital signatures. Every protocol message is signed by its sender and verified by all the receivers.

An attacker can also conduct more subtle, active attacks with the goal of introducing a known (to the attacker) or old key. These attacks can be prevented by the combined use of timestamps, unique protocol message identifiers and sequence numbers which identify the particular protocol run. For GDH this modification was formally proven secure against active adversaries in (75; 76).

We assume that the VS membership service is authenticated my using digital signatures (for example using 1024-bit RSA keys).

We do not address any attack that aims to cause denial of service. We also do not consider any form of Byzantine attacks.

4.3 Overview of the Group Diffie-Hellman Key Agreement Protocol

Group Diffie-Hellman (GDH) IKA.2 (56) (also referred in literature as GDH3) is a contributory key agreement protocol that is a straight-forward extension of the 2-party Diffie-Hellman key exchange protocol (57). The group key K_{group} is in the form $K_{group} = g^{N_1 N_2 \ldots N_n}$, where N_i is the contribution of member i to the group key K_{group}.

The protocol provides strong security properties: key independence, backward and forward secrecy, and perfect forward secrecy. In order to achieve these properties, the

shared key is never transmitted over the network, even in encrypted form. Instead, a set of partial keys, $K_i = g^{\frac{N_1 N_2 \ldots N_i \ldots N_n}{N_i}}$ (that are used by individual members to compute the group secret) is sent. One particular member – called group controller – is charged with the task of building and distributing this set. The contributions of new members are collected by passing a token between the controller and the new members. The group controller is not fixed and has no special security privileges.

GDH consists of a set of protocols, designed to refresh the group key depending of the nature of the event that changed the group: join of a new member, leave of a member, massive join, or massive leave. We note that in the context of group communication systems, massive join and massive leave are triggered by network partitions or merges. In the following we informally describe the merge and leaves protocols, for a more formal description see Appendix A.1.

The core of the GDH suite is the merge protocol, which can also be used for refreshing the key in case a join occurred. Its goal is to collect contributions to the shared group key from new members. The protocol works as follows. When a merge event occurs, the current controller refreshes its own contribution to the group key, generates a new token and then passes it to one of the new members. When the new member receives the token, it adds its own contribution and then passes the token to the next new member. The token passed between the members is referred as a *partial token*. The set of new members and its ordering is decided by the underlying group communication system. The actual order is irrelevant to GDH.

Eventually, the token reaches the last new member. This new member, who is slated to become the new controller, broadcasts the token to the group without adding its contribution. This message is referred as *final token* because its content will be used as a base of creating all the partial keys as follows. Upon receiving the broadcast token, each group member (old and new) factors out its contribution and unicasts the result (called a *factor-out token*) to the new controller. The new controller collects all the factor-out tokens, adds its own contribution to each of them, builds the set of partial keys and broadcasts it to the group. This message is referred as the *list of partial keys*. Every member can then obtain the group key by searching the list for their corresponding item and then factoring in its contribution.

We note that GDH treats merge of two or multiple groups by choosing one group as a base group, and then adding the rest of the members to the base group.

The leave protocol defines how the key will be refreshed if one or more members left the group. The protocol limits the communication to only one message, at the expense of having the group controller performing a number of exponentiations scaling linearly with the size of the new group. When some of the members leave the group, the controller (who, at all times, is the most recent remaining group member) removes their corresponding partial keys from the set of partial keys, refreshes each partial key in the set and broadcasts the set to the group. Every remaining member can then compute the shared key. Note that if the current controller leaves the group, any of the remaining members can become a group controller, the rule used by GDH is that the last remaining member becomes the group controller. The leave protocol can handle both individual leaves or multiple leaves.

4.4 A Basic Robust Algorithm

This section discusses in details a basic robust group key agreement algorithm (GKA). We describe the algorithm and prove its correctness, i.e. that the algorithm preserves the Virtual Synchrony semantics presented in Section 3.2.1.

4.4.1 Algorithm Description

The GDH IKA.2 protocol, briefly presented in Section 4.3, is secure and correct. Security is preserved independently of any sequence of membership events, while correctness holds only as long as no additional group change takes place before the key management protocol terminates.

To elaborate on this claim, consider what happens if a subtractive (leave or partition) group membership event occurs while the protocol is in progress, e.g., while the group controller is waiting for individual unicasts from all group members. Since the GDH protocol does not employ any fault-detection mechanisms, it is unaware of the fact that certain members are not alive anymore, or simply left the group. Following the protocol specifications, the group controller will not proceed until all factor-out tokens (including those from departed members) are collected. Therefore, the system will block. Similar scenarios are also possible, e.g., if one of the new members crashes while adding its contribution to a group key. In this case, the token will never reach the new group controller and the protocol will, once again, simply block.

If the nested event is additive (join or merge), the protocol operates correctly. In other words, it runs to completion and the nested event is handled serially. We note, however, that this is not optimal since, ideally, multiple additive events can be "chained" effectively reducing broadcasts and factor-out token implosions.

As the above examples illustrate, the protocol does not operate correctly in the presence of certain cascaded membership events (specifically, when the interrupting events are subtractive events). This behavior makes GDH impractical since its integration with a group communication system will break the high degree of robustness and fault-tolerance traditionally provided by such a system.

Our solution is to take advantage of the membership and ordering services provided by a group communication service. A natural and correct solution to this problem is as follows: every time a group membership change occurs, the group deterministically chooses a member (say, the oldest) and runs the merge GDH protocol with the chosen member initializing it. The protocol is informed about the group changes by the membership service. Note that this approach costs twice in computation and $O(n)$ more in the number of messages for the common case with no cascading membership events. This will be addressed in the second protocol described in Section 4.5.

The algorithm uses the following types of messages: GDH specific messages (see (103)) (partial_token_msg, final_token_msg, key_list_msg, fact_out_msg); membership notification messages (memb_msg); transitional signal messages (trans_signal_msg); data messages (data_msg); flush mechanism messages (flush_request_msg, flush_ok_msg).

A process starts executing the algorithm by invoking the *join* primitive of the key agreement module which translates into a group communication join call. In any state of the algorithm a process can voluntarily leave by invoking the *leave* primitive of the key agreement module which translates into a group communication leave call. A process can crash at any time, and the network connectivity can change at any time causing group members to be partitioned in two or more components, that can merge later on.

The output of our algorithm is a secure group communication service, therefore the Virtual Synchrony (VS) semantics as defined in Section 3.2.1 must be preserved. To achieve this, our algorithm takes extra care to provide delivery of the correct views, transitional signals and transitional sets to the applications. We will elaborate on these issues later.

The specification of the algorithm is defined in terms of the events presented in Table 4.1. The events are associated with a specific group and are received by the GKA. Note

Event	Message	Source
Partial_Token	partial_token_msg	GCS
Final_Token	final_token_msg	GCS
Fact_Out	factor_out_msg)	GCS
Key_List	key_list_msg	GCS
User_Message	data_msg	App.
Data_Message	data_msg	GCS
Transitional_Signal	trans_signal_msg	GCS
Membership	memb_msg	GCS
Flush_Request	flush_request_msg	GCS
Secure_Flush_Ok	flush_ok_msg	App.

Table 4.1: Events Received by the Group Key Agreement Algorithm

that both User_Message and Data_Message events are associated with a data_msg message received by the GKA, but in the first case the source of the message is an application, while in the second case the source is the GCS.

The algorithm is modeled by a state machine (see Figure 4.2) having the following states, (all events not mentioned are not possible in that specific state):

- SECURE (S): the secure group is functional, all of the members have the group key and can communicate securely; the possible events are Data_Message, User_Message, Secure_Flush_Ok, Flush_Request, and Transitional_Signal; receiving a Secure_Flush_Ok without receiving a Flush_Request is illegal;
- WAIT_FOR_PARTIAL_TOKEN (PT): the process is waiting for a partial_token_msg message; the possible events are Partial_Token, Flush_Request and Transitional_Signal; User_Message and Secure_Flush_Ok are illegal;
- WAIT_FOR_FINAL_TOKEN (FT): the process is waiting for a final_token_msg message; the possible events are Final_Token, Flush_Request and Transitional_Signal; User_Message and Secure_Flush_Ok are illegal;
- COLLECT_FACT_OUTS (FO): the process is waiting for $N-1$ fact_out_msg messages (where N is the size of the group); the possible events are Fact_Out, Flush_Request, and Transitional_Signal; User_Message and Secure_Flush_Ok are illegal;
- WAIT_FOR_KEY_LIST (KL): the process is waiting for a key_list_msg message; the

Notes: VS_set is delivered as part of the membership
: All Cliques messages but key_list_msg are sent FIFO; key_list_msg is sent as an AGREED message
: A process can leave the group in any state

Figure 4.2: Basic Robust Key Agreement Algorithm

possible events are Key_List, Flush_Request and Transitional_Signal; User_Message and Secure_Flush_Ok are illegal;

- WAIT_FOR_CASCADING_MEMBERSHIP (CM): the process is waiting for membership and transitional signal messages (memb_msg and trans_signal_msg); the possible events are Membership, Transitional_Signal, Data_Message (possible only the first time the process gets in this state), Partial_Token, Final_Token, Fact_Out and Key_List (they correspond to GDH messages from a previous instance of the GKA when cascaded events happen); User_Message and Secure_Flush_Ok are illegal;

The pseudo-code corresponding to the state machine from Figure 4.2 is presented in Algorithms 1,2, 3, 4, 5, 6, 7.

The algorithm handles an event by performing two types of actions: group communication specific operations (message delivery, message unicast, message broadcast, or send a flush acknowledgment) and GDH key agreement specific operations (computation or access to GDH state information).

The Cliques GDH primitives we use in describing our algorithm (clq_first_member, clq_new_member, clq_update_ctx, clq_update_key, clq_factor_out, clq_merge), clq_destroy_ctx, clq_get_secret, clq_new_gc, clq_next_member) are part of the Cliques GDH API specification (103) and are described in detail in Appendix B.

The group communication primitives used to describe the algorithm are: deliver, unicast and broadcast. The broadcast primitive allows specifying the message to be sent, the ordering service (FIFO, CAUSAL, AGREED) and the group to whom the message should be sent. The unicast primitive takes as arguments the name of the destination process and the message that must be sent.

In addition, we use several simple procedures that simplify the presentation of the algorithm:

- *alone*: given a list of all members of a group, it returns TRUE if the process invoking it is the only member of the group, FALSE otherwise;

- *ready*: given a key_list message, it returns TRUE when the list is ready to be broadcast (it contains all the partial keys), FALSE otherwise;

- *last*: given a list and a name of a process, it returns TRUE if the process is the last one on the list, FALSE otherwise;

- *is_in*: given an item and a list, returns TRUE if the list contains the item, FALSE otherwise;
- *empty*: given a list, returns TRUE if the list is empty, FALSE otherwise;
- *choose*: given a list, deterministically choose a member and returns that member;
- -: the subtraction operator for list;

We also use some important data structures. The *Membership* data structure keeps information regarding a membership notification:

- *mb_id*, the unique identifier of the view;
- *mb_set*, the list of all the members of this view;
- *vs_set*, the transitional set associated with this notification;
- *merge_set*, the members from the new view that are not in the transitional set of the new view;
- *leave_set*, the members from the previous view that are not in the transitional set of the new view.

Group communication systems usually provide only the first three pieces of information in a membership notification. The merge_set and leave_set can be computed by either the GKA or the GCS by using the membership set of the previous membership notification, and the current membership notification. To simplify the presentation of the pseudo-code of the algorithm we assume that the *merge_set* and *leave_set* are provided by the GCS as part of the membership notification[1].

Every process executes the algorithm for a specific group and maintains a list of global variables (see Figure 1):

- *Group_name*: the name of the group for which the algorithm is executed;
- *Group_key*: the shared secret of the group;
- *Me*: the name of the process executing the algorithm;

[1]Note that the way we define the *leave_set*, it includes not only the members that left the group, but also the members that are not yet completely synchronized with the rest of the group.

- *Event*: the current event being handled;
- *Clq_ctx*: maintains all the cryptographic context required by the GDH protocol, and includes the list of partial keys, the group key and the list of group members;
- *New_memb_msg*: the new membership that will be delivered;
- *VS_set*: used to compute the transitional set delivered to the application with a new secure membership.

A key point of the algorithm is ensuring that all connected and alive members either install a new secure membership or restart the protocol. The fact that the Flush_Request events are not ordered with respect to the other messages, makes the problem more challenging. In order to install a secure view, a member must receive the Key_List message. However, to ensure the correctness, a member must install a secure view only when the other will do the same way. Because of that it is not enough that a member receives the Key_List message, but it must be able to infer that eventually all connected members will receive that message and be able to compute the new key and install the secure view.

Five global boolean variables are used in order to facilitate the updating of the VS_set variable, the transitional signal delivery, the correctness of the Secure_Flush_Ok events and the delivery of secure membership notifications. They are:

- *First_transitional*: the purpose of this variable is to ensure that only the first transitional signal received from the GCS during a cascaded membership, is delivered by the GKA to the application. The variable is set to FALSE when a transitional signal was delivered by the GKA to the application and set to TRUE when a secure membership was delivered to the application.
- *First_cascaded_membership*: this variable is used to compute the new transitional set delivered to the application when a cascaded membership occurred. The variable is set to TRUE when a secure membership was delivered to the application, and FALSE when the first VS membership was received from the GCS.
- *Wait_for_sec_fl_ok*: this variable is used to ensure that the application sends a flush acknowledgement in response to a flush request. It becomes TRUE when a flush request was delivered by the GKA to the application, and indicates that a flush

acknowledgement must be received from the application. It becomes FALSE when the flush acknowledgement was received by the GKA from the application.

- *VS_transitional*: this variable memorizes if a Transitional_Signal was received by the GKA, in a cascaded membership, unlike to First_Transitional that memorizes only the first such event.

- *KL_got_flush_req*: this variable memorizes if a Flush_Request event was received by the GKA in the KL state.

The names of all global variables are capitalized whereas all other variables (lowercase) are assumed to be local.

For communication, we use the FIFO service to send all of the protocol messages, with the exception of the list of the partial keys for which we used the AGREED service. We chose to use a more expensive service for the last broadcast to reduce the complexity of the algorithm and of the proofs.

Algorithm 1 Initialization of Global Variables

```
MEMBERSHIP New_memb
PROCESS_NAME[] VS_set := EMPTY
BOOL First_transitional := TRUE
BOOL VS_transitional := FALSE
BOOL First_cascaded_membership := TRUE
BOOL Wait_for_sec_flush_ok := FALSE
BOOL KL_got_flush_req := FALSE
EVENT Event := NULL
CLQ_CTX Clq_ctx := NULL
KEY Group_key := NULL
STATE State := WAIT_FOR_CASCADING_MEMBERSHIP
/* for opt. Alg., replace the above line with:
State := WAIT_FOR_SELF_JOIN */
New_memb_msg.vs_set := EMPTY
New_memb_msg.merge_set := EMPTY
New_memb_msg.leave_set := EMPTY
New_memb_msg.mb_set := Me
New_memb_msg.mb_id := 0
```

Algorithm 2 Code Executed in SECURE (S) State

```
Case Event is
Data_Message:
    deliver(data_msg)
User_Message:
    broadcast(data_msg)
Flush_Request:
    Wait_for_sec_flush_ok := TRUE
    deliver(flush_request_msg)
Secure_Flush_Ok:
    if(Wait_for_sec_flush_ok)
        Wait_for_sec_flush_ok := FALSE
        send_flush_ok()
        State := WAIT_FOR_CASCADING_MEMBERSHIP
        /* for opt. Alg., replace above line with:
        State := WAIT_FOR_MEMBERSHIP */
    else
        illegal, return an error to the user
    endif
Transitional_Signal:
    deliver(trans_signal_msg)
    First_transitional := FALSE
    VS_transitional := TRUE
All other events:
    not possible
```

Algorithm 3 Code Executed in WAIT_FOR_PARTIAL_TOKEN (PT) State

```
case Event is
Partial_Token:
    if (!last(Clq_ctx, Me))
        partial_token_msg := clq_update_key(Clq_ctx, NULL, partial_token_msg)
        next_member := clq_next_member(Clq_ctx)
        unicast(FIFO, partial_token_msg, next_member)
        State := WAIT_FOR_FINAL_TOKEN
    else
        final_token_msg := partial_token_msg
        broadcast(FIFO, final_token_msg)
        State := COLLECT_FACT_OUTS
    endif
Flush_Request:
    send_flush_ok()
    State := WAIT_FOR_CASCADING_MEMBERSHIP
Transitional_Signal:
    if (First_transitional)
        deliver(trans_signal_msg)
        First_transitional := FALSE
    endif
    VS_transitional := TRUE
User_Message, Secure_Flush_Ok:
    illegal, return an error to the user
All other events:
    not possible
```

Algorithm 4 Code Executed in WAIT_FOR_KEY_LIST (KL) State

```
case Event is
Key_List:
    if (!VS_transitional)
        Group_key := clq_update_ctx(Clq_ctx, key_list_msg)
        New_memb_msg.vs_set := Vs_set
        deliver(New_memb_msg)
        First_transitional := TRUE
        First_cascaded_membership := TRUE
        State := SECURE
        if (KL_got_flush_req)
            Wait_for_sec_flush_ok := TRUE
            deliver(flush_request_msg)
        endif
    endif
Flush_Request:
    if (VS_transitional)
        send_flush_ok()
        State := WAIT_FOR_CASCADING_MEMBERSHIP
    else
        KL_got_flush_req := TRUE
    endif
Transitional_Signal:
    if (First_transitional)
        deliver(trans_signal_msg)
        First_transitional := FALSE
    endif
    if (KL_got_flush_req)
        send_flush_ok()
        State := WAIT_FOR_CASCADING_MEMBERSHIP
    endif
    VS_transitional := TRUE
User_Message, Secure_Flush_Ok:
    illegal, return an error to the user
All other events:
    not possible
```

Algorithm 5 Code Executed in WAIT_FOR_FINAL_TOKEN (FT) State

```
case Event is
Final_Token:
    fact_out_msg := clq_factor_out(Clq_ctx, final_token_msg)
    new_gc := clq_get_new_gc(Clq_cxt)
    unicast(FIFO, fact_out_msg, new_gc)
    KL_got_flush_req := FALSE
    State := WAIT_FOR_KEY_LIST
Flush_Request:
    send_flush_ok()
    State := WAIT_FOR_CASCADING_MEMBERSHIP
Transitional_Signal:
    if (First_transitional)
        deliver(trans_signal_msg)
        First_transitional := FALSE
    endif
    VS_transitional := TRUE
User_Message, Secure_Flush_Ok:
    illegal, return an error to the user
All other events:
    not possible
```

Algorithm 6 Code Executed in COLLECT_FACT_OUTS (FO) State

```
case Event is
Fact_out:
    key_list_msg := clq_merge(Clq_ctx, fact_out_msg,key_list_msg)
    if (ready(key_list_msg))
        broadcast(AGREED, key_list_msg)
        KL_got_flush_req := FALSE
        State := WAIT_FOR_KEY_LIST
    endif
Flush_Request:
    send_flush_ok()
    State := WAIT_FOR_CASCADING_MEMBERSHIP
Transitional_Signal:
    if (First_transitional)
        deliver(trans_signal_msg)
        First_transitional := FALSE
    endif
    VS_transitional := TRUE
User_Message, Secure_Flush_Ok:
    illegal, return an error to the user
All other events:
    not possible
```

Algorithm 7 Code Executed in WAIT_FOR_CASCADING_MEMBERSHIP (CM) State

```
Case Event is
Data_Message:
    deliver(data_msg)
Transitional_Signal:
    if (First_transitional)
        deliver(trans_signal_msg)
        First_transitional := FALSE
    endif
    VS_transitional := TRUE
Membership:
    if (First_cascaded_membership)
        VS_set := New_memb_msg.mb_set
        First_cascaded_membership := FALSE
    endif
    VS_set := VS_set - memb_msg.leave_set
    if (!empty(memb_msg.leave_set) && First_transitional)
        deliver(trans_signal_msg)
        First_transitional := FALSE
    endif
    New_memb_msg.mb_id := memb_msg.mb_id
    New_memb_msg.mb_set := memb_msg.mb_set
    if (!alone(memb_msg.mb_set))
        if (choose(memb_msg.mb_set) == Me)
            clq_destroy_ctx(Clq_ctx)
            Clq_ctx := clq_first_member(Me, Group_name)
            merge_set := memb_msg.mb_set - Me
            partial_token_msg := clq_update_key(Clq_ctx, merge_set, NULL)
            next_member := clq_next_member(Clq_ctx)
            unicast(FIFO, partial_token_msg, next_member)
            State := WAIT_FOR_FINAL_TOKEN
        else /* not chosen */
            clq_destroy_ctx(Clq_ctx)
            Clq_ctx := clq_new_member(Me, Group_name)
            State := WAIT_FOR_PARTIAL_TOKEN
        endif
    else /* alone */
        clq_destroy_ctx(Clq_ctx)
        Clq_ctx := clq_first_member(Me, Group_name)
        Group_key := clq_extract_key(Clq_ctx)
        New_memb_msg.vs_set := Me
        deliver(New_memb_msg)
        First_transitional := TRUE
        First_cascaded_membership := TRUE
        State := SECURE
    endif
    VS_transitional := FALSE
Partial_Token, Final_Token, Fact_out, Key_List:
    ignore
User_Message, Secure_Flush_Ok:
    illegal, return an error to the user
All other events:
    not possible
```

4.4.2 Security Considerations

The GDH protocol was proven secure against passive adversaries in (56). As evident from the state machine in Figure 4.2, the protocol remains intact, i.e., all protocol messages are sent and delivered in the same order as specified in (56). More precisely, with no cascaded events, our protocol is exactly the same as the original GDH MASS JOIN OR MERGE protocol (56). In case of a cascaded event, the protocol is the same as the IKA.2 (56) group key agreement protocol. Since both of these protocols are proven secure, our robust group key agreement protocol is, therefore, also provably secure. In this context, security means that it is computationally infeasible to compute a group key by passively observing any number of protocol messages. As discussed in Section 4.2, stronger, active attacks are averted by the combined use of timestamps, protocol message type and protocol run identifiers, explicit inclusion of message source and destination, and, most importantly, digital signatures by the source of the message. These measures make it impossible for the active adversary to impersonate a group member or to interfere with the key agreement protocol and thereby influence or compute the eventual group key (75; 76).

4.4.3 Correctness Proof

We now prove that the above algorithm preserves the Virtual Synchrony Model described in Section 3.2.1.

In the following, the term *secure membership notification* denotes a notification delivered by the GKA to the application. The term *VS membership notification* denotes a notification delivered by the GCS to the GKA. A *secure view* is a view installed by the GKA and a *VS view* is a view installed by the GCS.

Some useful observations can be made about membership notifications and application messages. The GKA discards VS membership events, not every VS view delivery event has a corresponding secure view delivery event. The secure membership notification is built and saved in the CM state (see Figure 7). For every VS membership received in the CM state, the list of members, the view identifier and the transitional set of the new secure membership are updated in the *New_memb_msg* variable. User messages are delivered immediately as they are received, they are not delayed or reordered.

The following two lemmas are obvious from the algorithm description and they represent the flush mechanisms properties.

Lemma 4.4.1 *The GKA blocks an application from sending messages between the time a secure_flush_ok_msg message was sent and the delivery of the new secure membership.*

Lemma 4.4.2 *When a group membership change occurs, the GKA delivers a flush_request_msg message to processes already part of the group. The new secure membership is delivered only after they answer with a secure_flush_ok message. For a joining process no flush_req is delivered and the secure membership is the first message delivered to it.*

We now prove the following lemmas.

Lemma 4.4.3 *The only state where VS membership notifications are received by the GKA is CM.*

Proof: By the Flush Acknowledgment property of the GCS, a membership notification delivery is preceded by the process sending a flush_ok_msg message, unless the process is the joining process. By the algorithm, immediately after sending a flush_ok_msg message, the process transitions to the CM state and does not leave the CM state until it receives a Membership event. A joining process starts executing the algorithm in the CM state and does not leave it until it receives a membership event.

Lemma 4.4.4 *The only states where user messages are received by the GKA from the GCS are S and CM. User messages are delivered by the GKA to the application only in the S and CM states.*

Proof: After receiving a VS membership notification in the CM state (by Lemma 4.4.3 this is the only state where membership notifications are received) the process moves to one of the states FT, PT, FO, KL, or S. The transition to state S installs a new secure view, so in that state the process can send and receive user messages. In any of the FT, PT, FO, KL or CM states the process is not allowed to send application messages.

If an application message is received in any of the FT, PT, FO or KL states, this can be a message sent in the previous secure view in state S, or a message sent by a process that completed the key agreement before this process did, have already installed the new secure view and sent messages. The first case is not possible because it contradicts the Sending View Delivery property. In the second case, note that the key list message is broadcast as an agreed message, so a user message can not be received in the KL state

before the key list message because it was sent after its sender processed the key list message (it contradicts the Causal Delivery property). Therefore, the only states where a process can receive user messages are S and CM. Since user messages are delivered as soon as they are received, they are delivered only in the S and CM states.

Lemma 4.4.5 *When process p installs a secure view v, the view includes p and the v's identifier is the identifier of the most recently installed VS view.*

Proof: By the algorithm, the view-to-be-installed is updated only when a membership notification is received from GCS (see Figure 7, Marks 1 and 2), which, by Lemma 4.4.3, occurs only in the CM state.

There are two transitions that install secure views. The first transition corresponds to a Membership event occurrence in the CM state, indicating that process p is alone. In this case, the secure membership notification is immediately delivered with p (the only one) in it and it contains the most recent VS identifier.

The second transition corresponds to a Key_List event occurrence in the KL state. In this case, at the time the new secure view is delivered, it indicates the VS group members list, and as GCS provides Self Inclusion, p is guaranteed to be on that list. In this case, when the secure view is delivered, it indicates the most recent VS identifier.

Self Inclusion

Theorem 4.4.1 *When process p installs a secure view, the view includes p.*

Proof: This holds due to Lemma 4.4.5.

Local Monotonicity

Theorem 4.4.2 *If process p installs a secure view v_sec after installing a secure view v_sec', then the identifier of v_sec is greater than the identifier of v_sec'.*

Proof: The algorithm does not create view identifiers, but uses the identifiers provided by the VS membership notifications without reordering them. By Lemma 4.4.5, p always delivers a secure view with the same identifier as the most recent VS identifier. Therefore, because it delivers a subsequence of VS identifiers and because GCS provides Local Monotonicity, the GKA provides Local Monotonicity too.

Sending View Delivery

Theorem 4.4.3 *A message is delivered by the GKA in the secure view that it was sent in.*

Proof: By Lemma 4.4.4, messages are delivered by the GKA only in the S and CM states. In the S state, the secure view is the most recent VS view (by Lemma 4.4.5), so by the Sending View Delivery of GCS, the theorem holds.

As specified by the algorithm, a process moves to the CM state after the application agreed to close the group membership by sending a flush_ok message (see Algorithm 2). Since the GKA delivers a message immediately after it was received and GCS provides Sending View Delivery, all the messages sent in a VS view will be delivered before the next VS view was received, and therefore, before a new secure view is installed.

Delivery Integrity

Theorem 4.4.4 *If a process p delivers a message m in a secure view v, then there exists a process q that sent m causally before p delivered m.*

Proof: If a process p delivers a message m in v, then there exists a process q that sent m in v, by Theorem 4.4.3. Also, by transitivity, the GKA delivers a message m causally after it was sent because:

- GKA sends m immediately after it was sent by the application.
- GCS delivers message m causally after it was sent (Delivery Integrity).
- GKA delivers m immediately after it was received from the GCS.

No Duplication

Theorem 4.4.5 *A message is sent only once using the GKA. A message is delivered only once to the same process by the GKA.*

Proof: By the algorithm, an application can send messages only in the S state, so a message is sent only once. Also, messages are delivered only in the S and CM states, immediately upon receipt from the GCS. Since GCS guarantees no duplication, the theorem holds. The GKA generates GDH messages, but these are never delivered to the application so they do not affect the No Duplication property.

Self Delivery

Theorem 4.4.6 *If a process p sends a message m, then p delivers m unless it crashes.*

Proof: By the algorithm, a message is sent by the application via the GCS, the GKA never discards application messages and it delivers them immediately after receiving them. Since GCS provides Self Delivery, the theorem is true.

Transitional Set

Theorem 4.4.7 *Every process is part of its transitional set for a secure view v_sec.*

Proof: This is true by the protocol (the way the transitional set is computed for a secure view), and by the Self Inclusion property of the GCS.

Lemma 4.4.6 *If a process p installed a secure view v_sec with process q in the members set, they both install the same next VS view, and p's VS transitional set includes q, then q must have installed v_sec.*

Proof: By the protocol, a process installs a secure view with more than one member only in the KL state. A process in the KL state installs a secure view if and only if it receives a key_list_msg message before a transitional signal for the current VS view. Because p and q move together to the new VS view and the key_list_msg is an agreed message, by the Agreed Delivery properties of GCS, q must also receive the key_list_msg message before the transitional signal. Therefore, q must also have installed *v_sec*.

Theorem 4.4.8 *If two processes p and q install the same secure view v_sec, and q is included in p's transitional set for this view, then p's previous secure view was identical to q's previous secure view.*

Proof: By the algorithm, the transitional set for a new secure membership notification is initialized to be the same as the previous secure view member set. Furthermore, members reported by VS membership notifications as not being in the VS transitional set (i.e. the *leave_set*), are removed from this set and no members are added. Due to this, if q is included in p's secure transitional set then q must have been included in all of p's VS transitional sets since the last secure view delivered at p. Additionally, p and q must have installed the same sequence of VS views prior to *v_sec* because they both installed the VS view corresponding

to v_sec and because of the GCS Transitional Set property number two. Therefore, by Lemma 4.4.6, q must have installed the same previous secure view as p.

To show that q installed no intermediary secure views, the same proof is repeated reversing p and q's roles with the additional information that p is in q's secure transitional set because of the way the set is computed and because of the GCS Transitional Set property number two.

Theorem 4.4.9 *If two processes p and q install the same secure view, and q is included in p's transitional set for this view, then p is included in q's transitional set for this view.*

Proof: Assume p and q install the same secure view, q is included in p's transitional set for this view, but p is not included in q's transitional set for this view. Two cases are possible. First, q's previous secure view was not the same as p's secure view. In this case, by theorem 4.4.8, q is not included in p's transitional set, contradicting our assumption.

Second, q's previous secure view was the same, but an intermediary VS notification delivered to q did not include p in its transitional set. Since p and q install the same secure view, it must be that p and q install the same VS view at some point. The first such view installed at q preserves the property that p is not in q's transitional set by the GCS Transitional Set property number one. By GCS Transitional Set property number two, p must not have q in its transitional set for that view. In addition, by the protocol, then q is removed from p's secure transitional set, and because p's transitional set never grows q will not be in p's secure transitional set when p and q install the new secure view, which contradicts our assumption.

Virtual Synchrony

Theorem 4.4.10 *Two processes p and q that move together through two consecutive secure views, deliver the same set of messages in the former view.*

Proof: User messages are delivered by the GKA only in the S or CM states (Lemma 4.4.4) and VS membership notifications are received by the GKA only in the CM state (Lemma 4.4.3). By the way we compute the transitional set), if process p and q move together from $v1\_sec$ to $v2\_sec$, then p and q moved together through the sequence of VS views $v1$ to $v1_1$, ..., $v1_{n-1}$ to $v1_n$, $v1_n$ to $v2$ [2]. Therefore, by the GCS Virtual Synchrony,

[2]Note that n can be zero with the in-between set potentially empty ($v1$ to $v2$).

processes p and q deliver the same set of messages between $v1$ and $v1_1$, $v1_1$ and $v1_2$, ... $v1_n$ and $v2$. No other messages are delivered between $v2$ and $v2\_sec$ installations because any such message has to be sent in $v2$ according to the GCS Sending View Delivery property.

By the protocol, upon sending the flush_ok_msg message that concludes $v1$ each process moves to the CM state and will not send data messages before installing $v2\_sec$. In particular, it will not send messages between $v2$ and $v2\_sec$. Therefore, p and q deliver the same set of messages in $v1\_sec$.

FIFO, Causal, Agreed and Safe Delivery

Lemma 4.4.7 *All the user messages delivered by the GCS are immediately delivered by the GKA, maintaining the ordering properties indicated by the GCS delivery for each message.*

Proof: By the protocol, the messages delivered by a process in secure view v_sec, are messages delivered by the GCS in a VS view v. Since messages are delivered to the application in the order they were received from the GCS, without being delayed, no application messages are dropped or duplicated, and no phantom messages are generated, the messages delivered in v_sec, support the same ordering requirements as they were delivered in v.

Theorem 4.4.11 *If message m is sent before message m' by the same process in the same secure view, then any process that delivers m' delivers m before m'.*

Proof: This holds by Lemma 4.4.7.

Theorem 4.4.12 *If message m causally precedes message m', and both are sent in the same secure view, then any process that delivers m' delivers m before m'.*

Proof: This is true by Lemma 4.4.7.

Theorem 4.4.13 *If messages m and m' are delivered at process p in this order, and m and m' are delivered by process q then q delivers m' after it delivered m.*
If messages m and m' are delivered by process p in secure view $v1\_sec$ in this order, and m' is delivered by process q in secure view $v2\_sec$ and message m was sent by a process r which is a member of secure view $v2\_sec$, then q delivered m.

Proof: This is true by Lemma 4.4.7 and because the secure transitional set is the intersection of all the VS transitional sets.

Theorem 4.4.14 *If process p delivers a safe message m in secure view v_sec before the transitional signal, then every process q of v_sec delivers m unless it crashes.*
If process p delivers a safe message m in secure view v_sec after the transitional signal, then every process q that belongs to p's transitional set delivers m after the transitional signal unless it crashes.

Proof: The claims are true because the GKA delivers messages with the same ordering guarantees with which they were delivered by the GCS (by Lemma 4.4.7), the first transitional signal received from GCS is delivered to the application and because the secure transitional set is the intersection of all the VS transitional sets.

Transitional Signal

Theorem 4.4.15 *Each process delivers exactly one transitional signal per view.*

Proof: GCS Transitional Signal property guarantees that exactly one transitional signal per view will be delivered by the GCS. In case of cascaded memberships, more than one transitional signal is received by the GKA from the GCS, but only the first one will be delivered to the application (see Mark 3 in Figures 2, 4, 3, 5, 6, 7).

4.5 An Optimized Robust Algorithm

In this section we show how the algorithm presented in the previous section can be optimized, resulting in lower-cost handling of common, non-cascaded events, while preserving the same set of group communication semantics and security guarantees.

4.5.1 Algorithm Description

The basic algorithm presented in Section 4.4 is robust even when cascaded group events occur. Every time a membership notification is delivered by the GCS, the algorithm "forgets" all the previous key agreement information (e.g. the list of partial keys) and starts the merge protocol choosing a member from the new group to initialize it. Therefore, this algorithm pays more than necessary to compute a group key in the regular case, because it does not use the existing key agreement information and treats every change as a cascaded membership.

Figure 4.3: Optimized Robust Key Agreement Algorithm

Notes: VS_set is delivered as part of the membership

: All Cliques messages but key_list_msg are sent FIFO; key_list_msg is sent as an AGREED message.

: A process can leave the group in any state

A natural idea is to give every membership change a chance to be handled according to its nature (join, leave, partition, merge or a combination of partition and merge), to take advantage of the already existing list of partial keys, and if a cascaded membership happens, invoke the more expensive basic algorithm. For join, leave, partition and merge, the GDH protocol suite defines specific optimized protocols, that we take advantage of. For example, in the case when a leave occurs, the leave protocol is invoked which requires the group controller to remove the leaving members from the list, refresh the list of partial keys and then broadcast it. Thus, leave events can be handled immediately with a lower communication and computation cost than the basic algorithm required. We discuss in Section 4.5.2 how a combined event including both joins and leaves simultaneously can be handled by a modified version of the GDH merge protocol.

The optimized algorithm (see Figure 4.3) utilizes the following two states in addition to those of the basic algorithm (all events not mentioned are not possible in that specific state):

- WAIT_FOR_SELF_JOIN (SJ): this is the initial state in which a process that joins a group enters the state machine; the process is waiting for the membership message that notifies the group about its joining. In case a network event happens between the join request and the membership notification delivery, the GCS will report the cause of the group change as being a network event and the transitional set will contain only the joining member. The possible event is a Membership. User_Message and Secure_Flush_Ok events are illegal.

- WAIT_FOR_MEMBERSHIP (M): the process is waiting for a membership notification. The possible events are: Transitional_Signal, Data_Message and Membership. The membership notification can be caused by client initiated events such as join or leave, or network events. User_Message and Secure_Flush_Ok events are illegal.

While a process starts the basic algorithm in the CM state, in the optimized algorithm a process starts the algorithm in state SJ, by invoking the *Join* primitive. At any moment, a process can voluntarily leave the algorithm by invoking the *Leave* primitive. The main difference between the robust and the optimized algorithm is that in case a membership change happens, the process moves to the M state and tries to handle the event depending on its nature (subtractive, additive or both). In the case of a cascading

Algorithm 8 Code Executed in WAIT_FOR_SELF_JOIN (SJ) State

```
Case Event is
Membership:
    VS_set := New_memb_msg.mb_set
    New_memb_msg.mb_id := memb_msg.mb_id
    New_memb_msg.mb_set := memb_msg.mb_set
    First_cascaded_membership := FALSE
    if (!alone(memb_msg.mb_set))
        if (choose(memb_msg.mb_set) == Me)
            Clq_ctx := clq_first_member(Me, Group_name)
            merge_set := memb_msg.merge_set
            partial_token_msg := clq_update_key(Clq_ctx, merge_set, NULL)
            next_member := clq_next_member(Clq_ctx)
            unicast(FIFO, partial_token_msg, next_member)
            State := WAIT_FOR_FINAL_TOKEN
        else
            Clq_ctx := clq_new_member(Me, Group_name)
            State := WAIT_FOR_PARTIAL_TOKEN
        endif
    else
    Clq_ctx := clq_first_member(Me, Group_name)
    Group_key := clq_extract_key(Clq_ctx)
    New_memb_msg.vs_set := Me
    deliver(New_memb_msg)
    First_cascaded_membership := TRUE
    State := SECURE
    endif
    VS_transitional := FALSE
User_Message, Secure_Flush_Ok:
    illegal, return an error to the user
All other events:
    not possible
```

Algorithm 9 Code Executed in WAIT_FOR_MEMBERSHIP (M) State

```
Case Event is
Data_Message:
    deliver(data_msg)
Transitional_Signal:
    if (First_transitional)
        deliver(trans_signal_msg)
        First_transitional := FALSE
    endif
    VS_transitional := TRUE
Membership:
    VS_set := New_memb_msg.mb_set
    VS_set := VS_set - memb_msg.leave_set
    New_memb_msg.mb_id := memb_msg.mb_id
    New_memb_msg.mb_set := memb_msg.mb_set
    New_memb_msg.vs_set := Vs_set
    First_cascaded_membership := FALSE
    if (!alone(memb_msg.mb_set))
        merge_set := memb_msg.merge_set
        leave_set := memb_msg.leave_set
        if (!empty(leave_set) && empty(merge_set))
            if (choose(memb_msg.mb_set) == Me)
                key_list_msg := clq_leave(Clq_ctx, leave_set)
                broadcast(AGREED, key_list_msg)
            endif
            State := WAIT_FOR_KEY_LIST
        else
            if (is_in(choose(memb_msg.mb_set), memb_msg.vs_set)) /* old member */
                if(choose(memb_msg.mb_set) == Me)
                    partial_token_msg := clq_new_update_key(Clq_ctx, leave_set, merge_set, NULL)
                    next_member := clq_next_member(Clq_ctx)
                    unicast(FIFO, partial_token_msg, next_member)
                endif
                State := WAIT_FOR_FINAL_TOKEN
            else /* new member */
                clq_destroy_ctx(Clq_ctx)
                Clq_ctx := clq_new_member(Me, Group_name)
                State := WAIT_FOR_PARTIAL_TOKEN
            endif
        endif
    else /* alone */
        Clq_ctx := clq_first_member(Me, Group_name)
        Group_key := clq_extract_key(Clq_ctx)
        New_memb_msg.vs_set := Me
        deliver(New_memb_msg)
        First_transitional := TRUE
        First_cascaded_membership := TRUE
        State := SECURE
    endif
    VS_transitional := FALSE
User_Message, Secure_Flush_Ok:
    illegal, return an error to the user
All other events:
    not possible
```

membership, everything is abandoned and the basic algorithm is executed by moving to the CM state.

The *merge_set* and *leave_set* fields of the membership notification are used to determine the cause of the group view change. In addition, we assume that the procedure *clq_update_key* procedure can handle combined network events, and we name it *clq_new_update_key* (see Appendix B).

The pseudo-code corresponding to the state machine from Figure 4.3 is presented in Algorithms 2, 3, 4, 5, 6, 7, 8, 9.

4.5.2 Handling Bundled Events

Most group events are homogeneous in nature: leave (partition) or join (merge) of one or more members. However, a group communication system can decide to bundle several such events if they occur in close proximity, i.e., within a very short time interval. The main incentive for doing so is to reduce communication costs and to limit the impact and overhead on the application.

As mentioned above, GDH defines two separate protocols that handle leave and merge events. Each of these protocols can trivially handle bundled events of the same type, i.e., the GDH merge protocol can accommodate any combination of bundled merges and the GDH leave protocol can do the same for any combination of leaves and partitions.

A more interesting scenario occurs when a single membership event bundles merges or joins with leaves or partitions. One obvious way to handle this type of event is to first invoke GDH leave to process all leaves or partitions and then invoke GDH merge to process joins or merges. However, this is inefficient since the group would essentially perform two separate key agreement protocols where only one is truly needed. We can take advantage of the fact that both protocols in GDH are initiated by the group controller. After processing all leaves or partitions, the group controller can suppress the usual broadcast of new partial keys and, instead, forward the resulting set to the first merging or joining member thereby initiating a merge protocol. This saves an extra round of broadcast and at least one cryptographic operation for each member.

4.5.3 Security Considerations.

Recall that in the merge protocol, the current controller begins by refreshing its contribution (to the group key) and forwarding the result to the first merging member. This message actually contains a set of partial keys, one for each "old" member and an extra partial key for the first new member. This message is also signed by the controller and includes the list of all members believed by the controller to be in the group at that instant.

In the optimized protocol, the controller effectively suppresses all partial keys corresponding to members who are leaving the group. This modification changes nothing as far as any outside attacks or threats are concerned. The only issue of interest is whether any members leaving the group can obtain the new key. We claim that this is impossible since the set of partial keys forwarded by the controller is essentially the same as the partial key set broadcast in the normal leave protocol. Therefore, former members are no better off in the optimized than in the leave protocol.

In addition, the new (merging) members are still unable to compute any prior group keys just as in the plain merge protocol. This is because the information available to (seen by) the new members in the optimized protocol is identical to that in the plain merge.

4.5.4 Correctness Proof

The proof that the optimized algorithm described above provides the Virtual Synchrony semantics presented in Section 3.2.1 is very similar to the proof we provided for the basic algorithm. There are some differences in the optimized algorithm:

- secure memberships can be installed in three states, CM, SJ and M;
- application messages are delivered in the S and M states;
- membership notifications are received from the GCS in the CM, SJ, and M states;
- a process is not allowed to send user messages while performing the GKA, therefore a process can not send user messages in any of the SJ, M, CM, PT, FT, FO, or KL states.

Using a reasoning similar to the one we used in the proof for the basic algorithm, the following lemmas can be proved.

Lemma 4.5.1 *The only states where VS membership notifications are received are the SJ, CM and M states.*

Lemma 4.5.2 *The only states where user messages can be received are S and M. User messages are delivered to the application only in the S and M states.*

All the Virtual Synchrony Model properties described in Section 3.2.1 can be proven by using the above lemmas and the properties provided by the underlying GCS communication system. We exemplify this, by proving the Virtual Synchrony property. Due to the similarity with the proofs we presented for the basic algorithm, we do not include a proof for each property. The full proof can be found in (104).

Virtual Synchrony

Theorem 4.5.1 *Two processes p and q that move together through two consecutive secure views, deliver the same set of messages in the former view.*

Proof: User messages are delivered to the application only in the S and M states (by Lemma 4.5.2) and VS membership notifications are received only in the SJ, CM and M states (by Lemma 4.5.1).

By the way we compute the transitional set, if process p and q move together from $v1\_sec$ to $v2\_sec$, then they moved together through the sequence of VS views $v1$ to $v1_1$, ..., $v1_{n-1}$ to $v1_n$, $v1_n$ to $v2$. If n is zero, $v2$ will be received in the M state, otherwise, $v1_1$ is received in the M state and all other possible VS views (including $v2$) will be received in the CM state. Therefore, by the GCS Virtual Synchrony property, processes p and q deliver the same set of messages between $v1$ and $v1_1$, $v1_1$ and $v1_2$, ... $v1_n$ and $v2$. No other messages are delivered between $v2$ and $v2\_sec$ installations because any such message has to be sent in $v2$ by the GCS Sending View Delivery property.

By the protocol, upon sending the flush_ok_msg message that concludes $v1$ each process moves to the M state and will not send data messages before installing $v2\_sec$. In particular, it will not send messages between $v2$ and $v2\_sec$. Therefore, p and q deliver the same set of messages in $v1\_sec$.

4.6 Conclusions

In this chapter, we demonstrated that implementing secure and robust handling of cascaded group membership is crucial in order to have a complete secure group communication system. It is possible, albeit difficult, to harden security protocols to make them robust to asynchronous network events. This chapter presented two robust contributory key agreement algorithms. We proved that, by integrating them with a group communication system supporting Virtual Synchrony, group communication membership and ordering guarantees are preserved.

Chapter 5

Performance of Group Key Agreement Protocols

In this chapter we focus on analyzing four popular contributory key management protocols for collaborative peer groups, and compare them with a centralized key management protocol, modified to provide the same security properties (e.g. key independence and perfect forward secrecy). They are: Centralized Group Key Distribution (CKD) (105), Burmester-Desmedt (BD)(61), Steer et al. (STR) (74), Group Diffie-Hellman (GDH) (56) and Tree-Based Group Diffie-Hellman (TGDH) (72) (for a detailed description of these protocols see also Appendix A). The protocols are integrated in the same framework, Spread, our reliable group communication system. We provide an in-depth comparison and analysis of the five protocols based on experimental results obtained in real-life local and wide area networks, by examining the time it takes to recompute a new group key for all possible events that can change the group membership: join, leave, partitions and merges. The analysis of the protocols' experimentally measured performance offers insights into their scalability and practicality.

All the above group key agreement protocols are extension to group of the Diffie-Hellman protocol and therefore they use exponentiations intensively. Some protocols, as Burmester-Desmedt, try to minimize the computation by distributing it to all participants, which in turn trigger an increase in the number of messages used by the protocol. Some other protocols use the opposite approach, their goal is to minimize the communication by increasing the amount of computation done by one participant.

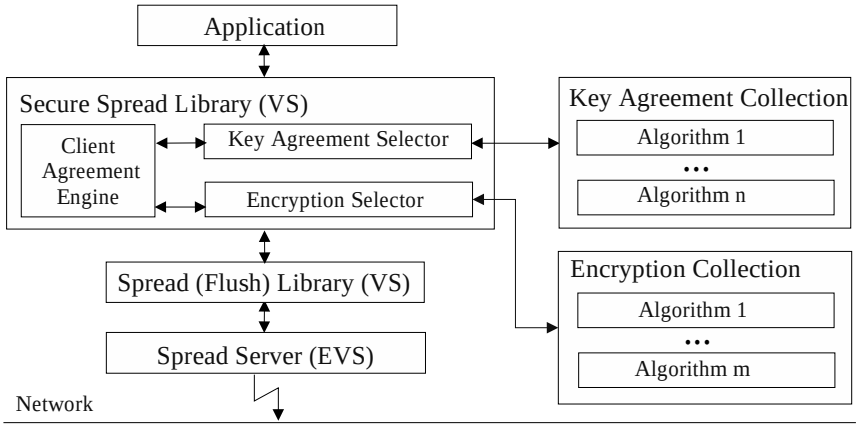


Figure 5.1: A Layered Architecture for Spread

Section 5.1 presents the framework we used for our evaluation, Secure Spread, a *layered architecture* for the Spread group communication system. We first analyze theoretically the above mentioned key management protocols in Section 5.2 and then we present experimental results in local and wide area networks in Sections 5.3 and 5.4. Section 5.5 concludes this chapter.

5.1 Layered Architecture

We evaluated the above mentioned protocols in the same framework, Secure Spread. a layered architecture for the Spread group communication system. Secure Spread is a client library providing data confidentiality and integrity, in addition to reliable and ordered message dissemination and membership services. The library is built on top of the Virtual Synchrony Spread client library; it uses Spread as its communication infrastructure and Cliques (106) group key management library primitives for key management implementation.

The Sending View Delivery property of the Virtual Synchrony model enables the use of a shared group view-specific key to encrypt client data, since the receiver is guaranteed to have the same view as the sender and, therefore, the same key. Therefore, the approach taken is to refresh the group secret key every time the view changes.

Figure 5.1 presents the layered architecture for Spread. The library has as main functionalities providing confidentiality of the data by encrypting/decrypting client data

using a group shared key and managing the shared key for each group in the system. A client that desires to communicate securely is required to connect to a server and then join a group before proceeding with the communication. The library provides an API interface very similar with the Spread interface allowing a client to connect/disconnect to a server, to join and leave a group, and to send and receive messages.

The core of Secure Spread is the Client Agreement Engine who operates as follows: every time the group membership changes, the Client Agreement Engine receives notifications from the membership service about the changes. Whenever the group membership changes, the Client Agreement Engine initiates an instance of the group key agreement protocol, ensuring its correct execution (making sure that the messages are sent to the correct destinations in the right order, and that all the members make consistent decisions with respect to installing the new secure view) . When this protocol terminates, a secure group membership change is delivered to the application and a new group key is ready for use. Applications are not allowed to send any messages while the key agreement protocol is executed. In addition, the library ensures that the Virtual Synchrony semantics are preserved.

The computation of a group key is group-specific. A client can be a member of multiple groups, each group managing its shared key with its own key agreement protocol. A Key Agreement Selector and an Encryption Selector modules are used to identify a group-specific key management and encryption algorithms. The Client Agreement Module is the one that manages the key agreement for each group.

Secure Spread currently supports five key management protocols. One of them implements centralized key distribution and is referred to as the Centralized Group Key Distribution (CKD). The protocol is adapted to provide the same security properties as the other four key agreement protocols. The other four are key agreement protocols: Burmester-Desmedt (BD) (61), Steer et al. (STR) (74), Group Diffie-Hellman (GDH) (56) and Tree-Based Group Diffie-Hellman (TGDH) (105). Each of the latter four protocols are based on various group extensions of the well-known (2-party) Diffie-Hellman key exchange (57) and provide similar security properties: key independence and perfect forward secrecy. For encryption, Secure Spread supports only one algorithm (Blowfish (44).

5.2 Theoretical Analysis

In this section, we analyze the conceptual costs of the five protocols previously mentioned. We evaluate the time to compute a new group key when the group membership changes. There are four events that can lead to a change in group membership. The first two are determined by actions initiated by users: a new user wants to become a member of the group or a current member leaves the group. We refer to these events as **join** and **leave**, respectively. Note that the latter can also happen when one member gets disconnected or simply crashes.

Another category of membership change events is related with the connectivity of the network. An unreliable network can split into disjoint components such that communication is possible within a component but not between components. For members in one component, it appears that the rest of the members have left. After the network fault heals, members previously in different components can communicate again. From the group perspective, it appears as if a collection of new members are added to the group. We refer to these events as **partition** and **merge**, respectively.

From the conceptual perspective, we are interested in two cost aspects: the cost of communication (number of rounds, number and service type of messages) and the cost of computation (number of exponentiations, signature generation and verification). Although the cost of communication in a modern high-speed LAN setting can appear negligible in comparison with the cost of, say, modular exponentiations, we discuss it nonetheless, since it becomes meaningful in LANs for protocols that trade off low computation for high communication costs. Of course, communication cost is very important in high-delay networks (e.g., WANs). Because of the distributed nature of group communication systems, we consider only serial cost of computation (e.g. computation that needs to be processed strictly serial). Computation that can be processed in parallel is collapsed accordingly.Thus, we stress that the number of cryptographic operations expressed in the Table 5.2 (for each protocol) is not the sum total for all participants.

Tables 5.1 and 5.2 summarize the communication and the computation costs associated with the five protocols we investigate. We make the following notations: n is the number of current group members, m is the number of merging members and p is the number of leaving members.

Protocols		Rounds	Messages	Unicast	Multicast
GDH	Join	4	$n+3$	$n+1$	2
	Leave	1	1	0	1
	Merge	$m+3$	$n+2m+1$	$n+2m-1$	2
	Partition	1	1	0	1
TGDH	Join, merge	2	3	0	3
	Leave	1	1	0	1
	Partition	h	$2h$	0	$2h$
STR	Join	2	3	0	3
	Leave, partition	1	1	0	1
	Merge	2	3	0	3
BD	Join	2	$2n+2$	0	$2n+2$
	Leave	2	$2n-2$	0	$2n-2$
	Merge	2	$2n+2m$	0	$2n+2m$
	Partition	2	$2n-2p$	0	$2n-2p$
CKD	Join	3	3	2	1
	Leave	1	1	0	1
	Merge	3	$m+2$	m	2
	Partition	1	1	0	1

Table 5.1: Key Management Protocols Comparison: Communication Cost

Protocols		Exponentiations	Signatures	Verifications
GDH	Join	$n+3$	4	$n+3$
	Leave	$n-1$	1	1
	Merge	$n+2m+1$	$m+3$	$n+2m+1$
	Partition	$n-p$	1	1
TGDH	Join, merge	$\frac{3h}{2}$	2	3
	Leave	$\frac{3h}{2}$	1	1
	Partition	$3h$	h	h
STR	Join	7	2	3
	Leave, partition	$\frac{3n}{2}+2$	1	1
	Merge	$3m+4$	2	3
BD	Join	3	2	$n+3$
	Leave	3	2	$n+1$
	Merge	3	2	$n+m+2$
	Partition	3	2	$n-p+2$
CKD	Join	$n+2$	3	3
	Leave	$n-2$	1	1
	Merge	$n+2m$	3	$m+2$
	Partition	$n-p-1$	1	1

Table 5.2: Key Management Protocols Comparison: Computation Cost

The height of the key tree constructed by the TGDH protocol is denoted by h[1]. The cost of the TGDH protocol depends on the tree height, the balancedness of the key tree, the insertion point of the joining tree (or node) and the location of the leaving node(s). To err on the side of safety, we compute the worst case cost for the TGDH.

The number of modular exponentiations for STR upon a leave event is determined by the location of the deepest leaving node. We, therefore, compute the average cost, i.e. the case when the $\frac{n}{2}$-th node left the group. All other protocols, except TGDH and STR, show exact cost numbers.

Our current implementations of TGDH and STR recompute a blinded key even though it has been computed already by the sponsor. This provides a form of key confirmation, since a user who receives a token from another member can check whether his blinded key computation is correct. This computation, however, can be removed for better efficiency, and we consider this optimization when counting the number of exponentiations.

The BD protocol has a hidden computation cost which is not reflected in Table 5.2. In Step 3 (see Algorithm 16), BD protocol has $n-1$ modular exponentiations with small exponents (ranging from n to 2) and $n-1$ modular multiplications. Although a single small exponentiation is negligible, the sum of all $n-1$ exponentiations is clearly not. Because of this hidden cost, it is difficult to compare the computational overhead of BD to the other protocols.

5.2.1 Join

All protocols excepting CKD require two communication rounds. CKD uses three rounds because the new member must first establish a secure channel (via Diffie-Hellman key exchange) with the current group controller. The most expensive protocol in terms of communication is BD which uses n broadcast messages for each round. The rest of the protocols use a constant number of messages, either two or three.

GDH and CKD are the most expensive protocols in terms of the computation overhead. Both schemes require a linear number of exponentiation operations relative to the group size.

[1] Instead of fully balancing the key tree, TGDH uses best-effort approach: it tries to balance the key tree only upon an additive event. The height of the key tree, however, is smaller than $2 \log n$ (maximum group size: n) (72). The tree can be better balanced when using the AVL tree management technique described in (78). However, this will incur a higher communication cost for a leave operation.

TGDH is comparatively efficient, scaling logarithmically in the number of exponentiations. STR, in turn, uses a constant number of modular exponentiations. BD requires the least modular exponentiations, but has the above discussed hidden cost.

5.2.2 Leave

Table 5.1 shows that, for a leave operation, the BD protocol is the most expensive from the communication point of view. The cost order between the CKD, GDH, STR and TGDH schemes is determined strictly by the computation cost, since they all have the same communication overhead, one round consisting of one message. Therefore, TGDH is best for handling leave events.

The computation overhead of STR, GDH and CKD scales linearly with the group size. We note that the cost of CKD is actually higher than the one listed in Tables 5.1 and 5.2, because in the case when the controller leaves the group, the new group controller must establish secure channels with all group members. Since BD, again, has a hidden cost, it is difficult to compare with other algorithms.

5.2.3 Merge

We first analyze the communication cost. Note that GDH scales linearly with the number of the members added to the group in communication rounds, while BD, CKD, STR and TGDH are more efficient using a constant number of rounds. Since BD uses n messages for each round and CKD uses $m + 1$ messages, STR and TGDH are the most communication-efficient for handling merge events.

Looking at the computation costs, it seems that BD has the lowest cost: only three exponentiations. However, the impact of the number of small exponent exponentiations is difficult to evaluate. TGDH scales logarithmically with the group size, being more efficient than STR, CKD and GDH which scale linearly with both the group size and the number of new members added to the group.

5.2.4 Partition

Table 5.1 shows that the GDH, STR and CKD protocols are bandwidth efficient: only one round consisting of one message. BD is less efficient with two rounds of n messages

each. Partition is the most expensive operation for TGDH, requiring a number of rounds bounded by the tree height.

As before, computation-wise it is difficult to compare BD with the other protocols because of its hidden cost in Step 3 (see Figure 16). TGDH requires a logarithmic number of exponentiations. GDH, STR, and CKD scale linearly with the group size.

5.3 Experimental Results in LAN

In this section we present, compare and evaluate the experimental costs of the five protocols discussed above, in two network environments: local area networks and high delay wide area networks.

We measure the "total elapsed time" from the moment the group membership changes until the moment when the group key agreement protocol finishes and the application is notified about the group change and the new key.

In case of join and leave which are events initiated by clients, this time includes all the communication and computation costs of the key agreement protocol as well as the cost of the membership service provided by the group communication system. In other words it represents the total delay experienced by an application (or user) using the Secure Spread group communication system, when it performs a join or a leave operation to a group.

In case of a merge or partition, where the group change happens due to a network connectivity change, this time includes all the communication and computation costs of the key agreement protocol as well as the cost of the membership service provided by the group communication system, but it does not include the time needed by Spread to detect that a network partition or merge occurred.

We begin by presenting performance results of the five protocols we investigate in a local area network setting. First, we present the testbed used in our tests, then discuss the particularities of the scenarios we considered and finally present results for join, leave, partition and merge operations.

5.3.1 Testbed and Basic Parameters

We used an experimental testbed consisting of a cluster of thirteen 666 MHz Pentium III dual-processor PCs running Linux. On each machine runs a Spread server. Group members are uniformly distributed on the thirteen machines. Therefore, more than

one process can be running on a single machine (which is frequent in many collaborative applications).

As communication services we used FIFO and AGREED ordering. Tests performed on our testbed show that the average cost of sending and delivering one Agreed multicast message is almost constant, ranging anywhere from 0.75 milliseconds to 0.92 milliseconds for a group size ranging from 2 to 50 members. Also, in a scenario (similar to a BD round) where each member of the group sends an Agreed broadcast message and receives all the $n-1$ messages from the rest of the members (n being the group size), the average cost is about 2 milliseconds for a group of 2 members and about 21 milliseconds for a group of size 50.

The cost of the membership service (see Figures 5.2 and 5.3) is negligible with respect to the key agreement overhead, varying between 2 and 8 milliseconds for a group between 2 and 50 members.

Message origin and data authentication are achieved via RSA (53) digital signatures. We chose RSA because the signature verification is quite inexpensive and all group key agreement protocols described in this paper rely heavily on source authentication, i.e., most protocol messages must be verified by all receivers. If all processes are located on different CPU platforms, verification is performed in parallel. In practice, however, a CPU may have multiple group processes and expensive signature verification (e.g., as in DSA (54)) noticeably degrades performance.

We used 1024-bit RSA signatures with the public exponent of 3 to reduce the verification overhead, although a quasi-standard in RSA parameter selection is $65,537$. This is because 1) there are no security risks for using 3 as a public exponent in RSA signature scheme (107), 2) BD and GDH require n simultaneous signature verifications, and 3) in our current topology, some machines can have multiple group member processes. On our hardware platform, the RSA sign and verify operations take 9.6 and 0.2 milliseconds, respectively.[2] In case the public exponent is chosen to be $65,537$, it takes 0.5 milliseconds to verify a RSA digital signature on our platform.

For the short-term group key, we use both 512- and 1024-bit Diffie-Hellman parameter p and 160-bit q. The cost of a single exponentiation is 1.7 and 5.3 milliseconds for a 512- and a 1024-bit modulus, respectively. Although the use of a 512-bit exponentiations

[2]This is not surprising since OpenSSL uses the Chinese Remainder Theorem to speed up RSA signatures.

might be considered insecure, we decided to include the results for 512-bit exponentiations to show the quantitative effect of the exponent (key generation) on the performance of the protocol.

5.3.2 Test Scenarios

The cost for each protocol, depends on several factors. We tried to design our tests such that we take into account all the factors, keeping experiments as similar and as less complex as possible.

Some of the protocols maintain specific data structures. The GDH and CKD schemes maintain a list, while TGDH and STR maintain a tree. The BD protocol is stateless. Depending on where the operations are performed in these data structures, the communication and computation cost might vary or not. For example, the cost of the GDH and BD protocols does not depend on the position of the joining or leaving member, i.e., all leave and all join operations cost the same in these protocols, while CKD can be expensive for a leave event if the leaving member is the current group controller. For STR the computation cost in case of a leave operation depends on the position in the tree of the leaving member. TGDH cost depends on many factors: the location of the leave or join node, tree height, and the balancedness of the tree.

Motivated by the above observations, we designed our tests as follows. For CKD, we take into the possibility that the controller might leave by factoring in the $1/n$ probability of the group controller leaving the group. Since the estimated cost presented for STR leave event is the average cost, we also tested the average case: the leaving member is the leaf node at height $n/2$, in the middle of the STR key tree.

TGDH is the most difficult protocol to evaluate because its cost depends on the location of the leave or join node, tree height, and the balancedness of the tree. For a truly fair comparison, Secure Spread must be first run with TGDH for a long time (with a random sequence of joins and leaves) in order to generate a random-looking tree. The experiments must then be conducted on this random tree. However, such tests are very difficult to perform. Instead, we chose a simpler experimental setting by measuring join and leave costs on an artificially balanced TGDH key tree with n members. We note that for a random tree, the cost of join will be less expensive since the member will be joined closer to the root, while leave will be more expensive, but still less expensive than GDH.

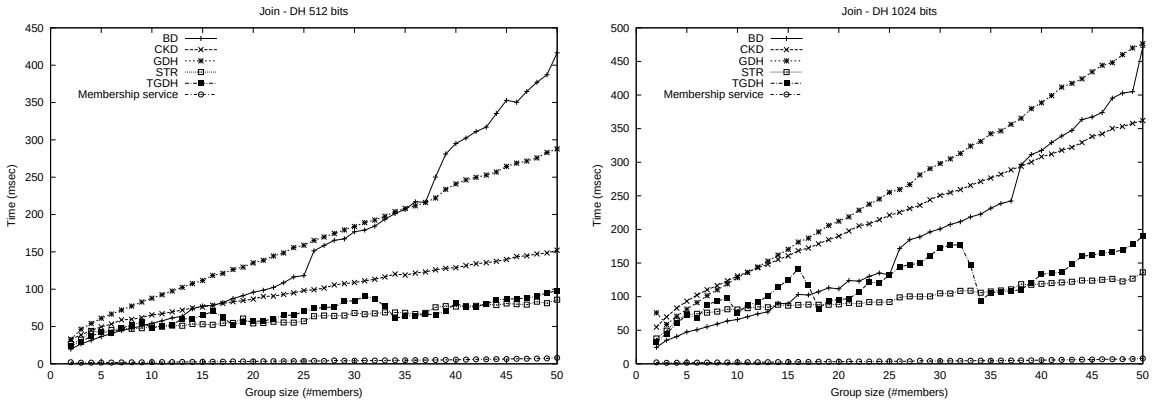


Figure 5.2: Join Average Time (LAN)

In case of partition and merge we run the same scenarios for all protocols: we partition the group is two subgroups of about the same size, and then merge them back. STR and TGDH have a clustering effect (See Section 5.3.5 for details) when partition events will be repeated in the same configuration. To emphasize this property, we partitioned the group in two subgroups, merge it back, and then partition it again. We analyze the cost of the partition operations.

5.3.3 Join Results

Figure 5.2 shows the total average time for a group to establish secure membership following a join of a new member.

In the graph on the left (512-bit modulus) it looks, overall, that STR outperforms other protocols. Closer inspection reveals that BD is actually the most efficient for small group sizes (less than 7 or so). Recall that BD involves only three full-blown exponentiations as opposed to STR's seven. However, BD has $(n+3)$ signature verifications, whereas STR only has 3. Furthermore, BD requires $O(n \log n)$ modular multiplications in Step 3 (to compute the key, see Figure 16). Finally, BD has two rounds of all-to-all broadcasts. Small group size makes all of these factors negligible. However, as the group size grows, BD deteriorates rapidly since both modular multiplications, RSA signature verifications and broadcasts add up. In fact, after passing the group size of thirty, BD becomes the worst performer if Diffie-Hellman parameter is 512. For 1024-bit modulus, GDH is the worst due to the sharp increase in modular exponentiation.

Another interesting observation about BD's performance in all measurements is that its cost roughly doubles as the group size grows in increments of 13. Recall that 13 is the number of machines used in the experiments. Because BD is fully symmetric, as soon as just one machine starts running one additional group member (process), the cost of BD doubles. Moreover, it can be noted that starting with the group size of 26, the performance degrades significantly. As mentioned before the machines we used are dual processors, so up to a group size of 26 it can be assumed that there is one client on one processor. For the other protocols this behavior is less obvious since in all of them, the most costly tasks are performed by a single member (controller or sponsor).

The graph on the right (1024-bit modulus) does not show the same deterioration in BD. It remains the best for very small groups up to 14 members. This is because the cost of exponentiations rises sharply from 512 to 1024 bits, while the cost of RSA signature verifications and broadcasts (which weighs BD down in the 512-bit case) is not felt nearly as much, while the other protocols are more affected since their cost is mainly determined by the number of exponentiations.

In both graphs, TGDH and STR are fairly close with the latter performing slightly better. Although the numbers in Table 5.2 show constant cost for STR, the measured cost increases slightly because: 1) a CPU may experience an increasing number processes as the number of members increases, and, 2) other minor overhead factors such as tree management. Conceptually, TGDH can never outperform STR in a join, since the latter's design includes the optimal case (i.e., join at the root) of the former. Experimental results, however, show that TGDH can sometimes outperform STR (see small dips in TGDH graph at around 18 and 34 members). This is because most members in a fully balanced TGDH tree compute two modular exponentiations in the last protocol round, as opposed to four in STR.

The difference between CKD and GDH comes from exponentiation and signature verifications: extra operations in GDH include n verifications, one RSA signature and one (DH) modular exponentiation. GDH and BD each have $n+3$ signature verifications, which is, as mentioned above, relatively expensive even with a 512-bit RSA modulus.

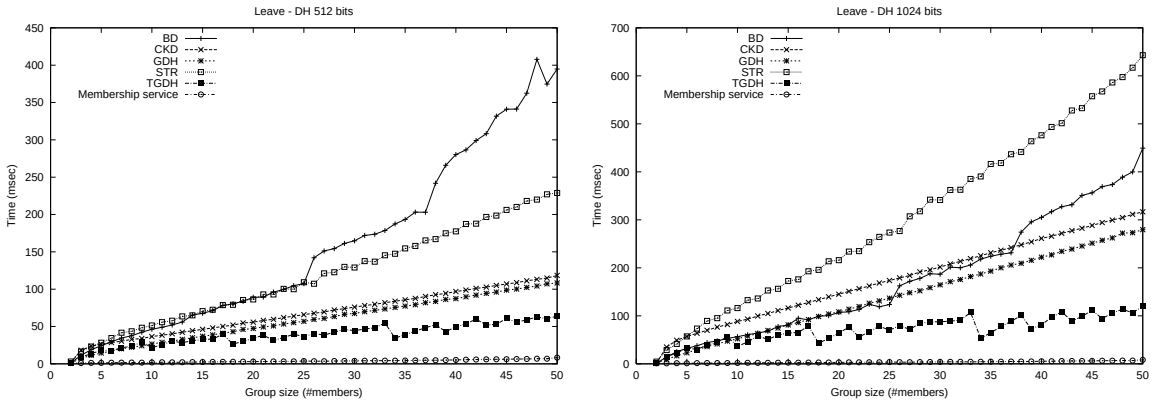


Figure 5.3: Leave Average Time(LAN)

5.3.4 Leave Results

Figure 5.3 shows the average time for a group to establish secure membership upon a member leave event. In line with the conceptual evaluation, TGDH outperforms the rest, as it requires the fewest ($O(log\ n)$) modular exponentiations. This sub-linear behavior becomes particularly evident past the group size of 30. Note that for a random tree although leave will be more expensive, it will be less expensive than leave for the GDH protocol (72) which is the second best.

BD is the worst in 512-bit leave; recall that BD is a stateless protocol and uses the same method to recompute the key, independent of the event that changed the group membership, therefore, leave and join incur the same cost. STR, CKD and GDH all exhibit linear increase in cost. CKD and GDH are quite close while STR's linear factor is $2n$ which makes its slope steeper. In addition, while STR, TGDH, CKD and GDH require only one broadcast, BD uses two rounds of n messages each.

In case of the 1024-bit modulus, STR is the most expensive protocol, since it involves (costlier in 1024- than in 512-bit case) modular exponentiations. TGDH exhibits the cost roughly twice that of the 512-bit case and remains the leader. BD, however, is no longer the worst and, at least for small group sizes, (less than 37 or so) performs close to, or better than, GDH. Once again, we attribute this to the relatively cheap cost of RSA signature verifications in the commensurately small number of full-blown 1024-bit exponentiations in BD.

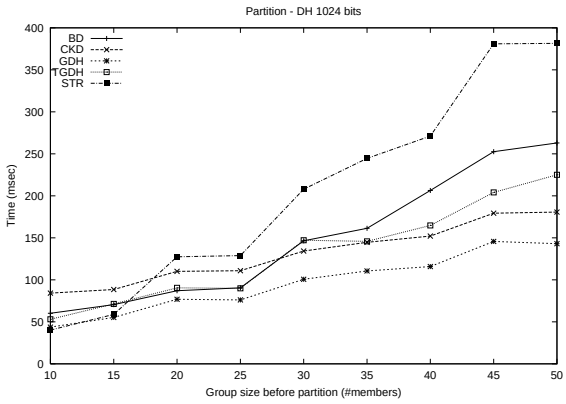


Figure 5.4: Partition Average Time (LAN)

5.3.5 Partition Results

The results in Figure 5.4 present the time required to establish a new secure membership when a group is splitted in two groups of about the same size. The partition is generated by simulating a network partition between the servers. The evaluated time does not include the time needed by Spread servers to detect that a network partition happened.

GDH outperforms the other protocols. We note that STR and TGDH are very costly in this specific scenario where half of the group is partitioned away because they need to rebuild the tree. In particular, STR is the most expensive since it also scales linearly in computation with respect to the new group size.

The cost associated with the CKD protocol is higher than we estimated in Table 5.2. When the group is partitioned in two subgroups, the old group controller will be part of one of the partitions. In this case, the cost is similar with the one we estimated in Table 5.2. However, in the other partition, all member will perceived the old group controller as being gone, so a new group controller will be chosen. In this case, all group members need to establish secure channels with the new group controller.

TGDH and STR have an advantage over the other protocols in case of partition. Due to the tree structure, both TGDH and STR have a clustering effect which basically decreases the cost of a partition when this happens multiple times in the same configuration (which statistical results show that it is a common case). To show this feature, we ran the following experiment. We partitioned a group in two equal groups, such that after the partition occurred, the odd numbered users are in one group (we refer to it as *Group 1*)

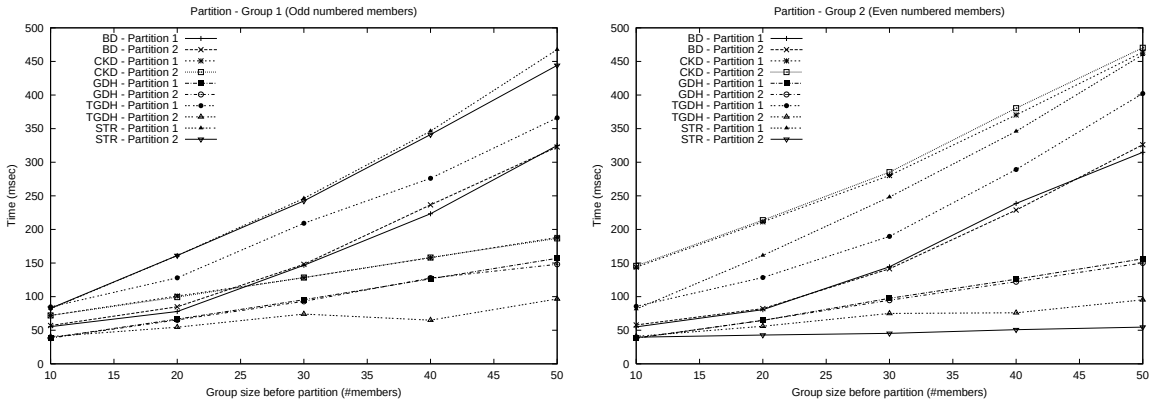


Figure 5.5: Partition Clustering Effect

and the even numbered users are in the other group (referred to as *Group 2*), merged it back and then repeat the partition event. The first partition (we refer to it as *Partition 1*), and the second partition (we refer to it as *Partition 2*) are identical.

The time needed to establish a key and install a secure membership in the two partitioned subgroups for *Partition 1* and *Partition 2* events are presented in Figure 5.5. As it can be noticed, for the BD, GDH and CKD protocols the time is about the same. For BD and GDH the only important factor in the cost of the protocol is the number of the remaining members. BD and GDH take the same amount of time, since the cost of partition depends only on the group size and the number of leaving members.

In case of CKD, there is a difference between the cost of the key agreement in *Group 1* and *Group 2*, since in *Group 2* all new members need to establish secure channels with the new group controller.

Unlike BD, GDH and CKD, for STR, a decrease can be noticed in the cost of *Partition 2* for *Group 2* (even numbered members). This is because the merge event we generated in between the two partitions, changed the structure of the tree, and the refresh for the second partition happened at a higher level in the tree. However, for *Group 1*, there is no improvement, since the refresh happens very low in the tree.

In case of TGDH, the partition protocol may cost as many as $\log n$ rounds. Then, when the partition heals, the previously separate groups are merged into a single key tree, however, they are still clustered along the lines of the partition. If another partition happens on the same link, the partitioned members are not scattered across the key tree any longer.

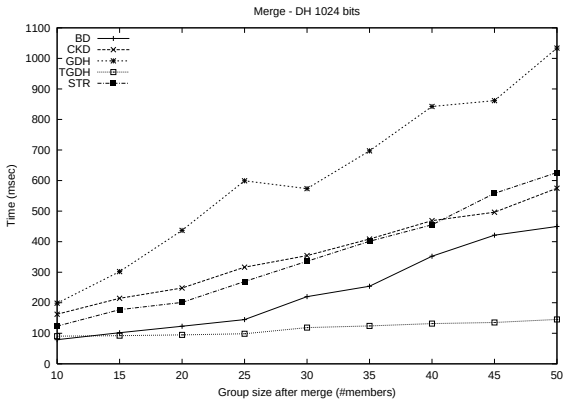


Figure 5.6: Merge Average Time (LAN)

Therefore, any subsequent partition on the same link will take only one round to complete. This improves the communication cost (also the number of signature generation) and we can see clear distinction between the first and the second partition.

5.3.6 Merge Results

The results in Figure 5.6 present the time required to establish a new secure membership when two groups of about the same size that were previously partitioned, are then merged together. This time does not include the time needed by the Spread servers to detect that the network healed. GDH is the most expensive protocol in case of merge because it scales linearly in computation with the number of the new group members and has the highest number of communication rounds. The CKD protocol has less communication rounds, however the computation required for merge is quite high since all new members (in our case $n/2$) need to establish a secure channel with the group controller.

The STR scheme has a comparable cost with CKD due to the fact that its tree is "almost" a list, so when $n/2$ members are merged, the refresh in the tree happens very deep so triggers a significant computational task. The high computation cost of STR and GDH, make BD competitive in case of a merge operation. The balanced tree structure and the small number of communication rounds allow TGDH to obtain the best performance in case of merge.

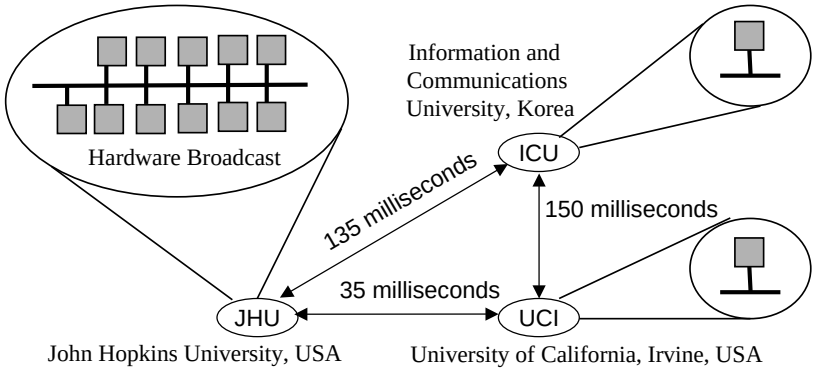


Figure 5.7: WAN Testbed

5.4 Experimental Results in WAN: An Extreme Case

In this section we present results in a WAN environment with high-delays. We focused on evaluating the total time needed to perform a join or a leave operation, from the moment the group membership changes due to the join or leave of a member, till the moment a new key was computed and delivered to all members of the group. This time includes all communication and computation costs of the key agreement protocol as well as the cost of the membership service of the group communication system. We present first the testbed used in our tests, then show results for join and leave operations.

5.4.1 Testbed and Basic Parameters

Figure 5.7 presents the network configuration we used for our experiments on WAN which uses three sites.

To achieve the same computation distribution as for the LAN experiments, we used an experimental testbed of thirteen PCs running Linux: ten 666 MHz Pentium III dual-processor PCs, one 1.1 MHz Athlon and one 930 MHz Pentium III PCs, located as follows: first eleven machines at Johns Hopkins University (JHU), Maryland, one machine at University of California at Irvine (UCI) and one at the Information and Communications University (ICU), Korea. We uniformly distributed group members among the thirteen machines with more than one group member process running on a single machine. Each

machine runs a Spread server. Approximate round-trip latencies (ping times) as reported by the ping program are: JHU - UCI 70 milliseconds, UCI - ICU 300 milliseconds and ICU - JHU 270 milliseconds. We emphasize that because of the message dissemination protocols employed by Spread, the determinant factor in the performance of our communication infrastructure is the diameter of the network.

The average delay of sending and delivering one Agreed multicast message depends on the sender's location. The actual delay (in milliseconds) is: sender at JHU – 392, sender at UCI – 328, and sender at ICU – 334. When each group member sends a broadcast message and waits to receive $n-1$ messages from the rest of the group (similar to a BD communication round), the average cost is about 1000 milliseconds for a group of size 50.

It is important to notice that, in a LAN setting, the cost of the group membership service provided by the underlying group communication system is negligible with respect to the key agreement overhead, e.g., about 7 milliseconds vs. hundreds of milliseconds. However, this relative cost becomes significantly higher in a WAN setting. The cost of the membership service as it can be seen in Figure 5.8, varies between 400 and 670 milliseconds for a join operation and between 250 and 650 for a leave operation, for a group of 2 to 50 members.

As in the LAN experiments, we used RSA 1024-bit with the public exponent of 3 to compute message signatures. On our test PCs, the RSA sign and verify operations take $6.9-17.9$ and $0.2-0.4$ milliseconds, respectively, depending on the platform. For the short-term group key, we use 512-bit Diffie-Hellman parameter p and 160-bit q. The cost of a single modular exponentiation is between 0.8 and 1.7 milliseconds.

5.4.2 Join Results

Figure 5.8 (left) presents our results for the average time required to establish a secure group membership when a new member joins the group. The graph also separately plots the cost of the insecure group membership service. The difference between the total time required by each protocol and the insecure group membership service cost, represents the overhead of the key agreement itself (both communication and computation).

The first observation is that the GDH protocol performs significantly worse than others. The main difference between GDH and the other protocols comes from communication. First, the number of rounds is greater than that of the other protocols as shown in

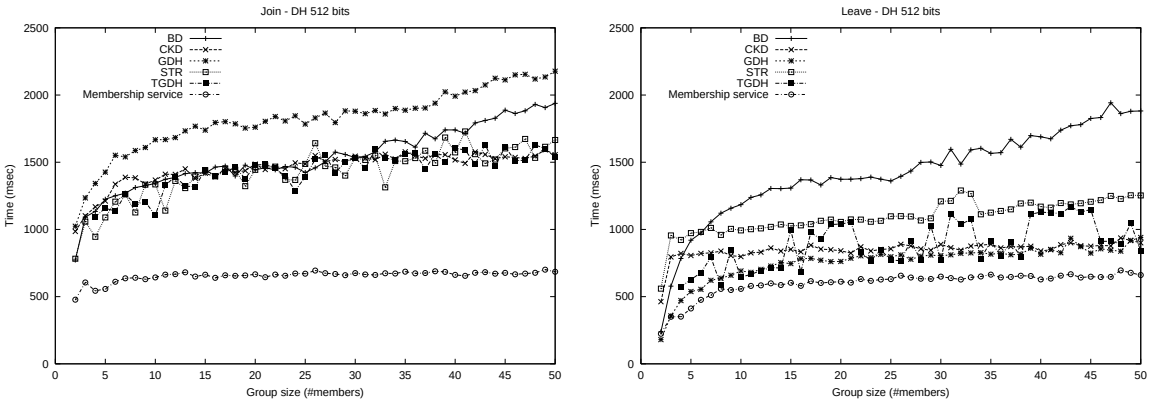


Figure 5.8: Join and Leave Average Time (WAN)

Table 5.1. GDH requires 4 rounds while others require only 2 rounds.

The rest of the protocols are more or less in the same range, with BD becoming more expensive for a group size bigger than 30, while STR and TGDH show similar performance. It is interesting to notice that both STR and TGDH come closer to the BD performance. This is also because of the communication aspect of the protocols. As stated in Table 5.1 and Sections A.4 and A.3, both STR and TGDH have 2 rounds, out of which the first round consists of two "simultaneous" broadcasts. In our implementation, these broadcasts are not simultaneous, since to achieve ordered delivery of the messages, group communication systems use a mechanism, where a token is passed between participants and only the entity that has the token is allowed to send. Because in our particular WAN setup, there are three main sites, JHU, UCI and ICU (the cost of passing the token inside a site is significantly smaller than the cost between sites), the cost of a STR and TGDH scenario - 2 members are sending two broadcasts and all the members need to receive them - is close to the cost of a BD scenario - n members broadcast and all members need to receive $n-1$ messages. [3]

BD deteriorates faster than other protocols due to the number of broadcast messages. Though CKD has three rounds, two of them involve single-message unicasts. This helps CKD to remain competitive with respect to the other protocols.

We can clearly conclude that communication costs (the number of rounds and the

[3]This is because if one member missed the token, it needs to wait for the token to pass the whole ring, while in the BD scenario if the token completes a cycle, no matter where it started, everybody succeeds to send.

numbers of messages sent in one round) of a group key agreement scheme affects severely its performance on the wide area network, particularly in one with high-delays as the one we used in our experiments.

5.4.3 Leave Results

In case of leave (see Figure 5.8, right), BD is the most expensive protocol in our WAN setup, due to the two rounds on n broadcasts and its high computational cost.

GDH, CKD and TGDH require only a single broadcast, thus, they exhibit similar performance results. Although STR also requires only one broadcast, it has significantly higher computation cost with respect to the rest.

We observe that TGDH exhibits a behavior more dynamic than GDH and CKD. We attribute this to the fact that, in CKD and GDH, the controller (who does the bulk of computation and broadcasts) was running on a fixed machine. Whereas, in TGDH, the sponsor (who also does most of computation and broadcasts) was running on any of the 13 testbed machines. If tested with a fixed sponsor, we suspect that TGDH, GDH, CKD would have almost identical cost.

5.5 Conclusions

We presented a framework for cost evaluation of group key agreement protocols in a realistic network setting. The focus was on four notable group key agreement protocols and a centralized key management protocol integrated with a reliable group communication system (Spread). After analyzing the protocols' conceptual costs, we measured their behavior in both LAN and WAN settings.

Following our experiments and their interpretation (as discussed above), we conclude that computation cost is most important in a low delay network and communication cost is most important in a high delay network.

In a LAN setting, TGDH is the best performing protocol overall. However, we also note that for small groups – no greater than, say, a dozen members – BD is a slightly better performer. Another factor in BD's favor is its simplicity: all operations are symmetric and are implemented via the same protocol with few data structures to manage. In contrast, TGDH involves some non-trivial tree management. (See (72) for details.)

An additional factor can skew the comparable performance of the evaluated protocols. TGDH was evaluated with a well-balanced tree. In a random (unbalanced) tree the join cost would have been less expensive since the joining node would have been inserted nearer the root node, while the leave cost would have been more expensive, but less expensive than GDH (72).

In a high-delay WAN setting, TGDH and CKD exhibited the best performance. Since TGDH has smaller computation overhead, we expect it to outperform CKD in a medium delay wide area network ($70 - 100$ milliseconds round-trip links).

The results we presented indicate that TGDH is the protocol that is the best compromise in performance in both environments.

Chapter 6

Integrated Secure Group Communication Architecture

There are two basic approaches to integrate security services into a client-server group communication system. The first approach (referred to as the *layered architecture*) places security services in a client library layered on top of the group communication system client library. The second approach entails housing some (or all) security services at the servers in order to obtain a more scalable solution (referred to as the *integrated architecture*).

In this chapter we present a scalable architecture for secure group communication, relying on a group key management protocol that is efficient, robust to process crashes and network partitions and merges, and protects confidentiality of the data even when long-term keys of the participants are compromised. We show how both the Virtual Synchrony and Extended Virtual Synchrony group communication semantics can be supported in the proposed architecture, discuss the accompanying trust issues and present experimental results that offer insights into its scalability and practicality. Because the scalability is achieved by integrating the group key management protocols in the server, we refer to this architecture as *integrated architecture*.

The rest of the chapter is organized as follows. First, we define our security goals in Section 6.1. Then, we propose three variants of an integrated security architecture in Section 6.2 and show the improved scalability of our integrated architecture in Section 6.3. We compare the advantages and drawbacks of each of the proposed architectures in Sections 6.4 and 6.5.

6.1 Security Goals

One of our main goals is to protect the data generated by a client and sent to a group, from being eavesdropped by both passive and active adversaries that are not current members of the group. Insider attacks are not relevant for this work since the confidentiality of the data relies on the secrecy of the group key, and any malicious insider can always reveal the group key or its own private key, thus compromising the communication.

The way the group key is computed is essential for the security of the system. A group key agreement protocol should provide: *Key Independence*, *Perfect Forward Secrecy* and *Backward/Forward Secrecy*. Informally, key independence means that a passive adversary who knows any proper subset of group keys cannot discover any future or previous group key. Forward Secrecy guarantees that a passive adversary who knows a subset of old group keys cannot discover subsequent group keys, while Backward Secrecy guarantees that a passive adversary who knows a subset of group keys cannot discover preceding group keys. Perfect Forward Secrecy means that a compromise of a member's long-term key cannot lead to the compromise of any short-term group keys. For a more precise definition of the above terminology, the reader is referred to Section 2.2.1 and (41), (56).

The key agreement protocol we use in our design is called Tree-Based Group Diffie-Hellman (74) (TGDH). It provides key independence and perfect forward secrecy; it was also proven secure with respect to passive outside (eavesdropping) adversaries (108). In addition, active outsider attacks – consisting of injecting, deleting, delaying and modifying protocol messages – that do not aim to cause denial of service are prevented by the combined use of timestamps, unique protocol message identifiers, and sequence numbers which identify the particular protocol execution.

Group members are authenticated when they connect to a server. Spread, the group communication system that is the subject of this work, provides a framework for authentication and access control.

Impersonation of group members is prevented by the use of public key signatures: every protocol message is signed by its sender and verified by all receivers.

Any form of attacks that aims to cause denial-of-service are not considered in this work.

6.2 Integrated Architecture

Early group communication systems were implemented as libraries, which meant that all distributed protocols were performed between all clients, per group. A substantial increase in performance and scalability was obtained by applying a client-server architecture to this model: a smaller number of servers run the expensive distributed protocols and, in turn, serve numerous clients.

Group key agreement protocols are, by nature, distributed and represent the most expensive security building block. Therefore, to improve the performance of the system in settings with multiple groups (or many clients) we amortize the cost of key management by placing the key agreement protocols at the servers and having the servers generating client group keys in a light-weight manner. This follows the integrated architecture model where security services are implemented at the server.

Since the server population is smaller and more stable than that of clients, server-based key agreement is both faster and less frequent. Specifically, the servers' shared secret key is refreshed only when network connectivity changes, and not when some client group changes. This results in fewer costly key refreshes in contrast to client-based key agreement because network connectivity changes are far less frequent than normal client group changes. Note that the shared server key can be vulnerable if it changes very infrequently and a security policy should impose additional refreshing operations, triggered, for example, by maximum elapsed time between successive key changes (time-out) or maximum volume of data exchanged (data-out).

Generating client group keys is much less costly in the integrated architecture, since, if no change occurs in the servers configuration, the cost of generating a new key for a group amounts to one keyed MAC (HMAC (52)) operation. When network connectivity does change (and so does the membership of the servers' group), the group key shared by the servers is refreshed using a full-blown group key agreement protocol. For this, we use the TGDH (74) protocol because of its superior performance.

The use of encryption for bulk data confidentiality results in decreased system throughput due to the extra consumption of CPU resources. Regardless of the location and particulars of the key management, bulk data encryption can be done by either clients or servers. In the following, we describe three integrated architecture variants that trade off encryption cost for complexity, overhead and group communication model support. We

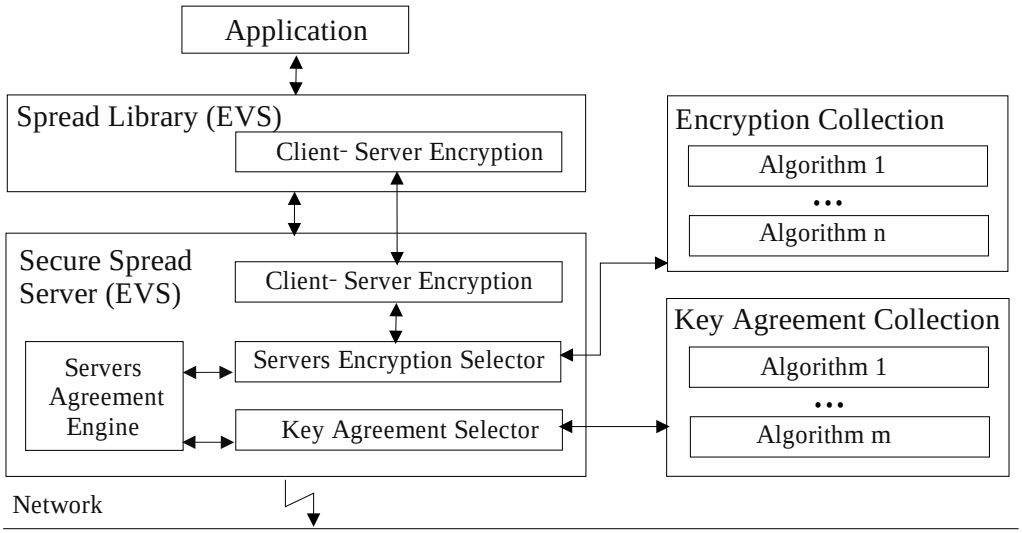


Figure 6.1: A Three-Step Client-Server Architecture for Spread

first discuss their different performance and security guarantees and then compare them to a layered approach.

6.2.1 Three-Step Client-Server

The most intuitive architecture is one derived from the the client-server model of the group communication system. The architecture can support both VS and EVS semantics at the expense of decreased (due to encryption) throughput. We refer to it as *Three-Step Client-Server*.

We note that the communication taking place in the system can be classified in two logical communication channels: client-server and intra-servers. The goal is to protect these two channels. Spread's architecture uses a TCP connection when a client connect remotely to a server. In this case, the best approach to protect the client-server communication is is using a standard two-party secure communication protocol, such as SSL/TLS (109). If a client connects to a server running on the same machine, Spread architecture uses IPC. In this case, no data protection is needed and client-server communication is not encrypted.

The intra-server communication channel is provided by a multicast protocol developed on top of UDP. In order to provide confidentiality of this communication, a block cipher encryption protocol based on a key shared by the servers is a good solution.

Figure 6.1 presents such an architecture. The Servers Agreement Engine detects

changes in the server group connectivity and for each connectivity change performs a key management protocol between servers. In addition, time-based or data-based key refresh can be enforced. As mentioned above, we use the TGDH (74) protocol for key management, due to its good performance and strong security properties.

Servers can distinguish between communication coming from peer servers and communication from the clients, and therefore, use the appropriate key in order to encrypt/decrypt the information.

One of the challenges with integrating a key agreement protocol into a group communication system is the interactions between the former and the membership protocol. Until the membership protocol completes, the key agreement protocol cannot run, since there is no fixed group of servers among which to perform key agreement. While the membership protocol is running, the set of known servers may change again (referred to as *cascaded membership*), and basic communication services between them may become unavailable.

To cope with this issue, the group key is provided only when the server group membership is stable and while the group communication membership protocol is not executing. This allows the key agreement protocol to run with its normal assumptions once the membership protocol completes, yet prior to notifying the client applications about the change. Thus, applications do not experience any change in semantics or the APIs (such as a new key message) but do experience an additional delay during each server membership change. (This is in order for the key agreement protocol to execute following the completion of the membership protocol.)

The membership protocol can be secured by using public key cryptography to encrypt and sign all membership messages, since the shared key is not available during its execution. The small number of messages sent during the membership algorithm and their small size, ensures that the overhead of public-private encryption can be tolerated.

The Three-Step Client-Server architecture allows individual policies for rekeying the server group key and the per-client SSL keys, as each is handled separately.

Once the master server group key is generated, the servers communication is protected by encryption using a key derived from it. The default protocol to encrypt communication between servers is Blowfish in CBC mode; however, the system supports any encryption algorithm in the OpenSSL (110) library, including AES (45).

The total end-to-end cost of sending an encrypted data message from one client

to another (both are connected to the Spread server remotely) includes six encryption and decryption operations: client encrypts the message and sends it over SSL to the server; server decrypts it and then re-encrypts using the server group key; servers that receive this message decrypt it and then re-encrypt it again using SSL for the receiving client; finally, each receiving client decrypts the message.

Note that the receiving servers need to encrypt the message separately for each remote client who needs to receive it. This is potentially a large number since each server can support about $1,000$ client connections. Thus, if more than one receiver is connected remotely on the same server, the load on the server will increase linearly with each remote receiver, since each remote receiver receives the same message encrypted separately on its own SSL connection. Local receivers do not require client-server encryption.

If two clients (sender and receiver) are executing on the same machine as the server that they connect to, then the cost of encryption under the Three-Step Client Server model reduces to one encryption by the sending server and one decryption by the receiving server.

6.2.2 Integrated VS

Although the Three-Step Client-Server architecture previously presented has the advantage that is less complex, it suffers from a decreased throughput due to the encryption load on the servers, and therefore is not recommended when clients connect remotely. The goal is to design an architecture that had a reasonable level of performance not only in key management, but also in throughput. This can be achieved if the encryption is pushed to the clients, which in turn requires client group keys.

We propose as second variant an architecture, referred to as *Integrated VS*, that supports the VS group communication model and combines the advantages of having a less expensive key management building block (by integrating it in the servers) with the advantage of having the encryption done in the client library. In this aspect, Integrated VS is similar to the layered architecture. The client groups are closed, i.e., a client needs to be a member of a group in order to send messages to that group. This design requires client groups keys. However, unlike the layered architecture where the key agreement was performed individually by each group, in this case, client group keys are generated by servers without involving costly key agreement protocols. Since the library operates in the VS model, in a manner similar to the layered architecture (see Section 5.1), a per-view

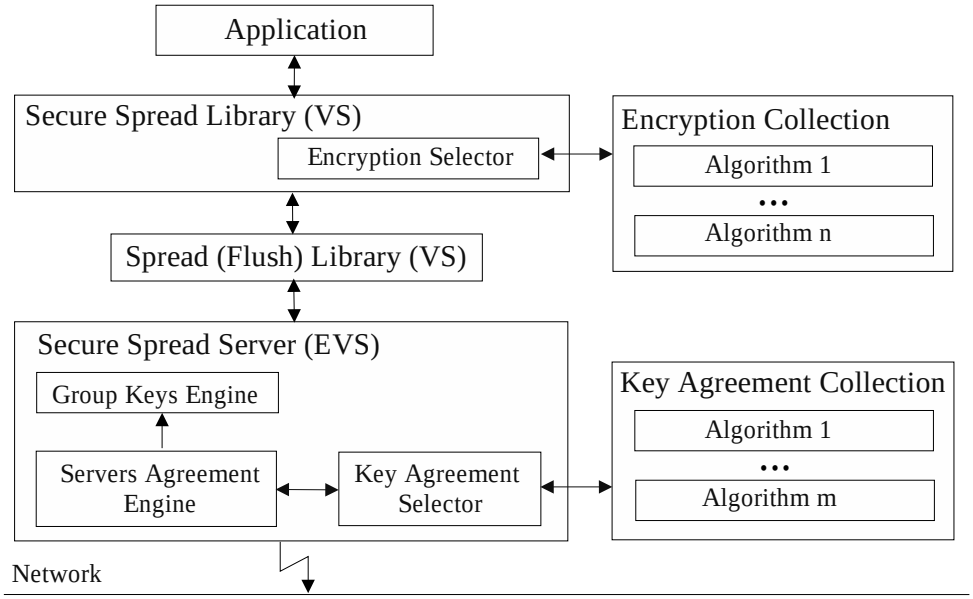


Figure 6.2: An Integrated VS Architecture for Spread

shared key associated with the group can be used to provide confidentiality. The key is refreshed by the servers when the group view changes.

Figure 6.2 depicts the Integrated VS architecture. The Servers Agreement Engine (SAE) initiates a key agreement protocol between the servers whenever it detects a change in server group connectivity. The Group Keys Engine (GKE) generates, for each group, a shared key whenever group membership changes. In case of a network connectivity change, the SAE is invoked first, followed by the GKE. The latter refreshes the key for each group that suffered changes in membership due to a change in server connectivity. The new group key is attached to the membership notification and delivered to the group. Client group keys are generated by the servers based on three values:

1. server group shared key K_s,
2. group name,
3. unique number that identifies the group view (list of members at a certain time).

The group key for group g in view i, uniquely identified by $view\_id(g, i)$ is

$$K_{g,i} = HMAC(K_s, g \| view\_id(g, i))$$

The shared server group key is computed in a manner identical with the one used by the Three-Step Client-Server architecture and can be refreshed as needed. The client group key is changed whenever a group event (join, leave, etc.) occurs. The new key is delivered within the secure membership message informing the clients about the group change. All client group members receive the same key for the same membership as a result of the VS semantics. If a key change is required because of the security policy (not caused by an underlying group membership change), the key refresh notification is delivered as an "artificial" group membership change. This is needed to preserve the semantic guarantees of VS that messages encrypted by a client with one key will be received by everyone while they also perceive as current key (have) that same key.

The encryption and decryption costs for Integrated VS consist of one encryption by the sender and multiple decryptions, one for each receiver. The worse case is when all receivers are situated on the same machine, whereas, the best case is when all receivers are running on distinct machines. In the latter case, decryption takes place in parallel. Once again, Blowfish is the preferred encryption algorithm.

6.2.3 Optimized EVS

Out of the architecture variants presented thus far, only Three-Step Client-Server supports the EVS model and open groups. As discussed in Section 3.2, EVS is faster, thus, it is desirable to have a secure group communication system supporting this model. The Three-Step Client-Server serves this purpose, but incurs a heavy encryption overhead when clients connect remotely to servers.

One method to alleviate the large number of encryption and decryption operations, is to have clients performing the encryption by using a shared per-view group key, in a manner similar to the Integrated VS architecture. However, unlike VS, EVS does not guarantee that all messages are delivered to receivers in the same view in which they were sent. Therefore, there might be messages that group members will not be able to decrypt as they do not have the key used to encrypt that message in the first place. Our next architecture variant addresses this issue.

In order to support EVS semantics and client message encryption, we developed an architecture that relies on the servers not only to generate client group keys, but also to "adjust" messages that are not encrypted with the current group key. The clients will

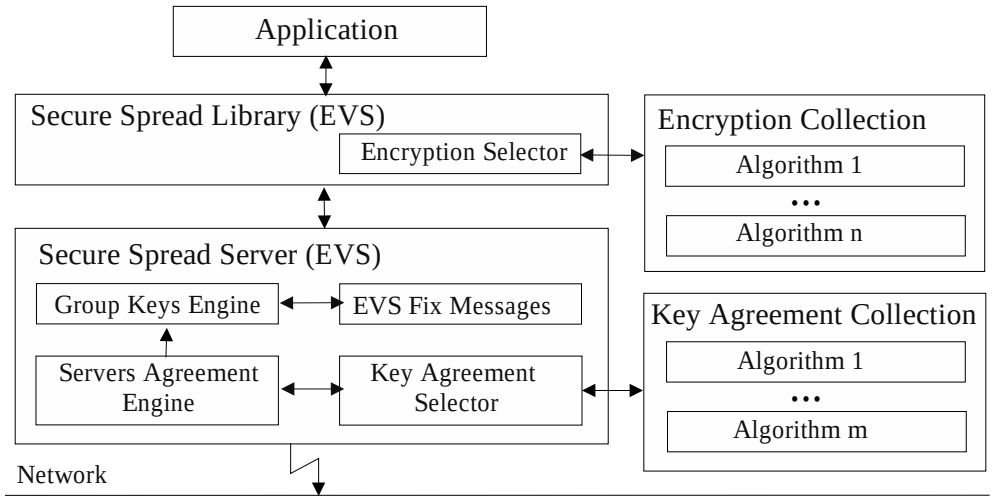


Figure 6.3: An Optimized EVS Architecture for Spread

operate without any disruption since the servers will guarantee that all messages delivered to the clients are encrypted with the current group key.

Figure 6.3 presents this architecture, referred to as *Optimized EVS*. The Servers Agreement Engine and Group Keys Engine perform key management of the servers' shared secret and client group keys, respectively. The method of generating client group keys is the same as the Integrated VS variant. The main change is that we add a new EVS-Fix-Messages module, that detects when a message for a certain group is encrypted with a key that is no longer valid. Each such message is decrypted and then re-encrypted with the current group key before being delivered to the clients. The clients, in turn, decrypt all group messages normally. TGDH is used as the server group key agreement protocol and Blowfish is used for data encryption.

The EVS-Fix-Messages module solves two problems:

1. Detects whenever a message is encrypted with the wrong key.
2. Determines the correct key to use for encrypting the message.

The first problem is addressed by having the sender include in each message a unique Key_id of the group key that was used to encrypt it. This Key_id is independently and randomly computed each time a new key is generated (it is also distributed along with each new client group key). However, since it does not provide integrity, but merely identifies the client group key, the Key_id can be relatively short, e. g., 32 bits. It is

transported in the un-encrypted portion of the message header.

To detect messages encrypted with an "old" key, the server stores each client group along with its Key_id. The server also tags one key as the "current" key for each client group. The current key is the key that matches the last membership (or key refresh) delivered to the group members. Then, before delivering a message to a client, it checks if the Key_id on the message matches that of the current key. If so, the message is immediately delivered. Otherwise, the message is decrypted with the appropriate stored "old" key and re-encrypted under the current key. Since the message stream delivered to each client is a reliable FIFO channel, the client eventually receives the message in the same view that the server expects it to.

Accumulating old keys and Key_ids *ad infinitum* is not a viable solution. Thus, old keys have to be periodically flushed from by each server. Two different expiration metrics can be used either alone or in concert: time-outs and key-outs. A time-out occurs when no message encrypted under a given key has been received for a certain length of time. A key-out takes place when some pre-set maximum number of keys-per-group is exceeded. Many combinations and variations on the theme are clearly possible.

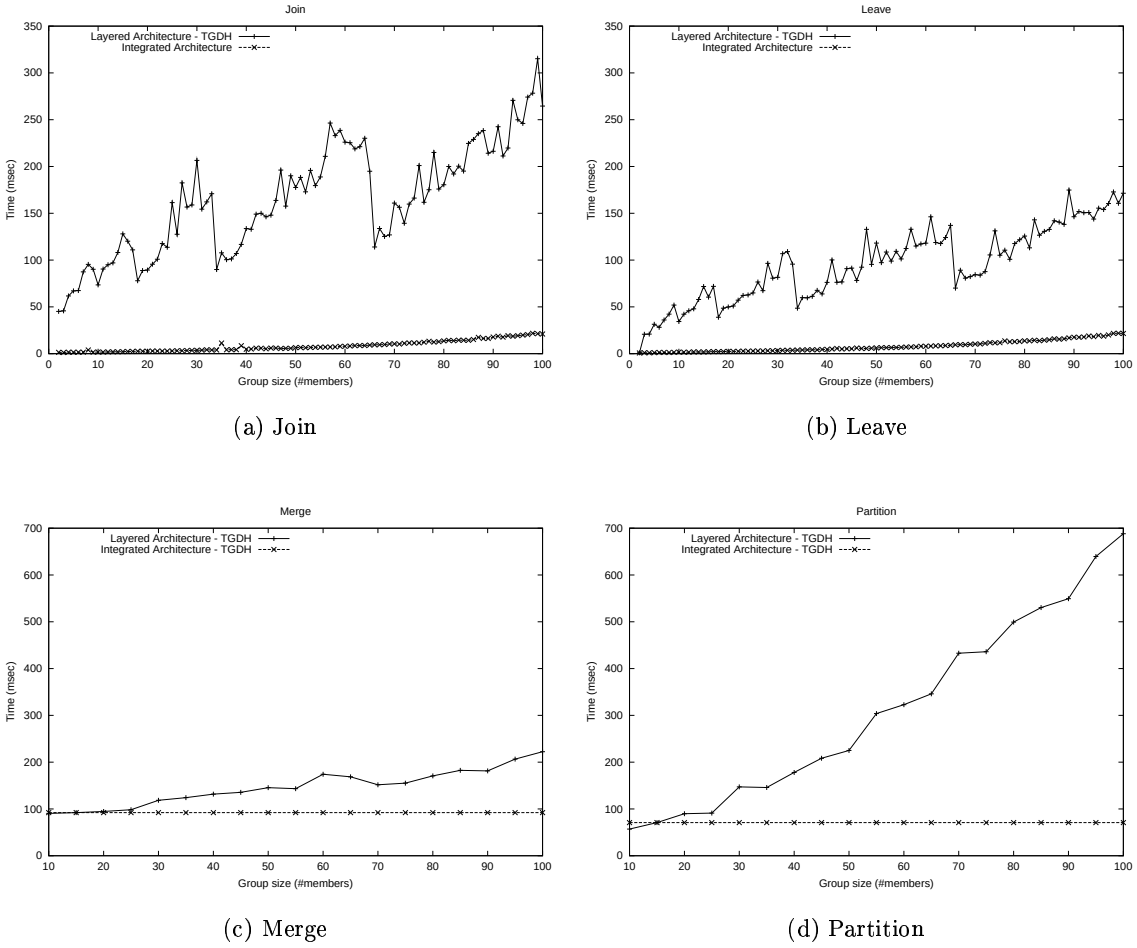
The choice of a key expiration methodology can affect the risk of a message being "indecipherable" even when the server, in theory, could have kept the required key.

6.3 Experimental Results

In this section we present experimental results for the group key management and data encryption building blocks. The experiments cover all architecture variants described in Section 6.2 measured in a local-area network environment and show the superior scalability of an integrated, over that of a layered, architecture.

6.3.1 Group Key Management

We now compare the cost of establishing a shared group key in a layered architecture and in an integrated architecture. For the layered architecture we chose the most efficient key agreement protocol that we have experimented with, TGDH (72). For the integrated architecture we also chose TGDH as the key agreement protocol between the servers.



(a) Join

(b) Leave

(c) Merge

(d) Partition

Figure 6.4: Key Agreement Cost: Layered Architecture vs. Integrated Architecture

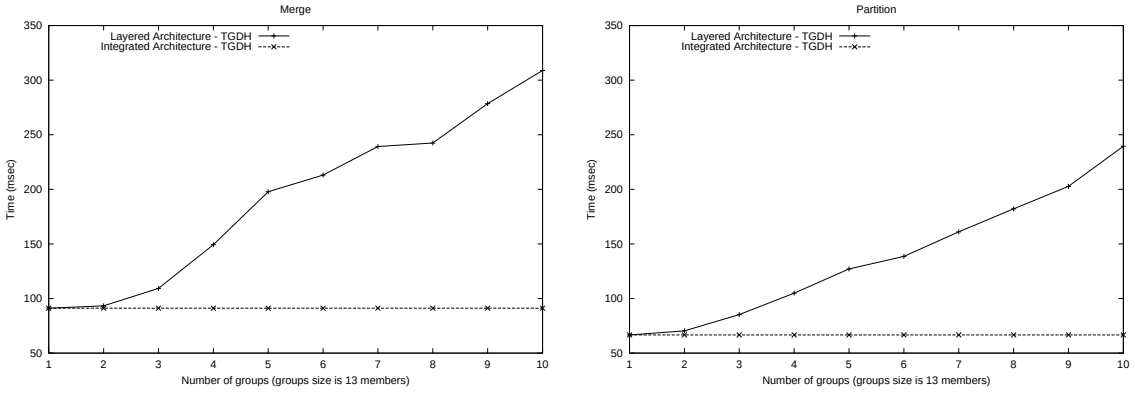
We used an experimental testbed consisting of a cluster of thirteen 667 MHz Pentium III dual-processor PCs running Linux. On each machine there is a Spread server running. Clients are uniformly distributed on the thirteen machines. Therefore, more than one process can be running on a single machine (which is frequent in many collaborative applications).

For the most common group changes, join and leave, the cost of establishing a new group key is reduced to almost the cost of the group communication membership protocol, since the servers can compute a new group key without performing any other key agreement protocol, just one HMAC operation is needed per group change. The results presented in Figure 6.4(a) and Figure 6.4(b) for the integrated architecture are for a VS group membership protocol. This is because the cost of the VS group membership protocol is in some sense the worst case: VS uses closed groups and it requires acknowledgments from each group member before changing the group membership. In the EVS case, the results for the integrated architecture will be much smaller.

In Figure 6.4(c) and Figure 6.4(d) we present the cost of establishing a secure membership for merge and partition. Remember that such a group event is triggered by a network connectivity change which determines a modification in the servers configuration, or by a server crash. In this case, a new servers' key needs to be computed by the servers, and only then the client group keys are computed. In Figure 6.4(c) and Figure 6.4(d) we present the cost of establishing a secure group membership for a test scenario where the servers are partitioned in half and then brought back together.

As it can be seen in Figures 6.4(c) and 6.4(d) the cost of the key management for the integrated architecture is slightly higher than in the case of join and leave because of the cost of the key agreement protocol performed between servers. However, since the number of servers is much smaller than the number of clients, the impact of the key agreement protocol is less significant. The cost of the secure membership decreases from about 220 milliseconds, to about 90 milliseconds where the size of the group after partition is 100 users, in case of a merge and from about 680 milliseconds to about 60 milliseconds for a partition, where the size of the group before partition is about 100 members.

The above results were obtained for a scenario when only one group exists in the system. In practice, this is not the case. When more than one group exists in the system and a change in the servers' configuration that affects more than one group occurs, the layered architecture performs a key agreement protocol for each of the existing groups affected by



(a) Merge

(b) Partition

Figure 6.5: Scalability with Number of Groups

the change. For the integrated architecture, there is only one small scale key agreement performed between servers, and then a number of HMAC operations equal with the number of groups affected by the change.

Figure 6.5 shows the average cost of recomputing a shared key for all groups, when more than one group exists in the system. All the groups have the same number of clients, 13 respectively. We chose this number, because this is also the number of the servers in our configuration. Even in this favorable setup for the layered architecture (small size groups), the integrated architecture scales much better than the layered architecture when the number of groups in the system increases. Based on the results we present in Figure 6.5 we estimate that even with a very small group size (13 in our case), it will take more than 4 seconds to refresh the key for 200 groups in a layered architecture, while it will take about 50 times less to perform the same operation for an integrated architecture.

6.3.2 Data Encryption

Besides the membership service, group communication systems provide reliable and ordered message delivery, therefore an important metric used to characterize the performance of the system is data throughput.

We evaluated the throughput of the secure system in a 100 MBits local area network. The testbed consists of a cluster of 13 667 MHz dual-processor machines running Linux. On each machine there is a Spread server running.

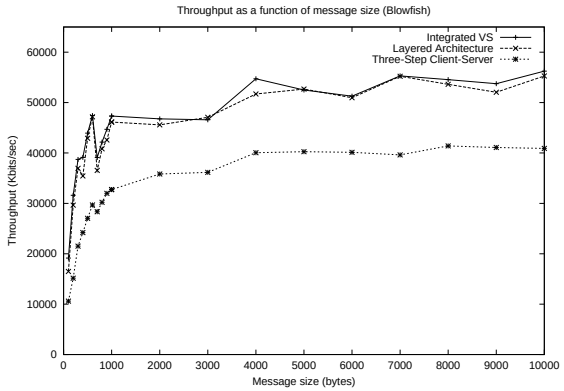


Figure 6.6: Data Throughput

We are interested in the throughput of the system, for the following architectures: the Layered Architecture, the Integrated VS and the Three-Step Client-Server, in a local area network. We consider a scenario where clients connect to servers running locally, so in the Three-Step Client-Server setup, encryption is performed only between servers.

In Figure 6.6 we present the data throughput for the above setups, when there is only one sender multicasting to a group, only one group in the system and the encryption algorithm used is Blowfish.

As expected, the results for the Integrated VS are similar with the results for the Layered Architecture, because in both models encryption and decryption are performed by the clients. The Three-Step Client-Server exhibits about 0.66 of the throughput achieved in the other two models. The major reason for this decrease is due to the fact that in the Three-Step Client-Server case both headers and data are encrypted and the message delivery protocol employed by our system is not able to decide if it needs to process a message or not, without first decrypting it. Additional cost is due to the fact that the maximum message size exchanged by the servers is about the size of an Ethernet frame (minus the UDP protocol header), so a message of large size that gets encrypted in a client in only one encryption operation, translates into a number of encryption operations in the server. We note that for the Three-Step Client-Server since the encryption operation takes place at the data link layer, the servers encrypt not only client data, but also control information, so this model provides a stronger service than the other two models.

This experiment only had one sender and that server was the bottleneck. In cases where several servers are sending messages, this cost will be amortized and the throughput

will considerably increase (by a bit less than a factor of 2).

We did not include results for the Three-Step Client Server architecture when clients connect remotely, but from the results in Figure 6.6 we can extrapolate that the achieved throughput in this case will be much smaller, and therefore unacceptable. In case of the Optimized EVS architecture, the throughput will be similar with the one for the Integrated VS if the servers membership does not change, and smaller, but still better than the Three-Step Client-Server performance, when changes occurs, so some messages will need to be decrypted and re-encrypted under new keys.

The drop in throughput for all of the methods (as seen in Figure 6.6) at a message size of around 700 bytes is actually a positive thing. Up to 700 bytes Spread is able to pack multiple messages into one network packet, thus paying less per packet and increasing throughput considerably. Above 700 bytes, that optimization can not be employed because of the Ethernet maximum packet size.

6.4 Layered Architecture vs. Integrated Architecture

The layered and each of the integrated architectures have benefits and limitations. In this section we compare a layered architecture approach to an integrated architecture approach, when providing security services to a group communication system. We compare them by investigating the following aspects: trust, key management scalability, impact of the compromise of the shared secret and, complexity and ability to efficiently support other group services.

The layered architecture has the advantage that no trust is put into anything outside of the end user's control with respect to protecting the data generated by a client. The client needs to trust the servers with respect to the membership service and ordered and reliable delivery, but these are outside the scope of our security goals for this work. The compromise of a group key, does not affect the security of the rest of the groups in the system, since each group is running its own protocol and computes its shared key independently of the other groups. In addition, this architecture is less complex and easier to develop, allowing us to explore the inter-relationship between key agreement and group communication [1]. However, this model, due to the security strong, but expensive key

[1] For example, we used this architecture to design robust contributory protocols, resilient to any sequence of group changes, possibly cascading.

agreement protocols we used, has limited scalability, to no more than 100 members for the protocol with the best performance.

The integrated architectures we proposed overcome the key management scalability problem by using a secret key shared by the servers, and thus putting more trust in the servers. However, the security of the groups relies on the security of the servers shared key which is used in generating the group keys. If the servers' key is compromised, the security confidentiality of the communication of all the groups in the system is compromised, as opposed to the layered model where in order to compromise all the groups in the system, an attacker needs to compromise the shared key for each group.

An integrated architecture is also more appropriate for providing other security services such as client authentication upon connection and access control to perform group specific operations. A security policy can be easily configured and enforced by an administrator controlling a server configuration file.

Another advantage of an integrated architecture vs. a layered architecture regards the protection of the control information messages exchanged by the servers. If designed appropriately, an integrated architecture can provide this service based on the secret key shared between servers, while the layered architecture can not.

Choosing the most appropriate architecture depends on the desired scalability and trust guarantees. An integrated approach scales better, but the security of all groups relies on one key; a layered architecture scales worse, but the security of a group is independent of the security of the rest of the groups and gives more control to the client.

6.5 Integrated Architectures Variants Comparison

As we discussed in Section 6.2 there is no one-size-fits-all architecture solution that will perform the best in all possible environments, under both VS and EVS group communication semantics. Therefore, we proposed three integrated architecture variants that trade off encryption cost for complexity, overhead and group communication model support. In this section we compare them by focusing on the group communication model supported, design and implementation of the key management building block (do they use or not client group keys) and the place where the encryption and decryption operations are performed performed (only between clients, only between servers, or between client and server).

	Group Keys	Encryption	Group Comm. Model
Three-Step Client-Server	No	Client-Server, Server-Server	VS and EVS
Integrated VS	Yes	Client-Clients	VS
Optimized EVS	Yes	Client-Clients mostly	EVS

Table 6.1: Secure Group Communication Integrated Architectures

Table Table 6.1 summarizes their features. The Three-Step Client-Server approach does not use client group keys, but requires a client to share a key with the server it connects to. The approach is very appealing because is uses a less complex key management mechanism. However, it is expensive in encryption and decryption operations when clients connect to servers remotely. If clients connect to servers locally this is the best architecture since theoretically it only requires one encryption/decryption of each message and it can easily protect not only client data, but also the control information exchanged by the servers. Note, that depending on the implementation, even when clients connect locally, more than one one encryption/decryption of each message can take place as discussed in Section 6.3.2. This architecture supports both the VS and the EVS semantics.

Both the Integrated VS and the Optimized EVS architectures use client group keys generated by servers. Our experimental results in Section 6.3 show that the scalability of the system is improved substantially with respect to the layered architecture.

For all the integrated architectures the confidentiality of the data ultimately relies on the secret shared by the servers.

The smallest encryption overhead is exhibited by the Integrated VS approach The Optimized EVS solution has the same encryption cost as the Integrated VS if the group membership is stable. When membership changes and there are messages not delivered in the membership they were sent in, four additional encryption/decryption operations per message are performed, to decrypt the messages encrypted with an old key and re-encrypt them under the current key. The encryption overhead incurred by the Three-Step Client-Server approach, even when clients connect locally, is larger that that of Integrated VS. However, it provides a stronger service since it also protects the information exchanged by the servers.

6.6 Conclusions

In this chapter we presented several secure integrated architectures for client-server group communication system, discussing their different performance and security guarantees. The experimental results we present demonstrate the increased scalability of integrated approaches over layered approaches.

Chapter 7

Conclusions

This work investigated how security mechanisms can be integrated with reliable group communication systems, such that the resulted secure group communication system does not suffer from a severe degradation of the performance and preserves its fault-tolerance properties. In this chapter we summarize the contributions of this work and present our conclusions.

With the increased use of collaborative applications such as voice- and video-conferencing, white-boards, distributed simulations, games and replicated servers of all types, designing protocols that enable robust, secure and reliable delivery of critical information and services is essential to reduce the impact of security breaches on critical infrastructure. Since group communication systems are a powerful tool in designing collaborative applications, enhancing group communication systems with security services provides a strong platform for easily developing secure and efficient secure collaborative services.

In spite of the progress in the cryptography community in designing secure and scalable protocols to provide specific security services, such as data secrecy, data integrity, entity authentication and access control, to multicast and group applications, less emphasis has been put on how to integrate security protocols with modern, highly efficient group communication systems and address problems arising as a result of the integration. This work fills this gap by focusing on designing efficient secure group communication systems.

Key management protocols represent a critical building block of a secure group communication system. While taking advantage of the advances in the cryptographic research community in designing secure key management protocols, this work points out the limitations and incorrect assumptions that these protocols usually have about network and

group services and proposes practical solutions. In particular, we identified the effect that key management has on the robustness of the system, and showed how multi-round group key management protocols can be made fault-tolerant, by using the membership and reliable and ordered message delivery services of group communication systems. A major contribution of this work is the design of the first robust contributory group key agreement protocol, for a general group communication service supporting Virtual Synchrony group communication semantics.

As opposed to previous work, we investigated the use of key agreement protocols as building blocks for our security services. Such protocols have the advantage that they provide strong security services such as key independence and perfect forward secrecy. The security of our system relies on a group key management protocol that is efficient, robust to process crashes and network partitions and merges, and protects confidentiality of the data even when long-term keys of the participants are compromised.

Besides the theoretical contributions, this work maintained a practical aspect by focusing on securing Spread, a local and wide area group communication system. Therefore, maintaining the performance of the membership and message delivery services is essential. The actual costs associated with group key management have been poorly understood in the past. Consequently, there has been a dual undesirable tendency: on the one hand, adopting suboptimal security for reliable group communication, while, on the other hand, constructing excessively costly group key management protocols. We considered a number of key agreement protocols and not only analyzed them theoretically, but also implemented them in the same framework and evaluated them in experiments conducted on both local and wide area networks. The experiments considered scenarios describing all types of group changes, including partitions and merges. The software resulted from this work, which we refer to as, layered architecture, was publicly made available and is used by other research groups, both in academic and industry environments.

The main focus of this work was designing a high-performance security architecture for a client-server group communication system. In particular, we focused on designing a security architecture for Spread, under two well-known group communication semantics: Virtual Synchrony and Extended Virtual Synchrony. Both models support network partitions and merges and present their particular challenges. Contributory key agreement protocols when used in a layered architecture have limited scalability. We overcame this by using an integrated approach that relies on contributory group key management in a light-

weight/heavy-weight group architecture such that the cost of key management is amortized over many groups, while each group has its own unique key.

When designing an efficient architecture supporting the Virtual Synchrony model, we took advantage of the fact that Virtual Synchrony provides a form of synchronization between the group membership changes and data messages delivery. Our approach was to use of a shared group key per view, securely refreshed upon each membership change. Data confidentiality can be relatively easily provided in a system supporting Virtual Synchrony because the synchronization between membership notifications and message delivery (guaranteed by the Sending View Delivery Property defined in Section 3.2.1) ensures that any message is guaranteed to be encrypted, delivered and decrypted in the same group view and, hence, with the same current key.

Although it provides a more efficient and relaxed model, the Extended Virtual Synchrony is more challenging when providing security services, because there is no synchronization between membership notifications and data delivery to the clients. However, there is some knowledge about what was the application group membership when the message was generated (and also encrypted) and the group membership when the message will be delivered (and also decrypted). We provided also solutions for handling security for systems supporting Extended Virtual Synchrony, by using information shared by the group communication servers that provide the membership and message ordering and delivery services.

We proposed three variants of an integrated architecture that trade off encryption cost for complexity and group communication model support. We showed how both group communication semantics could be supported in the proposed architecture, discussed the accompanying trust issues and presented experimental results that offered insights into its scalability and practicality.

Appendix A

Key Management Protocols

A.1 Group Diffie-Hellman Protocol

GDH IKA.2 is a contributory key agreement protocol that extends the two-party Diffie-Hellman protocol to groups. It consists of a suite of protocols, each of them specifying how the group key will be refreshed depending of the event that occurred in the group. More specifically, it defines how the group key changes when a new members joins the group (join), when more than one member becomes part of the group (merge), one member leaves the group (leave), more than one member leaves the group (partition).

The group key K_{group} is in the form $K_{group} = g^{N_1 N_2 \ldots N_n}$, where N_i is the contribution of member i to the group key K_{group}. The protocol is designed based on the idea that the shared key is never transmitted over the network, even in encrypted form. Instead, a set of partial keys, $K_i = g^{\frac{N_1 N_2 \ldots N_i \ldots N_n}{N_i}}$ (that are used by individual members to compute the group secret) is sent. One member of the group – referred as group controller– is charged with the task of building and distributing this list. The controller is not fixed and has no special security privileges.

In the following we describe only how GDH handles merge and partition, since join and leave can be seen as special cases of merge and partition, respectively.

The merge protocol runs as follows. When a merge event occurs (see Algorithm 10), the current group controller generates a new key token by refreshing its contribution to the group key and then passes the token to one of the new members. When the new member receives this token, it adds its own contribution and passes the token to the next

Algorithm 10 GDH Merge Protocol

Assume that k members are added to a group of size n.

Step 1:

Mmeber M_n:

generates a new exponent r'_n,

computes $g^{r_1 \cdots r_{n-1} r'_n}$,

unicasts the message to M_{n+1}.

Step $j+1$ for $j \in [1, k-1]$:

New merging member M_{n+j}:

generates an exponent r_{n+j},

computes $g^{r_1 \cdots r_n{}' \cdots r_{n+j}}$,

forwards the result to M_{n+j+1}.

Step $k+1$:

Upon receipt of the accumulated value, M_{n+k}:

broadcasts the accumulated value to the entire group.

Step $k+2$:

Upon receipt of the broadcast, every member $M_i, \forall i \in [1, n+k-1]$:

computes $g^{r_1 \cdots r_n{}' \cdots r_{n+k-1}/r_i}$,

sends it back to M_{n+k}.

Step $k+3$:

After collecting all the responses M_{n+k}:

generates a new exponent r_{n+k},

produces the set $S = \{g^{r_1 \cdots r_n{}' \cdots r_{n+k}/r_i} | \forall i \in [1, n+k-1]\}$,

broadcasts it to the group.

Step $k+4$:

Upon receipt of the broadcast, every member $M_i, \forall i \in [1, n+k]$:

computes the key $K = (g^{r_1 \cdots r_n{}' \cdots r_{n+k}/r_i})^{r_i} = g^{r_1 \cdots r_n{}' \cdots r_{n+k}}$.

new member[1]. Eventually, the token reaches the last new member. This new member, who is slated to become the new group controller, broadcasts the token to the group without adding its contribution. Upon receiving the broadcast token, each group member (old and new) factors out its contribution and unicasts the result (called a factor-out token) to the new group controller. The new group controller collects all the factor-out tokens, and adds its own contribution to each of them. Each such a factor-out token on which the group controller added its contribution represents a partial key. Once all the partial keys were computed, the list of partial keys is broadcasted to the group. Every member can then obtain the group key by factoring in its contribution from the corresponding partial key from the list.

Algorithm 11 GDH Partition Protocol

Assume that a set L of members is leaving a group of size n.

Step 1:

The controller M_d:

generates a new exponent $r_d{}'$,

produces the set $S = \{g^{r_1 \cdots r_d{}'/r_i} | M_i \notin L\}$,

broadcasts it to the remaining group.

Step 2:

Upon receipt of S, every remaining member M_i, $\forall i \notin L$:

computes the key $K = (g^{r_1 \cdots r_d{}'/r_i})^{r_i} = g^{r_1 \cdots r_d{}'}$

In case a leave event occurs in the group, the protocol runs as follows (see Algorithm 11). The group controller removes their corresponding partial keys from the list, refreshes its share from each partial key in the list to ensure the freshness of the new computed key and broadcasts the list to the group. Each remaining member can then compute the shared key from their partial key. In case among the leaving members there is the group controller, the last remaining member becomes the group controller of the group. Note that since the list is broadcasted to the group any from the remaining members can take on the task of the group controller.

[1]The new member list and its ordering is decided by the underlying group communication system; Spread in our case. The actual order is irrelevant to GDH.

A.2 Centralized Key Distribution Protocol

Centralized Key Distribution ()CKD) protocol is a simple centralized group key distribution scheme. The group key is not contributory, but it is always generated by one member, namely, the current group controller. We use the term *current* to mean that, even in the CKD protocol suite, a controller can fail or be partitioned out thus causing the controller role to be reassigned to another member.

Algorithm 12 CKD Merge Protocol

Assume that k members are added to a group of size n. M_1 is the group controller.

Step 1:

M_1:

selects random $r_1 \bmod q$ (this selection is performed only once),

computes $\{g^{r_1} \bmod p \mid j \in [1,k]\}$,

sends it to $\{M_{n+j} \mid j \in [1,k]\}$.

Step 2:

For each $j \in [1,k]$, M_{n+j}:

selects random $r_{n+j} \bmod q$,

sends it to $M_{n+j} : g^{r_{n+j}} \bmod p$.

Step 3:

M_1:

selects a random group secret K_s,

computes ${K_s}^{g^{r_1 r_i}} \bmod p$,

sends it to $M_i, \quad \forall i \in [2, n+k]$.

Step 4:

From the broadcast message, every member can compute the group key.

The group controller establishes a separate secure channel with each current group member by using authenticated two-party Diffie-Hellman key exchange. Each such key stays unchanged as long as both parties (controller and regular group member) remain in the group. The controller is always the oldest member of the group. Whenever the group membership changes, the group controller generates a new secret and distributes it to the group using the long-term pairwise key. In case of a merge (see Algorithm 12), the controller

also establishes a secure channel with each new member.

Algorithm 13 CKD Partition Protocol

Assume that a set L of members is leaving a group of size n.

Step 1:

The controller M_1:

selects a random group secret K_s,

computes ${K_s}^{g^{r_1 r_i}} \bmod p$, $M_i \notin L$,

sends it to M_i, $M_i \notin L$.

Step 2:

From the broadcast message, every member can compute the group key.

When a partition occurs (see Algorithm 13), in addition to refreshing the key and distributing it via secure channels to the remaining members, the controller discards the long-term key it shared with each leaving member. A special case is when the group controller itself leaves the group. In this case, the oldest remaining member becomes the new group controller. An additional cost is incurred since before distributing the key, the new group controller must first establish secure channels with all of remaining group members.

A.3 Tree Group Diffie-Hellman Protocol

Tree Group Diffie-Hellman (TGDH) is an adaptation of key trees (111; 112) in the context of fully distributed, contributory group key agreement. TGDH computes a group key derived from the contributions of all group members using a binary tree.

The tree is organized in the following manner: each node $\langle l, v \rangle$ is associated with a key $K_{\langle l,v \rangle}$ and a corresponding blinded key $BK_{\langle l,v \rangle} = g^{K_{\langle l,v \rangle}} \bmod p$. The root is associated with the group and a leaf with a member. The key at the root node represents the group key shared by all members, and a key at the leaf node represents the random session contribution by a group member. Each internal (non-leaf) node has an associated secret key and a public blinded key. The secret key is the result of a Diffie-Hellman key agreement between the node's two children. Every member knows all the keys on the path from its leaf node to the root, as well as all blinded keys on the key tree.

The protocol relies on the observation that every member can compute a group

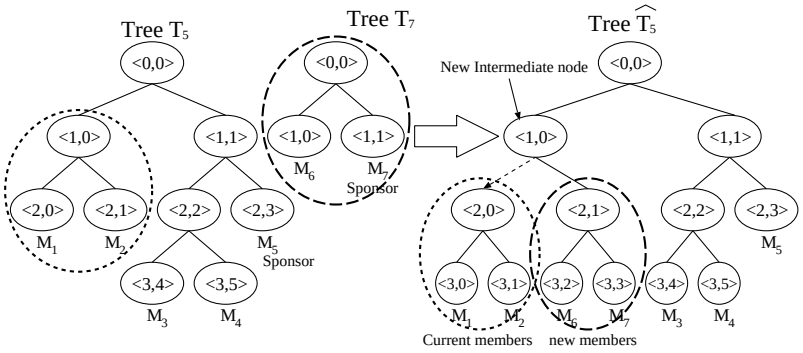


Figure A.1: TGDH Merge Operation

key when all blinded keys on the key tree are known. After any group membership change, in order to preserve backward and forward secrecy, every member unambiguously adds or removes some nodes related with the event, and invalidates all keys and blinded keys related with the affected nodes. A special group member – the sponsor– , then, takes on a role to compute keys and blinded keys and to broadcast the key tree[2] to the group. If a sponsor could not compute the group key, then the next sponsor comes into play. Eventually, some sponsor will compute the group key and all blinded keys, and broadcast the entire key tree to facilitate the computation of the group key by the other members of the group.

Algorithm 14 TGDH Merge Protocol

Round 1:

request for merge by both groups

$$M_{s_1} \xrightarrow{T_{s_1}(BK^*_{s_1})} \{M_1, ..., M_{n+k}\}$$
$$M_{s_2} \xrightarrow{T_{s_2}(BK^*_{s_2})} \{M_1, ..., M_{n+k}\}$$

Round 2:

update tree $T_{s'}$ to get $\widehat{T}_{s'}$ and broadcast it

$$M'_s \xrightarrow{\widehat{T}_{s'}(BK^*_{s'})} \{M_1, ..., M_{n+k}\}$$

When a merge event happens (See Figure 14), each sponsor (the rightmost member of each group) broadcasts its tree information to the merging sub-group after refreshing its

[2]The keys are never broadcasted.

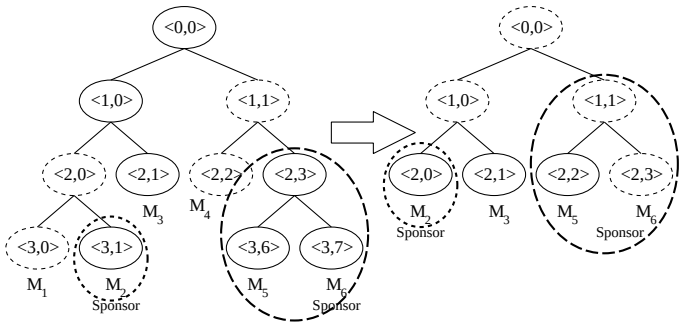


Figure A.2: TGDH Partition Operation

session random and blinded keys. Upon receiving this message, all members uniquely and independently determine the merge position of the two trees.[3] As described above, all the keys and blinded keys on the path from the merge point to the root node are invalidated. The rightmost member of the subtree rooted at the merge point becomes the sponsor of the key update operation. The sponsor computes all the keys and blinded keys and broadcasts the tree with the blinded keys to all the other members. All members now have the complete set of blinded keys, which allows them to compute all keys on their key path. Figure A.1 shows an example of the merge protocol. Members M_6 and M_7 are added to a group consisting of members M_1, M_2, M_3, M_4 and M_5.

Algorithm 15 TGDH Partition Protocol

for all M_{s_i}:

$M_{s_i} \xrightarrow{\widehat{T}_{s_i}(BK^*_{s_i})} \{M_1, \ldots, M_n\}$

Round 1 to $h' + 1$:

update tree T_{s_i} to get $\widehat{T}_{s_i}$

Following a partition, the protocol runs as follows (see Figure 15). In the first round, each remaining member updates its view of the tree by deleting all leaf nodes associated with the partitioned members and (recursively) their respective parent nodes. To prevent re-use of an old group key, one of the remaining members changes its key share. To this end, in the first protocol round, the shallowest rightmost sponsor changes its share. Each sponsor then computes the keys and blinded keys as far up the tree as possible, and,

[3] Authors' heuristic is to choose the joining node as the rightmost "shallowest" node, which does not increase the height. For more details, see (72)

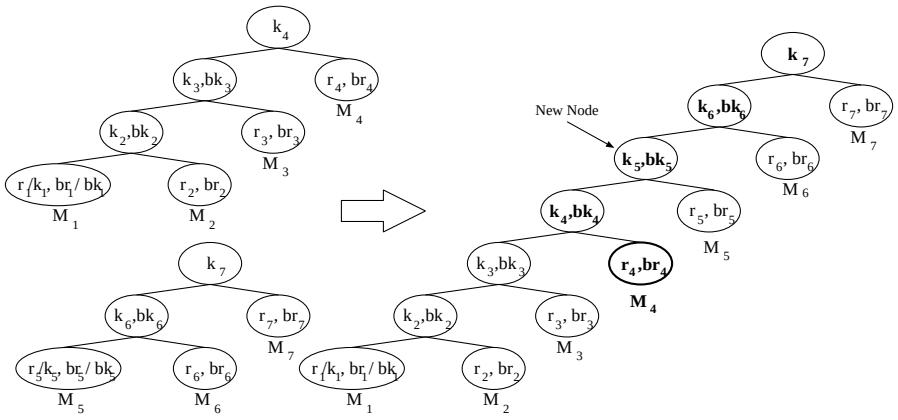


Figure A.3: STR Merge Operation

then, broadcasts the set of new blinded keys. Upon receiving the broadcast, each member checks whether the message contains a new blinded key. This procedure iterates until all members obtain the group key. Figure A.2 shows a partition example where members M_1 and M_4 are removed from the group.

A.4 STR Protocol

STR (74) is basically an "extreme" version of TGDH, where the key tree structure is completely imbalanced or stretched out.

Like TGDH, the STR protocol uses a tree structure that associates the leaves with individual session random contributed by the group members. Every internal (non-leaf) node has an associated secret key and a public blinded key. The secret key is the result of a Diffie-Hellman key agreement between the node's two children. The group key is the key associated with the root node.

The merge protocol runs in two rounds. In the first round, each of the two sponsors (topmost members or right children of the respective root nodes in each tree) exchange their respective key trees containing all blinded keys after refreshing its session random and computing keys and blinded keys up to the root node. The highest-numbered member of the larger tree becomes the sponsor of the second round in the merge protocol (see Figure A.3). Using the blinded session random of the other group, the sponsor computes every (key, blinded key) pair up to the intermediate node just below the root node. It then broadcasts

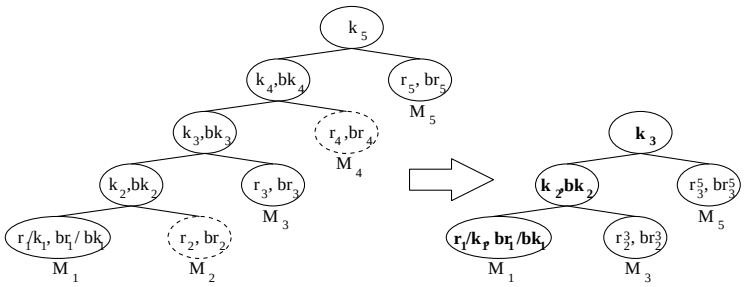


Figure A.4: STR Partition Operation

the key tree with the blinded keys and blinded session random to the other members. All members now have the complete set of blinded keys which allows them to compute the new group key.

In a partition, the sponsor is the lowest-numbered remaining member. After deleting all leaving nodes (see Figure A.4), the sponsor refreshes its session random, computes keys and blinded keys up the tree terminating with the root key. It then broadcasts the updated key tree containing only blinded values. Each member can compute then the group key.

A.5 BD Protocol

Unlike the other protocols discussed thus far, the Burmester-Desmedt (BD) protocol (61) is independent of the type of group membership change. Furthermore, it has no sponsors, controllers or any other members charged with any special duties.

The main idea in BD is to distribute the computation among members, such that each member performs only three exponentiations. This is achieved by using two communication rounds, each of them consisting of n broadcasts. Figure 16 depicts the protocol.

Algorithm 16 BD Protocol

Assume a group of size n.

Step 1:

Each member M_i:

selects random $r_i \bmod q$,

computes $Z_i = g^{r_1} \bmod p$,

broadcasts the message to the group.

Step 2:

Each member M_i, after receiving Z_{i-1} and Z_{i+1}:

computes $Xi = (Z_{i+1}/Z_{i-1})^{r_i} = g^{r_{i+1}r_i - r_i r_{i-1}}$,

broadcasts it to the group.

Step 3:

Each member M_j, after receiving all $X_i, i \neq j$:

computes

$K = K_j = (Z_{j-1})^{nr_j} {X_i}^{n-1} ... X_{i+(n-2)} \bmod p = g^{r_1 r_2 + r_2 r_3 + ... + r_{n-1} r_n} \bmod p.$

Appendix B

Cliques GDH API

CLQ_CTX *clq_first_member*(PROCESS_NAME p, GROUP_NAME g)

This function is called by the first member of the group. It takes as arguments the name of the process calling it, p, and the name of the group to be created, g, and it returns the CLQ_CTX associated for that group.

CLQ_CTX *clq_new_member*(PROCESS_NAME p, GROUP_NAME g)

This function is called by a new member, that wants to be added to a group. It takes as arguments the name of the process calling it, p, and the name of the group it wants to become part of, g, and it returns the CLQ_CTX associated for that group.

VOID *clq_destroy_ctx*(CLQ_CTX ctx)

This function cleans a CLQ_CTX structure specified by *ctx*.

KEY *clq_update_ctx*(CLQ_CTX ctx, MSG key_list_msg)

This function is called by all participants to compute the group key. It takes as argument the CLQ_CTX associated with the group, and the key list message *key_list_msg*, and it returns the key.

MSG *clq_update_key*(CLQ_CTX ctx, PROCESS_NAME[] merge_set, MSG partial_token_msg)

First member that calls the function, it passes the context associated with that group, *ctx*, the list of the new members that will be added, *merge_set*. For subsequent calls, the list of processes that will be added should be NULL, and the message m is the partial token message received.

PROCESS_NAME *clq_next_member*(CLQ_CTX ctx)

The function returns the name of the next process that a partial token should be passed to. The first member that generated a partial token specified the list of the new members added to the group. That list is maintained in the CLQ_CTX structure. It takes as argument the group associated context, *ctx*.

MSG *clq_factor_out*(CLQ_CTX ctx, MSG final_token_msg)

This function is called by all group members to factor out their contribution from the token. It takes as arguments the CLQ_CTX associated with the group, *ctx*, and the final token message, *final_token_msg*, and it returns a factor out message.

PROCESS_NAME *clq_get_new_gc*(CLQ_CTX ctx)

This function returns the name of the process that is the new group controller, given a CLQ_CTX structure, *ctx*.

MSG *clq_merge*(CLQ_CTX ctx, MSG fact_out_msg, MSG key_list_msg)

This function builds the set of partial keys. It takes as argument a CLQ_CTX, *ctx* and two messages, the factor out message, *fact_out_msg*, and the key list message, *key_list_msg*. The output is a modified message list message. In order to check if the list is ready, the function *ready*(MSG key_list_msg) should be called.

KEY *clq_extract_key*(CLQ_CTX ctx)

This function returns the current valid key for a group, given the associated CLQ_CTX, *ctx*.

MSG *clq_leave*(CLQ_CTX ctx, PROCESS_NAME[] leave_set)

This function is used to refresh the key when a set of members left the group. The function removes the contributions of the members that left from the list of partial keys and refreshes each item of the list. It takes as arguments the associated CLQ_CTX for that group, *ctx*, and the list of the processes that left the group, *leave_set*, and it generates a key list message.

MSG *clq_new_update_key*(CLQ_CTX ctx, PROCESS_NAME[] leave_set, PROCESS_NAME merge_set, MSG partial_token_msg)

This is a modified version of the clq_update_key function, that can handle additive and subtractive group events in one step. First it removes the processes that left the group, then it handles the list of processes that must be added to the group. It takes as arguments the CLQ_CTX associated with the group, *ctx*, the set of processes that left, *leave_set*,the set of processes that must be added to the group, *merge_set*, and a partial token message, *partial_token_msg*. The list of processes that left the group can be NULL. First member that calls the function, it passes the list of the new members that will be added. For subsequent calls, the list of processes that will be added or removed should be NULL, and the message m is the partial token message received.

Bibliography

[1] L. Lamport, "Time, clocks, and the ordering of events in a distributed system," *Communications of ACM*, vol. 21, pp. 558–565, July 1978.

[2] F. B. Schneider, "Implementing fault tolerant services using the state machine approach: A tutorial," *ACM Computing Surveys*, vol. 22, pp. 299–319, December 1990.

[3] I. Keidar and D. Dolev, "Efficient message ordering in dynamic networks," in 15^{th} *ACM Symposium on Principles of Distributed Computing (PODC)*, pp. 68–76, May 1996.

[4] Y. Amir, *Replication using group communication over a partitioned network*. PhD thesis, Institute of Computer Science, The Hebrew University of Jerusalem, Jerusalem, Israel, 1995.

[5] A. Fekete, N. Lynch, and A. Shvartsman, "Specifying and using a partitionable group communication service," *ACM Transactions on Computer Systems*, vol. 19, pp. 171–216, May 2001.

[6] R. Friedman and A. Vaysburg, "Fast replicated state machines over partitionable networks," in 16^{th} *IEEE International Symposium on Reliable Distributed Systems (SRDS)*, October 1997.

[7] A. Montresor, R. Davoli, and O. Babaoglu, "Middleware for dependable network services in partitionable distributed systems," in *PODC Middleware Symposium*, July 2000.

[8] R. Guerraoui and A. Schiper, "Software-based replication for fault tolerance," *IEEE Computer*, vol. 30, pp. 68–74, April 1997.

[9] I. Keidar, "A highly available paradigm for consistent object replication," Master's thesis, Institute of Computer Science, The Hebrew University of Jerusalem, Jerusalem, Israel, 1994.

[10] R. Guerraoui and A. Schiper, "Transaction model vs virtual synchrony model: bridging the gap," pp. 121–132, September 1995.

[11] A. Schiper and M. Raynal, "From group communication to transactions in distributed systems," *Communications of the ACM*, vol. 39, pp. 84–87, April 1996.

[12] B. Kemme and G. Alonso, "A suite of database replication protocols based on group communication primitives," in *Proceedings of the* 18^{th} *International Conference on Distributed Computing Systems (ICDCS)*, (Amsterdam, The Netherlands), May 1998.

[13] B. Kemme and G. Alonso, "Don't be lazy, be consistent: Postgres-R, a new way to implement database replication," in *Proceedings of the* 26^{th} *International Conference on Very Large Databases (VLDB)*, (Cairo, Egypt), September 2000.

[14] M. F. Kaashoek and A. S. Tanenbaum, "An evaluation of the Amoeba group communication system.," in 16^{th} *International Conference on Distributed Computing Systems (ICDCS)*, pp. 436–447, May 1996.

[15] G. Goft and E. Y. Lotem, "The AS/400 cluster engine: A case study," in *IGCC 1999, in conjunction with ICPP*, 1999.

[16] D. Cheriton and W. Zwaenepoel, "Distributed process groups in the V kernel," *ACM Transactions on Computer Systems*, vol. 3, no. 2, pp. 77–107, 1985.

[17] I. Rhee, S. Cheung, P. Hutto, and V. Sunderam, "Group communication support for distributed multimedia and CSCW systems," in 17^{th} *International Conference on Distributed Computing Systems (ICDCS)*, 1997.

[18] K. Birman, R. Friedman, M. Hayden, and I. Rhee, "Middleware support for distributed multimedia and collaborative computing," in *Multimedia Computing and Networking (MMCN98)*, 1998.

[19] T. Anker, G. V. Chockler, D. Dolev, and I. Keidar, "The Caelum toolkit for CSCW: The sky is the limit," in 3^{rd} *International Workshop on Next Generation Information Technologies and Systems(NGITS 97)*, pp. 69–76, June 1997.

[20] E. Al-Shaer, A. Youssef, H. Abdel-Wahab, K. Maly, and C. M. Overstreet, "Reliability, scalability and robustness issues in IRI," in *IEEE* 6^{th} *Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises (WETICE'97)*, June 1997.

[21] S. Chodrow, M. Hircsh, I. Rhee, and S. Y. Cheung, "Design and implementation of a multicast audio conferencing tool for a collaborative computing framework," in *JCIS*, March 1997.

[22] A. Krantz, S. Chodrow, and M. Hircsh, "Design and implementation of a distributed multiplexor," in 18^{th} *International Conference on Distributed Computing Systems (ICDCS)*, May 1998.

[23] J. Sussman and K. Marzullo, "The bancomat problem: An example of resource allocation in a partitionable asynchronous system," in 12^{th} *International Symposium on Distributed Computing (DISC)*, September 1998.

[24] O. Babaoglu, A. Davoli, A. Montresor, and R. Segala, "System support for partition-aware network applications," in 18^{th} *International Conference on Distributed Computing Systems (ICDCS)*, pp. 184–191, May 1998.

[25] R. Khazan, A. Fekete, and N. Lynch, "Multicast group communication as a base for a load-balancing replicated data service," in 12^{th} *International Symposium on Distributed Computing (DISC)*, (Andros, Greece), pp. 258–272, September 1998.

[26] S. Dolev, R. Segala, and A. Shvartsman, "Dynamic load balancing with group communication," in 6^{th} *International Colloquium on Structural Information and Communication Complexity (SIROCCO'99)*, pp. 111–125, 1999.

[27] Y. Amir, D. Breitgand, G. Chockler, and D. Dolev, "Group communication as an infrastructure for distributed system management," in 3^{rd} *International Workshop on Services in Distributed and Networked Environment (SDNE)*, pp. 84–91, June 1996.

[28] Y. Amir, R. Caudy, A. Munjal, T. Schlossnagle, and C. Tutu, "N-way fail-over infrastructure for survivable servers and routers," in *IEEE International Conference on Dependable Systems and Networks (DSN03), San Francisco, June 2003*, (San Franciso, CA), June 1993.

[29] E. Al-Shaer, H. Abdel-Wahab, and K. Maly, "Hifi: A new monitoring architecture for distributed system management," in 19^{th} *International Conference on Distributed Computing Systems (ICDCS)*, pp. 171–178, June 1999.

[30] Y. Amir, Y. Gu, T. Schlossnagle, and J. Stanton, "Practical cluster applications of group communication," in *International Conference on Dependable Systems and Networks (DSN00)*, (New York, NY), June 2000.

[31] S. Mishra and G. Pang, "Design and implementation of an availability management service," in 19^{th} *International Conference on Distributed Computing Systems (ICDCS) Workshop on Middleware (June 1999)*, pp. 128–133, June 1999.

[32] A. Fekete and I. Keidar, "A framework for highly available services based on group communication," in 19^{th} *International Conference on Distributed Computing Systems (ICDCS) the International Workshop on Applied Reliable Group Communication (WARGC)*, April 2001.

[33] T. Anker, D. Dolev, and I. Keidar, "Fault tolerant video-on-demand services," in 19^{th} *International Conference on Distributed Computing Systems (ICDCS)*, pp. 244–252, June 1999.

[34] S. Johnson, F. Jahanian, S. Ghosh, B. Vanvoorst, and N. Weininger, "Experiences with group communication middleware," in *International Conference on Dependable Systems and Networks (DSN)*, 2000. Practical Experience Report.

[35] S. Landis, S., and Maffeis, "Building reliable distributed systems with CORBA," *Theory and Practice of Object Systems*, vol. 3, no. 1, 1997.

[36] L. E. Moser, P. M. Melliar-Smith, , and P. Narasimhan, "Consistent object replication in the Eternal system," *Theory and Practice of Object Systems*, vol. 4, no. 2, pp. 81–92, 1998.

[37] P. Felber, R. Guerraoui, and A. Schiper, "The implementation of a CORBA object group service," *Theory and Practice of Object Systems*, vol. 4, no. 2, pp. 93–105, 1998.

[38] T. Chandra, V. Hadzilacos, S. Toueg, and B. Charron-Bost, "On the impossibility of group membership.," in 15^{th} *ACM Symposium on Principles of Distributed Computing (PODC)*, pp. 322–330, May 1996.

[39] K. P. Birman and T. Joseph, "Exploiting virtual synchrony in distributed systems," in 11^{th} *Annual Symposium on Operating Systems Principles*, pp. 123–138, November 1987.

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

[41] A. Menezes, P. van Oorschot, and S. Vanstone, *Handbook of Applied Cryptography*. CRC Press, 1996.

[42] *Data Encryption Standard (DES),*. No. 46-3 in FIPS, National Institute for Standards and Technology (NIST), 1999. http://csrc.nist.gov/publications/fips/fips46-3/fips46-3.pdf.

[43] *Escrowed Encryption Standard (EES)*. No. 185 in FIPS, National Institute for Standards and Technology (NIST), 1994. http://www.itl.nist.gov/fipspubs/fip185.htm.

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

[45] *Advanced Encryption Standard (AES)*. No. 197 in FIPS, National Institute for Standards and Technology (NIST), 2001. http://csrc.nist.gov/encryption/aes/.

[46] *Recommendations for Block Cipher Modes of Operation.* No. 800-38A in SP, National Institute for Standards and Technology (NIST), 2001. http://csrc.nist.gov/publications/nistpubs/index.html.

[47] *Recommendations for Block Cipher Modes of Operation: The RMAC Authentication Mode.* No. 800-38B in SP, National Institute for Standards and Technology (NIST), 2002. http://csrc.nist.gov/publications/nistpubs/index.html.

[48] I. Mantin, "Analysis of the stream cipher RC4," Master's thesis, Weizmann Institute of Science, 2001.

[49] H. Handschuh and H. Gilbert, "Cryptanalysis of the SEAL encryption algorithm," in *Fast Software Encryption (FSE'97), LNCS 1267* (Springer-Verlag, ed.), pp. 1–12, 1997.

[50] M. Bellare, R. Canetti, and H. Krawczyk, "Keyed hash functions for message authentication," in *Advances in Cryptology – CRYPTO 96* (S.-V. Lecture Notes in Computer Science, ed.), pp. 1–15, 1996.

[51] *Secure Hash Standard (SHA1).* No. 180-1 in FIPS, National Institute for Standards and Technology (NIST), 1995. http://www.itl.nist.gov/fipspubs/fip180-1.htm.

[52] *The Keyed-Hash Message Authentication Code (HMAC).* No. 198 in FIPS, National Institute for Standards and Technology (NIST), 2002. http://csrc.nist.gov/publications/fips/index.html.

[53] R. L. Rivest, A. Shamir, and L. M. Adleman, "A method for obtaining digital signatures and public-key cryptosystems," *Communications of the ACM*, vol. 21, pp. 120–126, Feb. 1978.

[54] *Digital Signature Standard (DSS).* No. 186-2 in FIPS, National Institute for Standards and Technology (NIST), 2000. http://csrc.nist.gov/publications/fips/fips186-2/fips186-2.pdf.

[55] Y. Amir and J. Stanton, "The Spread wide area group communication system," Tech. Rep. 98-4, Johns Hopkins University, Center of Networking and Distributed Systems, 1998.

[56] M. Steiner, G. Tsudik, and M. Waidner, "Key agreement in dynamic peer groups," *IEEE Transactions on Parallel and Distributed Systems*, August 2000.

[57] W. Diffie and M. E. Hellman, "New directions in cryptography," *IEEE Trans. Inform. Theory*, vol. IT-22, pp. 644–654, Nov. 1976.

[58] T. Ballardie, P. Francis, and J. Crowcroft, "Core based trees: An architecture for scalable interdomain multicast routing," in *Proceedings of ACM SIGCOMM'93*, pp. 85–95, 1993.

[59] H. Harney and C. Muckenhirn, "Group key management protocol (GKMP) specification." RFC 2093, July 1997.

[60] T. Hardjono, B. Cain, and I. Monga, "Intradomain group key management protocol." Work in Progress, September 2000.

[61] M. Burmester and Y. Desmedt, "A secure and efficient conference key distribution system," *Advances in Cryptology – EUROCRYPT'94*, May 1994.

[62] D. Harkins and N. Doraswamy, "A secure scalable multicast key management protocol (MKMP)." Work in Progress, November 1997.

[63] T. Ballardie, "Scalable multicast key distribution." RFC 1949, 1996.

[64] G. Caronni, M. Waldvogel, D. Sun, N. Weiler, and B. Plattner, "The VersaKey framework: Versatile group key management," *IEEE Journal of Selected Areas in Communication*, vol. 17, September 1999.

[65] H. Harney and E. Harder, "Logical key hierarchy protocol." Work in Progress, March 1999.

[66] D. Balenson, D. McGrew, and A. Sherman, "Key management for large dynamic groups: One-way function trees and amortized initialization." Work in progess, 2000. draft-irtf-smug-groupkeymgmt-oft-00.txt.

[67] H. Harney, A. Schuett, U. Meth, and A. Colegrove, "GSAKMP." Work in Progress, February 2003.

[68] H. Harney, A. Schuett, and A. Colegrove, "GSAKMP light." Work in Progress, July 2002.

[69] M. Baugher, T. Hardjono, H. Harney, and B. Weis, "The group domain of interpretation." Work in Progress, December 2002.

[70] M.Baugher, R. Canetti, L. Dondeti, and F. Lindholm, "Group key management architecture." Work in prograss, 2002. draft-irtf-smug-gkmarch-02.txt.

[71] D. Steer, L. Strawczynski, W. Diffie, and M. Wiener, "A secure audio teleconference system," *Advances in Cryptology – CRYPTO'88*, August 1990.

[72] Y. Kim, A. Perrig, and G. Tsudik, "Simple and fault-tolerant key agreement for dynamic collaborative groups," in 7^{th} *ACM Conference on Computer and Communications Security*, pp. 235–244, ACM Press, November 2000.

[73] W.-G. Tzeng and Z.-J. Tzeng, "Round-efficient conference-key agreement protocols with provable security," in *Advances in Cryptology – ASIACRYPT '2000*, Lecture Notes in Computer Science, (Kyoto, Japan), Springer-Verlag, December 2000.

[74] Y. Kim, A. Perrig, and G. Tsudik, "Communication-efficient group key agreement," in *IFIP SEC 2001*, June 2001.

[75] E. Bresson, O. Chevassut, and D. Pointcheval, "Provably authenticated group Diffie-Hellman key exchange — the dynamic case," in *Asiacrypt 2001* (C. Boyd, ed.), Lecture Notes in Computer Science, (Gold Coast, Australia), Springer Verlag, 2001.

[76] E. Bresson, O. Chevassut, D. Pointcheval, and J.-J. Quisquater, "Provably authenticated group Diffie-Hellman key exchange," in 8^{th} *ACM Conference on Computer and Communications Security* (P. Samarati, ed.), (Philadelphia, PA, USA), ACM Press, Nov. 2001.

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

[78] O. Rodeh, K. Birman, and D. Dolev, "Using AVL trees for fault tolerant group key management," Tech. Rep. 2000-1823, Cornell University, Computer Science; Tech. Rep. 2000-45, Hebrew University, Computer Science, 2000.

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

[80] Y. Amir, D. Dolev, S. Kramer, and D. Malki, "Transis: A communication sub-system for high availability," *Digest of Papers, The* 22^{nd} *International Symposium on Fault-Tolerant Computing Systems*, pp. 76–84, 1992.

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

[82] Y. Amir, L. E. Moser, P. M. Melliar-Smith, D. Agarwal, and P. Ciarfella, "The Totem single-ring ordering and membership protocol," *ACM Transactions on Computer Systems*, vol. 13, pp. 311–342, November 1995.

[83] B. Whetten, T. Montgomery, and S. Kaplan, "A high performance totally ordered multicast protocol," in *Theory and Practice in Distributed Systems, International Workshop*, Lecture Notes in Computer Science, p. 938, September 1994.

[84] T. Anker, G. V. Chockler, D. Dolev, and I. Keidar, "Scalable group membership services for novel applications," in *Proceedings of the workshop on Networks in Distributed Computing*, 1998.

[85] I. Keidar, K. Marzullo, J. Sussman, and D. Dolev, "A client-server oriented algorithm for virtually synchronous group membership in WANs," in 20^{th} *International Conference on Distributed Computing Systems*, pp. 356–365, April 2000.

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

[87] O. Rodeh, K. Birman, and D. Dolev, "The architecture and performance of security protocols in the Ensemble Group Communication System," *ACM Transactions on Information and System Security*, vol. 4, pp. 289–319, August 2001.

[88] M. K. Reiter, "Secure agreement protocols: reliable and atomic group multicast in Rampart," in *Proceedings of the* 2^{nd} *ACM Conference on Computer and Communications Security*, pp. 68–80, ACM, November 1994.

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

[90] P. Zimmermann, *The Official PGP User's Guide*. MIT Press, 1995.

[91] K. P. Birman, M. Hayden, O. Ozkasap, Z. Xiao, M. Budiu, and Y. Minsky, "Bimodal multicast," *ACM Transactions on Computer Systems*, vol. 17, May 1999.

[92] M. A. Hiltunen and R. D. Schlichting, "Adaptive distributed and fault-tolerant systems," *International Journal of Computer Systems Science and Engineering*, vol. 11, pp. 125–133, September 1996.

[93] M. A. Hiltunen, R. D. Schlichting, and C. Ugarte, "Enhancing survivability of security services using redundancy," in *Proceedings of The International Conference on Dependable Systems and Networks*, June 2001.

[94] L. Gong, "Enclaves: Enabling secure collaboration over the Internet," *IEEE Journal on Selected Areas in Communications*, vol. 15, pp. 567–575, April 1997.

[95] P. McDaniel, A. Prakash, and P. Honeyman, "Antigone: A flexible framework for secure group communication," in *Proceedings of the* 8^{th} *USENIX Security Symposium*, pp. 99–114, August 1999.

[96] S. Floyd, V. Jacobson, C. Liu, S. McCanne, and L. Zhang, "A reliable multicast framework for light-weight sessions and application level framing," *IEEE/ACM Transactions on Networking*, vol. 5, pp. 784–803, December 1997.

[97] J. Schultz, "Partitionable virtual synchrony using extended virtual synchrony," Master's thesis, Department of Computer Science, Johns Hopkins University, January 2001.

[98] G. V. Chockler, I. Keidar, and R. Vitenberg, "Group communication specifications: A comprehensive study," *ACM Computing Surveys*, vol. 33, pp. 427–469, December 2001.

[99] Y. Amir, Y. Kim, C. Nita-Rotaru, J. Schultz, J. Stanton, and G. Tsudik, "Exploring robustness in group key agreement," in *Proceedings of the* 21^{st} *IEEE International Conference on Distributed Computing Systems,*, pp. 399–408, IEEE Computer Society Press, April 2001.

[100] K. P. Birman, *Building Secure and Reliable Network Applications.* Manning, 1996.

[101] R. Friedman and R. van Renesse, "Strong and weak virtual synchrony in Horus," Tech. Rep. 95-1537, Cornell University, Computer Science, August 1995.

[102] N. Asokan, V. Schoup, and M. Waidner, "Optimistic fair exchange of digital signatures," *IEEE Journal on Selected Area in Communications*, vol. 18, no. 4, pp. 593,610, 2000.

[103] G. Ateniese, O. Chevassut, D. Hasse, Y. Kim, and G. Tsudik, "Design of a group key agreement API," in *DARPA Information Security Conference and Exposition (DISCEX 2000)*, January 2000.

[104] Y. Amir, Y. Kim, C. Nita-Rotaru, J. Schultz, J. Stanton, and G. Tsudik, "Exploring robustness in group key agreement," Tech. Rep. CNDS 2000-4, Johns Hopkins University, Center of Networking and Distributed Systems, 2000. http://www.cnds.jhu.edu/publications/.

[105] Y. Amir, Y. Kim, C. Nita-Rotaru, and G. Tsudik, "On the performance of group key agreement protocols," in *Proceedings of the* 22^{nd} *IEEE International Conference on Distributed Computing Systems,*, (Viena, Austria), June 2002.

[106] Cliques Project team, "Cliques," 1999. http://sconce.ics.uci.edu/cliques/.

[107] D. Boneh, "Twenty years of attacks on the RSA cryptosystem," *Notices of the American Mathematical Society (AMS)*, vol. 46, no. 2, pp. 203–213, 1999.

[108] Y. Kim, A. Perrig, and G. Tsudik, "Tree-based group key agreement," 2002. In Submission.

[109] *The TLS Protocol Version 1.0.* No. 2246 in RFC, T. Dierks and C. Allen, 1999. http://www.faqs.org/rfcs/rfc2246.html.

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

[111] D. Wallner, E. Harder, and R. Agee, "Key Management for Multicast: Issues and Architectures." RFC 2627, June 1999.

[112] D. McGrew and A. Sherman, "Key establishment in large dynamic groups using one-way function trees." Manuscript, May 1998.