Secure Group Communication Using Robust Contributory Key Agreement [*]

Yair Amir Yongdae Kim Cristina Nita-Rotaru John Schultz

Jonathan Stanton Gene Tsudik

Abstract

Contributory group key agreement protocols generate group keys based on contributions of all group members. Particularly appropriate for relatively small collaborative peer groups, these protocols are resilient to many types of attacks. Unlike most group key distribution protocols, contributory group key agreement protocols offer strong security properties, such as key independence and perfect forward secrecy. This paper presents the first robust contributory key agreement protocol resilient to any sequence of group changes. The protocol, based on the Group Diffie-Hellman contributory key agreement, uses the services of a group communication system supporting Virtual Synchrony semantics. We prove that it provides both Virtual Synchrony and the security properties of Group Diffie-Hellman, in the presence of any sequence of (potentially cascading) node failures, recoveries, network partitions and heals.

We implemented a secure group communication service, Secure Spread, based on our robust key agreement protocol and Spread group communication system. To illustrate its practicality, we compare the costs of establishing a secure group with the proposed protocol and a protocol based on centralized

[*]This work was supported in part by a grant from the National Security Agency under the LUCITE program and by grant F30602-00-2-0526 from the Defense Advanced Research Projects Agency. Parts of this work have appeared as conference publication in ICDCS 2000 [2] and ICDCS 2001 [4].

group key management, adapted to offer equivalent security properties.

Keywords: security, group communication, contributory group key agreement, fault tolerance, cryptographic protocols, robustness.

1 Introduction

Many collaborative settings such as audio- and video-conferencing, white-boards, clustering and replication applications, require services which are not provided by the current network infrastructure. A typical collaborative application operates as a peer group where members communicate via reliable many-to-many multicast, sometimes requiring reliable ordered message delivery. In some settings, group members must be aware of the exact (agreed upon) group membership. Since group communication systems provide these services, many collaborative applications use group communication systems (GCS) as the underlying messaging infrastructure.

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 group setting are largely the same as in point-to-point communication: data secrecy and integrity, and entity authentication. These services cannot be attained without secure, efficient and robust group key management. Many critical applications (e.g., military and financial) applications, require that all intra-group communication to remain confidential. Consequently, not only sufficiently strong encryption must be used to protect intra-group messages, but the underlying group key management must also provide strong security guarantees.

Group keys can be viewed as a sequence of values sorted by time of use, with each key corresponding to a different "snapshot" of a group. A group key is changed whenever the group changes or a periodic re-key is needed. The strongest known security guarantees are *key independence* and *perfect forward secrecy* (PFS). Key independence states that a passive adversary – who, in the worst case, might know all group keys except one – cannot use its knowledge to discover the one key that is missing. PFS demands that the compromise of group members' long-term keys should not lead to the compromise of any previously used group keys (see [41] for formal definitions).

Contributory group key agreement protocols that compute a group key as a (usually, one-way) function of individual contributions from all members, can provide both key independence and PFS properties. At the same time, contributory group key agreement presents a tough practical challenge: its multi-round nature must be reconciled with the possibility of crashes, partitions and other events affecting group membership, that can occur *during* the execution of the group key agreement. Therefore, this paper focuses on **robust** contributory group key agreement.

1.1 Group Key Management

Traditional centralized key management relies on a single fixed key server to generate and distribute keys to the group. This approach is not well-suited 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 or faults. Although a key 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), can support continued operation within an arbitrary partition as long as it includes the source.

The requirement to provide 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 PFS. These properties can only be provided if the key server maintains pairwise secure channels with each group member in order to distribute group keys. 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 disadvantage (common to all centralized fixed-server methods) in that it relies on a single entity to generate good (i.e., cryptographically strong) random keys.

Our approach is to use a fully distributed, contributory group key management algorithm 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 PFS [41].

1.2 Goal and Contribution

Secure, robust and efficient key management is critical 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. The goal of this work is to provide a robust and secure group communication that offers Virtual Synchrony (VS) [14] semantics. Our contribution is three-fold:

1. We present the first robust contributory key agreement protocols that are resilient to any *finite* (even cascading) sequence of events. Our protocols (basic and optimized) are based on Group Diffie-Hellman (GDH) [49] key agreement.

2. We design a robust and secure group communication service by combining our robust key agreement with a reliable group communication service. We prove that the resulting system preserves the Virtual Synchrony properties as well as the security properties of GDH.

3. We provide an insight into the cost of adding security services to GCS, focusing on group key management costs. We describe the implementation of a secure group communication service – Secure Spread – based on our optimized robust key agreement protocol and the Spread [7] group communication system. We present experimental results measuring the delay incurred by a group installing a secure membership following group membership changes. The cost of establishing a secure group when our protocol is used, is compared with the cost of establishing a secure group when a centralized key management protocol, modified such that it provides the **same strong security properties** as our group key agreement, is used.

The rest of the paper is organized as follows. We present our failure and security models in Section 2. Section 3 presents both the group communication service and the key agreement protocol used in designing the robust secure group communication service. We then describe our protocols in Sections

4 and 5 and provide implementation details and performance results in Sections 6 and 7, respectively. Related work is overviewed in Section 8 and the paper concludes with a brief summary in Section 9.

2 Failure Model and Security Assumptions

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

Any process can crashes and recover. A crash of any component of a process, (i.e. key agreement layer or the group communication system), is considered a process crash. We assume that the crash of one of any component is detected by all the other components and is treated as a process crash.

Due to congestion or outright failures the network can be split into disconnected fragments. At the group communication layer, this is referred to as a *partition*. When a partition is repaired, disconnected components merge into a larger connected component, this is referred at the group communication layer as a *merge*. While processes are in separate disconnected components, they cannot exchange messages. Since we are interested in a practical and reasonably efficient solution, we do not consider Byzantine failures in this work.

We do not assume authenticity of membership events. Authentication of new members is obtained as part of group key management. When members leave, no explicit authentication of their departure is obtained. Furthermore, we do not assume any access control mechanisms to enforce membership policies, if any. We recognize that such mechanisms are necessary in real group applications; their development is the subject of recent and on-going work [6, 38, 35].

Our adversary model takes into account only outside adversaries, both passive and active. 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 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.

Passive outsider attacks involve eavesdropping with the aim of discovering the group key(s). Active outsider attacks involve injecting, deleting, delaying and modifying protocol messages. Some of these attacks aim to cause denial of service and we do not address them. Attacks that aim to impersonate a group member are prevented by the use of public key signatures. 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 attacks are prevented by the combined use of timestamps, unique protocol message identifiers and sequence numbers which identify the particular protocol run. This modification of GDH was formally proven secure against active adversaries in [17, 18].

3 Problem Definition

Our goal is to design a secure group communication service by combining a robust key agreement algorithm with a reliable group communication system (GCS). We define the semantics provided by the GCS and overview the GDH key agreement protocol suite, both of which are used later in the paper.

3.1 Group Communication Service

A GCS provides two important services: group membership and dissemination, reliability and ordering of messages. The membership service notifies the application of the current list of group members every time the group changes. The output of this notification is called a *view*.

Several different group communication models [42, 22] have been defined in the literature, each providing a different set of semantics to the application. Many communication models claim to offer Virtual Synchrony or some variant thereof. Such claims are often based on a loose definition of *Virtual Synchrony* stating that: processes moving together from one view to another, deliver the same set of messages in the former view. However, not all the models offer the same set of properties and to the best of our knowledge, a canonical "Virtual Synchrony (VS) Model" has not been defined in the literature. A good survey of many flavors of virtual synchrony semantics can be found in [20].

The ordering and reliability guarantees are provided within a view. In order to specify when the ordering and delivery properties are met, GCS deliver to the application an additional notification referred as a *transitional signal*. Additional information provided with the view by a GCS 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.

One property of the VS model that also has relevance for security, is the *Sending View Delivery* [20] property, which requires messages to be delivered in the same view they were sent in. This enables the use of a shared view-specific key to encrypt data, since the receiver is guaranteed to have the same view as the sender and, therefore, the same key. To satisfy *Sending View Delivery* without discarding messages from group members, a GCS must block the sending of messages before the new view is installed [24]. This is achieved as follows. When a group membership change occurs, the GCS sends a message, *flush request*, to the application asking for permission to install a new view. The application must respond with a *flush acknowledgment* message which follows all the messages sent by the application in the old view. After sending the acknowledgment, the application is not allowed to send any message until the new view is delivered.

Virtual Synchrony Semantics

The GCS is assumed to support VS semantics as defined below. This set of properties is largely based on the survey in [20] and the definition of related semantics in [42] and [47].

We define that some event occurred in view v at process p if the most recent view installed by process p before the event 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 after installing a view v' then $v's$ identifier id_v is greater than v''s identifier $id_{v'}$.
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 and v' deliver the same set of messages in v.

9. *FIFO Delivery*: If message m is sent before message m' by the same process in the same view, then any process that delivers m' delivers m before m'.

10. *Causal Delivery*: If message m causally precedes message m' and both are sent in the same view, then any process that delivers m' delivers m before m'.

11. *Agreed Delivery*:

   1) Agreed delivery maintains all causal delivery guarantees.

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

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

[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.

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 GDH Contributory Key Agreement Protocol

GDH IKA.2 [49] is an extension of the 2-party Diffie-Hellman key exchange protocol [21] to groups. The shared key is never transmitted over the network even in encrypted form. Instead, a set of partial keys (that are used by individual members to compute the group secret) is sent. One particular member, group controller, is charged with the task of building and distributing the set. This is done by passing a token between the members of the group to collect contributions of the new members. The group controller is not fixed and has no special security privileges.

The protocol works as follows. When a merge event occurs the current controller refreshes its own contribution to the group key (to prevent any incoming members from discovering the old group key), generates a new token and passes it to one of the new members. When the chosen new member receives the token, it adds its own contribution and then passes the token to the next new member[2]. 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. 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. Every member can then obtain the group key by factoring in its contribution.

[2]The set of new members and its ordering is decided by the underlying group communication system. The actual order is irrelevant to GDH.

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, the newest remaining member becomes the group controller.

4 Basic Robust Algorithm

This section discusses the details of a basic robust group key agreement algorithm (GKA). We describe the algorithm and prove its correctness, i.e. we show that it preserves virtual synchrony semantics presented in Section 3.1. Throughout the remainder of the paper, we use the term GCS to mean a group communication system providing virtual synchrony semantics.

4.1 Algorithm Description

The GDH IKA.2 protocol, briefly presented in Section 3.2, is secure and correct. Security is preserved independently of any sequence of membership events, while correctness holds only as long as no additional group view change takes place before the protocol terminates. To elaborate further, consider what happens if a leave or partition 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 incorporate a membership protocol (including a fault-detection mechanism), it is not aware of the membership change and the group controller does not proceed until all factor-out tokens (including those from departed members) are collected. Therefore, the system simply blocks. Similar scenarios are also possible if one of the new members crashes while adding its contribution to a group key. In this case, the token never reaches the new group controller and the GDH protocol, once again, blocks.

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. However, this is not optimal since, ideally, multiple additive events ought to be "chained" effectively reducing broadcasts and factor-out token implosions.

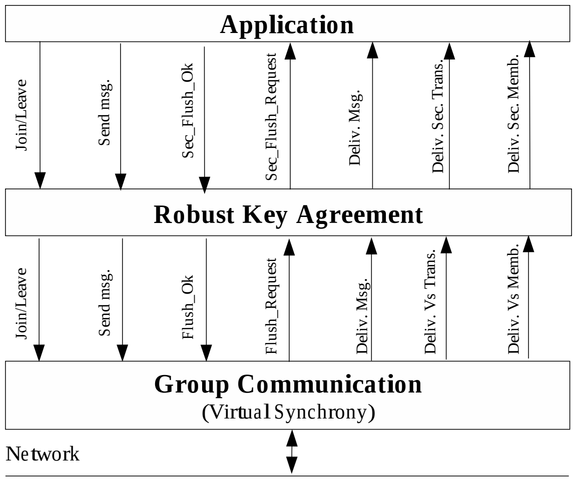


Figure 1. Secure GCS protocol stack

As the above examples illustrate, the GDH protocol does not operate correctly in the face of certain cascaded membership events (specifically, when the interrupting events are subtractive events). This behavior basically violates the high degree of robustness and fault-tolerance of the GCS.

We propose a basic solution as follows: each time a membership change occurs, the group deterministically selects a member (say, the oldest) to initiate the GDH merge protocol. The algorithm uses the membership service to consistently choose that member and the FIFO and Agreed ordering services to ensure that, if one member installs a secure view, all other members eventually install the same view. The approach we propose is twice more expensive in computation and requires $O(n)$ more messages for the common case with no cascading membership events, n being the group size. We discuss it because it is simpler and it allows us to show algorithm correctness with respect to the group communication semantics and stated security goals. In Section 5, we present an optimized algorithm that offers better performance and uses the basic algorithm as a "subroutine" in exceptional cases.

Since the output of the algorithm is a secure GCS, VS semantics as defined in Section 3.1 must be preserved. To achieve this, our algorithm takes extra care to provide delivery of the correct views, transitional signal and transitional sets to the applications, as well as the list of connected group members.

We model the algorithm as a state machine (see Figure 2) where transitions from one state to another take place based on the event that occurred. An event is defined as receiving a particular type of message. In Figure 2, all transitions numbered with the same number, denote the same set of events and actions for that particular state. The following types of messages are used: GDH messages (see [9]) (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).

Figure 1 presents the secure GCS protocol stack. Our group key agreement (GKA) protocol interacts with both the application and GCS and implements the blocking mechanism as follows. When a flush_request_msg message is received from GCS, it is delivered to the application. When the application acknowledgment message is received, it is sent down to the GCS.

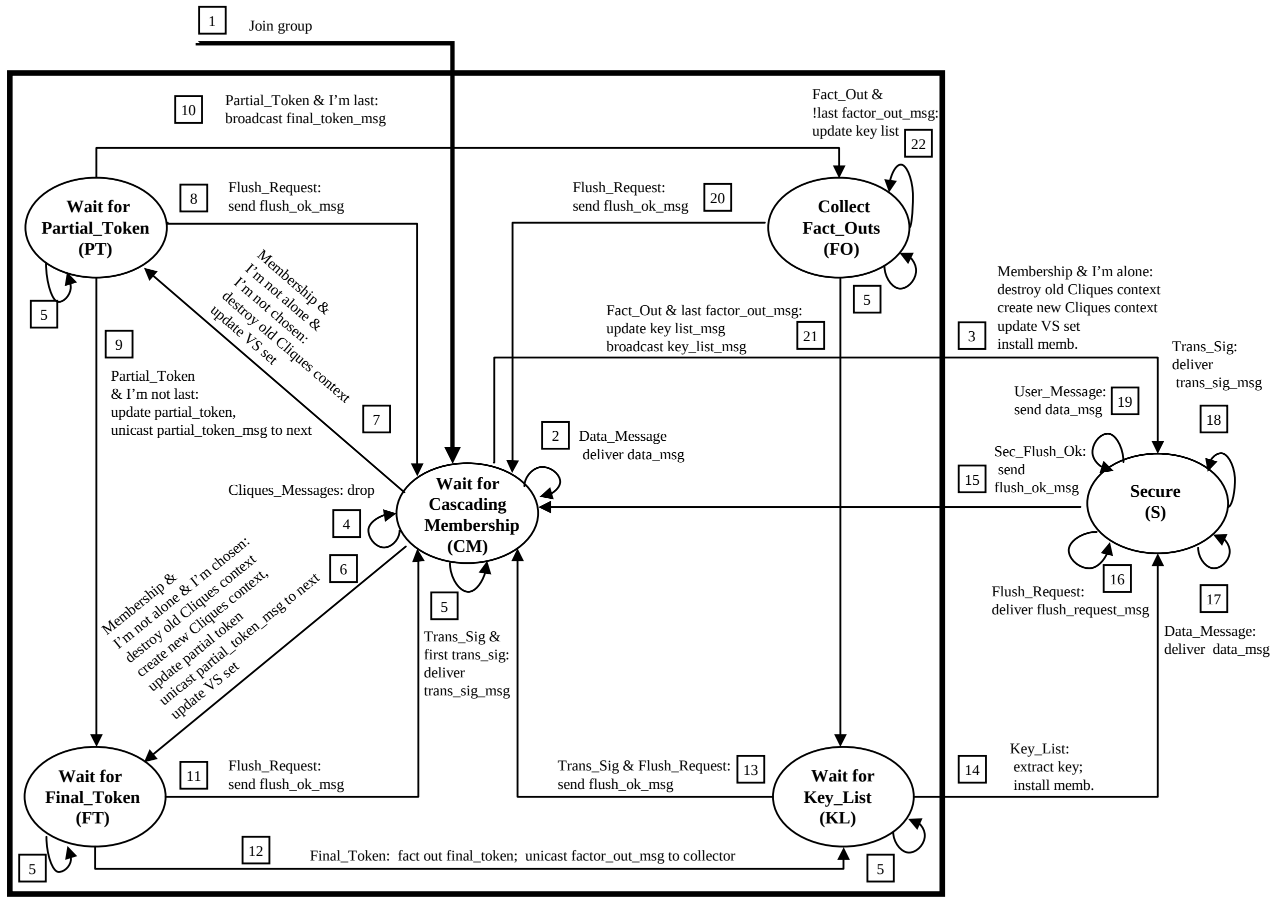
A process starts executing the algorithm by invoking the *join* primitive of the key agreement module which translates into a GCS join call. In any state of the algorithm, a process can voluntarily leave by invoking the *leave* primitive of the key agreement module which translates into a GCS leave call.

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 1. Events received by GKA

The set of events that can trigger transitions from one state to another are presented in Table 1. The events are associated with a specific group and are received by the GKA. Note 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. Also, note that all messages specific to the GDH protocol, are particular cases of data_msg. We define specific messages and associate events with them to simplify the description of the protocol.

Each state of the algorithm (in Figure 2) is described by three types of events: events that trigger


1 Join group
10 Partial_Token & I'm last: broadcast final_token_msg
Fact_Out & !last factor_out_msg: update key list 22
Wait for Partial_Token (PT)
8 Flush_Request: send flush_ok_msg
Flush_Request: send flush_ok_msg 20
Collect Fact_Outs (FO)
5
Membership & I'm not alone & I'm not chosen: destroy old Cliques context update VS set
5
Fact_Out & last factor_out_msg: update key list_msg broadcast key_list_msg 21
3 Membership & I'm alone: destroy old Cliques context create new Cliques context update VS set install memb.
9 Partial_Token & I'm not last: update partial_token, unicast partial_token_msg to next
7
Trans_Sig: deliver trans_sig_msg 18
User_Message: send data_msg 19
2 Data_Message deliver data_msg
Sec_Flush_Ok: send flush_ok_msg 15
Cliques_Messages: drop
4
Wait for Cascading Membership (CM)
Secure (S)
6
Membership & I'm not alone & I'm chosen: destroy old Cliques context create new Cliques context, update partial token unicast partial_token_msg to next update VS set
5
Trans_Sig & first trans_sig: deliver trans_sig_msg
16 Flush_Request: deliver flush_request_msg
17 Data_Message: deliver data_msg
Wait for Final_Token (FT)
11 Flush_Request: send flush_ok_msg
Trans_Sig & Flush_Request: send flush_ok_msg 13
Wait for Key_List (KL)
14 Key_List: extract key; install memb.
5
12 Final_Token: fact out final_token; unicast factor_out_msg to collector
5

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 2. Basic GKA

transitions, events that are considered illegal (in which case an error is returned to the application) and events that should not occur when the algorithm operates correctly. The following states are used:

- SECURE (S): group is functional, all members have the group key and can communicate securely; possible events are: Data_Message, User_Message, Secure_Flush_Ok, Flush_Request, and Transitional_Signal; receiving a Secure_Flush_Ok without receiving a corresponding Flush_Request is illegal;
- WAIT_FOR_PARTIAL_TOKEN (PT): process is waiting for the token accumulating contributions; possible events are: Partial_Token, Flush_Request and Transitional_Signal; User_Message and Secure_Flush_Ok are illegal;
- WAIT_FOR_FINAL_TOKEN (FT): process is waiting for the token containing all contributions;

possible events are: Final_Token, Flush_Request and Transitional_Signal; User_Message and Secure_Flush_Ok are illegal;

- COLLECT_FACT_OUTS (FO): process is waiting for $n-1$ factor out messages; possible events are: Fact_Out, Flush_Request, and Transitional_Signal; User_Message and Secure_Flush_Ok are illegal;
- WAIT_FOR_KEY_LIST (KL): process is waiting for the set of partial keys; possible events are: Key_List, Flush_Request, Transitional_Signal; User_Message and Secure_Flush_Ok are illegal;
- WAIT_FOR_CASCADING_MEMBERSHIP (CM): process is waiting for membership and transitional signal messages; 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 state machine is built around the CM state, which is used to restart the protocol. Four other states are just a map of the GDH merge protocol and are used to pass the token accumulating contributions and used to build the set of partial keys. The S state is the operational state. The pseudo-code corresponding to the state machine from Figure 2 and the correctness proofs are presented in Appendix A.

4.2 Security Considerations

The GDH protocol was proven secure against passive adversaries in [49]. As evident from the state machine in Figure 2, the protocol remains intact, i.e., all protocol messages are sent and delivered in the same order as specified in [49]. More precisely, with no cascaded events, our protocol is exactly the same as the original GDH join protocol [49]. In the case of a cascaded event, the protocol is the same as the IKA.2 [49] group key agreement protocol. Since both of these protocols are proven secure, our robust 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 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 [17, 18].

5 An Optimized Robust Algorithm

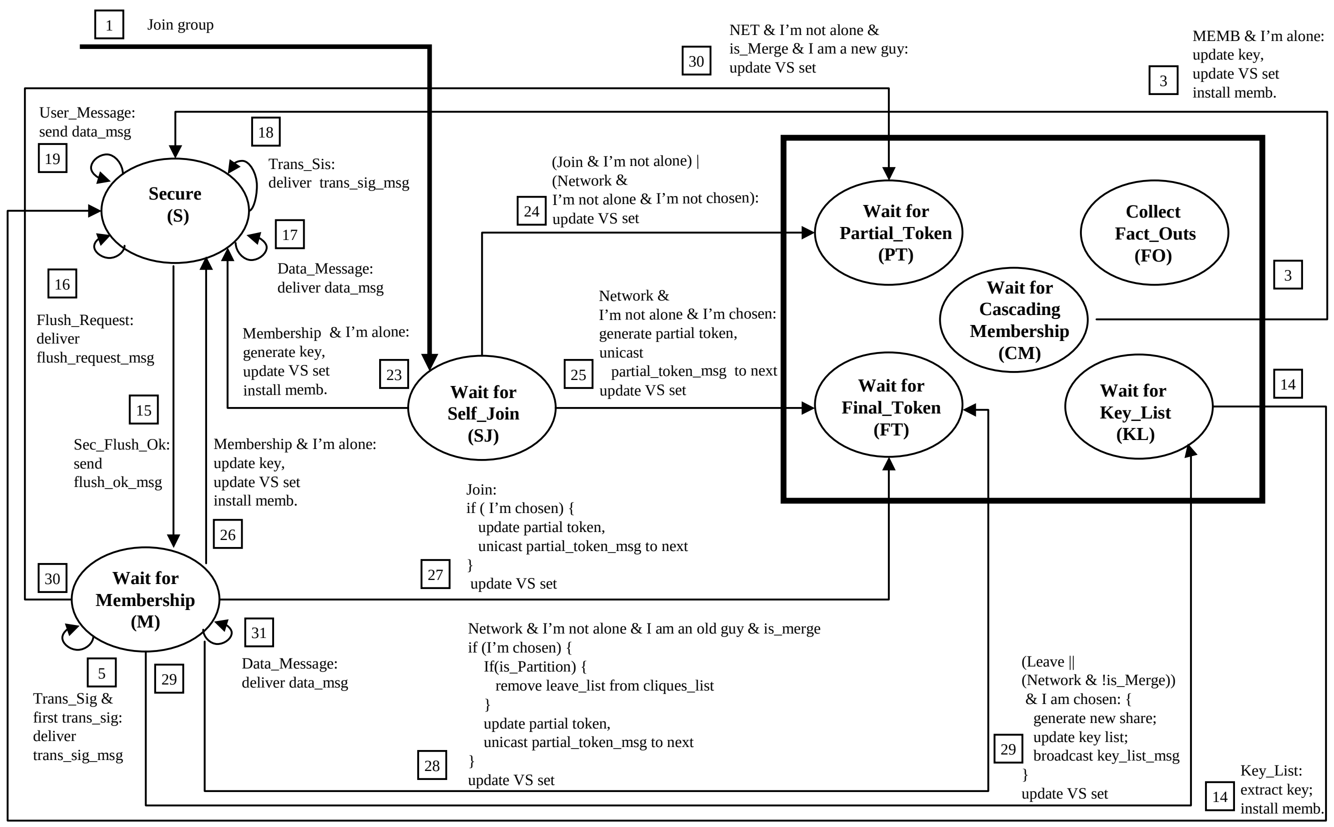
In this section we show how the algorithm presented in the previous section can be optimized, resulting in lower-cost handling of common, non-cascaded events, while preserving the same set of group communication and security guarantees.

5.1 Algorithm Description

The basic algorithm presented in Section 4 is robust even when cascaded group events occur. Every time a membership notification is delivered by the GCS, the algorithm "forgets" all previous key agreement information (i.e., the set of partial keys) and restarts the merge protocol selecting a member from the new group to initialize it. Thus, this algorithm costs more than necessary since it does not attempt to use the existing accumulated information (partial keys) and avoid unnecessary computation.

We propose to improve the basic protocol by using optimized protocols for each type of group change (join, leave, partition, merge or a combination of partition and merge) and by taking advantage of the already existing set of partial keys. We also utilize the basic algorithm to handle more complex cascaded membership events. For example, in case of a leave, the leave protocol is invoked which requires the group controller to remove the leaving member(s) from the set, refresh the set of partial keys and broadcast it. Thus, leave events can be handled immediately, with lower communication and computation costs than those of in the basic algorithm. In Section 5.2, we discuss how a combined event – including both joins and leaves – can be handled by a modified version of the GDH merge protocol.

The optimized algorithm is modeled by a state machine that, in addition to the states in the basic algorithm, uses two more states, as shown in Figure 3. Each state is described by three types of events:


1
Join group
30
NET & I'm not alone & is_Merge & I am a new guy: update VS set
MEMB & I'm alone: update key, update VS set install memb.
3
User_Message: send data_msg
19
18
Trans_Sis: deliver trans_sig_msg
Secure (S)
17
Data_Message: deliver data_msg
16
Flush_Request: deliver flush_request_msg
(Join & I'm not alone) | (Network & I'm not alone & I'm not chosen): update VS set
24
Wait for Partial_Token (PT)
Collect Fact_Outs (FO)
3
Wait for Cascading Membership (CM)
Network & I'm not alone & I'm chosen: generate partial token, unicast partial_token_msg to next update VS set
25
Membership & I'm alone: generate key, update VS set install memb.
23
Wait for Self_Join (SJ)
Wait for Final_Token (FT)
Wait for Key_List (KL)
14
15
Sec_Flush_Ok: send flush_ok_msg
Membership & I'm alone: update key, update VS set install memb.
26
Join: if (I'm chosen) { update partial token, unicast partial_token_msg to next } update VS set
27
30
Wait for Membership (M)
31
Data_Message: deliver data_msg
5
Trans_Sig & first trans_sig: deliver trans_sig_msg
29
Network & I'm not alone & I am an old guy & is_merge if (I'm chosen) { If(is_Partition) { remove leave_list from cliques_list } update partial token, unicast partial_token_msg to next } update VS set
28
(Leave || (Network & !is_Merge)) & I am chosen: { generate new share; update key list; broadcast key_list_msg } update VS set
29
Key_List: extract key; install memb.
14

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 3. Optimized GKA

events that trigger transitions, events that should never occur when the algorithm operates correctly, and events that are considered illegal:

- WAIT_FOR_SELF_JOIN (SJ): initial state wherein 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 a network event and the transitional set will contain only the joining member; possible event is Membership; User_Message and Secure_Flush_Ok are illegal.
- WAIT_FOR_MEMBERSHIP (M): process is waiting for a membership notification; possible events are: Transitional_Signal, Data_Message and Membership; 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 time, 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 of a membership change, the process moves to the M state and tries to handle the event depending on its nature (subtractive, additive or both). In case of cascading membership, everything is abandoned and the basic algorithm is invoked, by moving to the CM state.

The pseudo-code corresponding to the state machine from Figure 3 and the corectness proofs are presented in Appendix B.

5.2 Handling Bundled Events

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

Recall that GDH defines two separate protocols for leave and merge. Each of them can trivially handle bundled events of the same type: the GDH merge protocol can accommodate any combination of bundled merges, while the GDH leave protocol can do the same for any combination of partitions. A more interesting scenario is when a single membership event bundles merges/joins with leaves/partitions. One way to handle such an event is to first invoke GDH leave to process all leaves/partitions and then invoke GDH merge to process joins/merges. However, this is inefficient since the group would perform two separate key agreement protocols where only one is truly needed. Since both GDH protocols are initiated by the group controller we propose the following optimized solution. After processing all leaves/partitions, the group controller can suppress the usual broadcast of new partial keys and, instead, forward the resulting set to the first merging/joining member thereby initiating a merge protocol. This saves an extra round of broadcast and at least one cryptographic operation for each member.

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. Also, 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 the new members in the optimized protocol is identical to that in the plain merge.

6 Implementation

We implemented the optimized algorithm described above using the Spread [7] GCS, and the Cliques key agreement library. In this section we overview the Spread and Cliques toolkits, and present the concrete outcome of this work, the Secure Spread library.

6.1 The Spread Toolkit

Spread [7] is a general-purpose GCS for wide- and local-area networks, 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.

The main services provided by the system are reliable and ordered delivery of messages (FIFO, causal, total/Agreed order, safe) and a membership service in a model that considers benign network and computer faults (crashes, recoveries, partitions and merges). Spread supports two well-known semantics, Virtual Synchrony (VS) [22, 47] and Extended Virtual Synchrony (EVS) [42, 1]. In this work we use only the latter.

The system consists of a server and a client library. The client library provides an API 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 [23]. 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.

The Spread toolkit is publicly available 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.

6.2 The Cliques Toolkit

Cliques is a cryptographic toolkit providing key management services for dynamic peer groups. The toolkit assumes the existence of a communication platform for transporting protocol messages and maintaining group membership. It includes several protocol suites:

- GDH: based on group extensions of the 2-party Diffie-Hellman key exchange [49]; it provides fully contributory group key agreement.
- CKD: centralized key distribution where the key server is dynamically chosen among the group members; the key server uses pairwise Diffie-Hellman key exchange to distribute keys.
- TGDH: combines tree structure with Diffie-Hellman [36] to minimize the computation cost.
- STR: extended version [37] of the protocol presented by Steer et al. [48]; it optimizes communication at the expense of computation.
- BD: a protocol based on the Burmester-Desmedt [19] variation of group Diffie-Hellman.

All Cliques protocol suites offer key independence, perfect forward secrecy and resistance to known key attacks. (See [41] for precise definitions of these properties.) In this paper, we focus only on the GDH protocol suite within the Cliques toolkit.

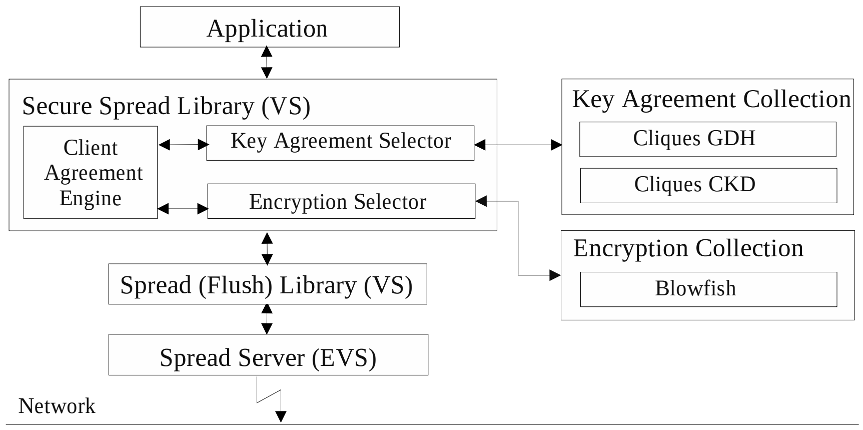


Figure 4. Secure Spread Architecture

6.3 The Secure Spread Library

The Secure Spread library provides client data confidentiality and integrity. It is built on top of the VS Spread client library; it uses Spread as its communication infrastructure and Cliques [9] library for key management.

A major consideration in designing a secure communications toolkit is the security and trust that go along with the algorithms used in the toolkit. As time goes by, trust in these algorithms may change: ciphers are broken, algorithms are proven insecure, better algorithms are designed, etc. Therefore, any system hoping to secure communication and be viable in the long run, needs to be flexible and easily modified. Another desirable feature is the ability of the administrators and/or users to easily change the security policy. One way to achieve these goals is to design an extremely modular system.

Figure 4 shows the architecture of our secure GCS. The Flush library is the component of the Spread Toolkit providing the Virtual Synchrony model as described in Section 3.1 [3].

The core of the library is the Client Agreement Module which is the connection between the library and the GCS. When it receives a notification from GCS about a group membership change, the module starts the key agreement protocol. When the key agreement protocol completes and a new key is

[3]Actually, the Flush library provides all the properties we described in Section 3.1 but one. It does not deliver exactly one transitional signal per view. However, in the Flush library the Flush_Request and Transitional_Signal events are delivered in AGREED order. Using this property, with a minor modification, our GKA can avoid generating unnecessary transitional signals for the application.

available, the module delivers a secure group membership change notification to the application.

The library has two components: Key Agreement Selector and Encryption Selector that allow, respectively, the selection of a specific key agreement module and a specific encryption module.

Secure Spread currently has two different modules for key agreement, both using primitives provided by the Cliques library: the robust optimized GDH protocol (presented in this paper) and a centralized key management protocol (described below), both having the same security properties. The architecture allows each group to run its own key agreement protocol. The library uses Blowfish for encryption.

6.4 Centralized Key Distribution Protocol

In general, centralized key distribution protocols do not provide key independence, since, for efficiency reasons, they rely on previous group or subgroup keys to distribute new keys. When using such a method the compromise of some group keys can lead to the compromise of other group keys. To compare protocols having the same security properties, we designed a Centralized Key Distribution (CKD) scheme that provides the same level of security as GDH, as far as key independence and PFS [41].

Algorithm 1 CKD protocol

```
Let $(x_1, \alpha^{x_1})$ and $(x_{n+1}, \alpha^{x_{n+1}})$ be the secret and public keys of $M_1$, the group controller, and $M_{n+1}$ respectively.
Let $K_{1n+1} = \alpha^{x_1 x_{n+1}}$ mod p. Assume that the group has $n$ members, and that $M_{n+1}$ wants to join the group.
Round 1:
    $M_1$ selects random $r_1 \bmod q$ (this selection is performed only once),
    $M_1 \longrightarrow M_{n+1} : \alpha^{r_1} \bmod p$
Round 2:
    $M_{n+1}$ selects random $r_{n+1} \bmod q$,
    $M_1 \longleftarrow M_{n+1} : \alpha^{r_{n+1}} \bmod p$
Round 3:
    $M_1$ selects a random group secret $K_s$ and computes
    $M_1 \longrightarrow M_i : {K_s}^{\alpha^{r_1 r_i}} \bmod p \;\; \forall i \in [2, n+1]$
```

The group secret is always generated by one member, the current group controller.[4] Following each membership change, the controller generates a new secret and distributes it securely to the group. Of course, an efficient symmetric cipher can be used to securely distribute the group key. However, the resulting security properties would differ from those of our key agreement protocol which relies solely

[4]We use the term *current* to mean that a controller can fail or be partitioned out thus causing the controller role to be reassigned to the oldest surviving member.

on the Decision Diffie-Hellman assumption [16] and the Discrete Logarithm Problem [41]. Therefore, to provide an equivalent level of security, we encrypt the group key via modular exponentiation.

The controller in CKD is always the oldest member. Regardless of the group operation, the CKD protocol consists of two phases (see also Algorithm 1):

1. Each group member and the controller agree on a unique pairwise key using authenticated two-party Diffie-Hellman. This key does not need to change as long as both users remain in the group. If the controller leaves the group, the new controller has to perform this operation with every member. If a regular member leaves, the controller simply discards this pairwise key.

2. The group controller unilaterally generates and distributes the group secret.

7 Performance Evaluation

In this section we compare the GDH key agreement protocol with the CKD protocol presented in Section 6.4, in a LAN environment. We evaluate the time it takes the system to establish secure membership for most common group events: join and leave.

		Communication				Computation		
		Rounds	Messages	Unicast	Multicast	Exp.	Sign.	Verif.
GDH	Join	4	$n+3$	$n+1$	2	$n+3$	4	$n+3$
	Leave	1	1	0	1	$n-1$	1	1
	Merge	$m+3$	$n+2m+1$	$n+2m-1$	2	$n+2m+1$	$m+3$	$n+2m+1$
	Partition	1	1	0	1	$n-p$	1	1
CKD	Join	3	3	2	1	$n+2$	3	3
	Leave	1	1	0	1	$n-2$	1	1
	Merge	3	$m+2$	m	2	$n+2m$	3	$m+2$
	Partition	1	1	0	1	$n-p-1$	1	1
	Controller Leave	3	$3n-6$	$2n-4$	2	$2n-3$	$2n-2$	n

Table 2. Communication and Computation Costs

Cryptography relies on expensive exponentiations, so it seems that measuring CPU time will be a good approach demonstrating the cryptographic overhead. Table 2 presents the number of serial exponentiations for a join or leave event, where n is the group size before the operation, while m and p represent the number of new and partitioned members, respectively. A more relevant measure for a GCS is the latency that a user experiences from the moment the group change was detected, until the

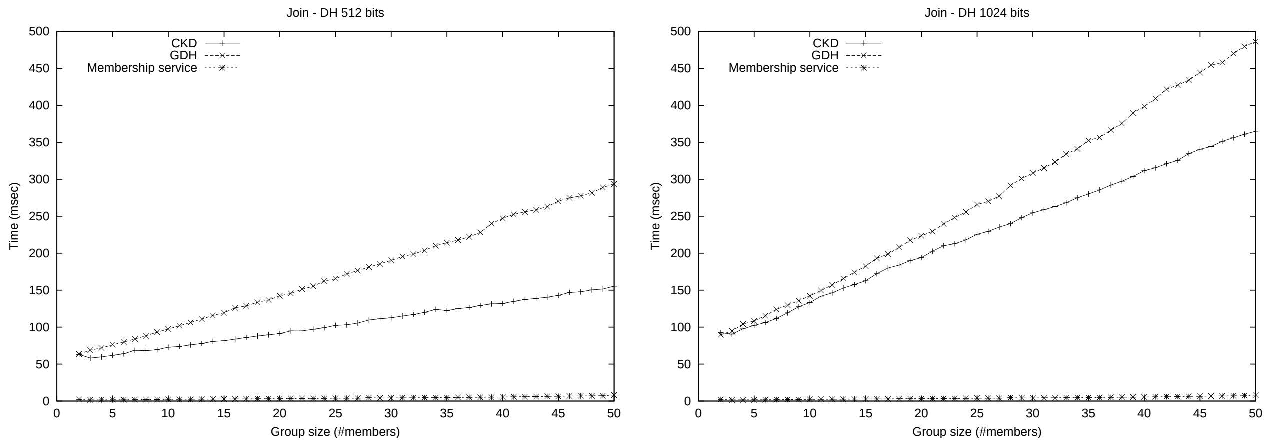


Figure 5. Join - average time

new secure group is established. This time is greater that just the analytical cryptographic cost, since it includes network latency. It can also exhibit increased computation cost if several processes compete for the same CPU.

Our experimental testbed is a cluster of thirteen 667 MHz Pentium III dual-processor PCs running Linux. Each machine runs a Spread server, while group members are uniformly distributed on the machines. Therefore, more than one process can be running on a single machine (which is frequent in many collaborative applications).

Each member measures the time it took to complete the key agreement and establish a secure view. We compute the average cost of the membership service, and secure membership with GDH and CKD, respectively. This time was averaged over 20 distinct runs of the experiment.

Since CKD is particularly expensive if the current controller leaves the group, we take this into account by considering that, with $1/n$ probability, the member leaving the group is the group controller.

Experiments performed on our testbed for the insecure GCS show that the average cost of sending and delivering one agreed multicast message is almost constant, ranging between 0.75 and 0.92 milliseconds for a group size ranging between 2 to 50 members. The cost of the membership service (see Figures 5 and 6) is negligible with respect to key agreement overhead, varying between 2 and 8 milliseconds for a group size between 2 and 50 members. We use 1024-bit RSA signatures for message origin and data authentication, with the public exponent of 3 to reduce the verification overhead. On our hardware

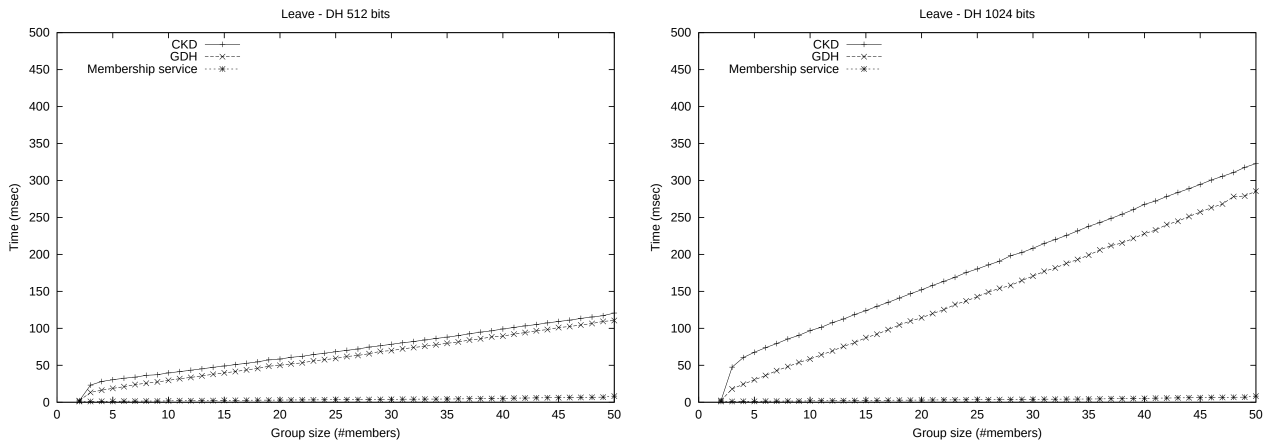


Figure 6. Leave - average time

platform, the RSA sign and verify operations take 9.6 and 0.2 milliseconds, respectively. 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.

Figures 5 and 6 present the respective costs of: 1) our robust GDH key agreement protocol, 2) the centralized protocol (CKD), and 3) the insecure group communication membership service. Note that, the cost of the membership service is insignificant when compared to key agreement overhead.

For join (Figure 5,) the contributory protocol is more expensive than CKD. For example, for a group of 20 members, the time to install a secure membership is about 142 milliseconds for the contributory protocol, while for the centralized protocol it is about 91 milliseconds, when the 512-bit modulus is used. The difference between CKD and GDH comes from exponentiation and signature verifications: extra operations in GDH include n verifications, one RSA signature and one modular exponentiation.

For leave events (see Figure 6), the centralized protocol is more expensive. For the 512-bit modulus, for a group of 20 members, it takes about 49 milliseconds to establish secure membership with the contributory protocol, while the centralized protocol takes about 58 milliseconds. The overhead of the centralized protocol over the contributory protocol comes from one exponentiation, plus the cost of establishing $n - 1$ secure channels if the leaving member is the group controller.

Both protocols scale linearly with group size, in the number of exponentiations. Performance deteriorates when 1024-bit modulus is used for shared key generation, as shown in Figures 5 and 6.

8 Related Work

In this section we consider related work in group key management and reliable group communication.

8.1 Group Key Management

Cryptographic techniques for securing all types of multicast or group based protocols require all parties to share a common key. This requires a group key management protocol to generate new group keys and update existing keys. Group key management protocols generally fall into two classes:

- Protocols designed for large-scale (e.g., IP Multicast) groups, with a one-to-many communication paradigm and relatively weak security requirements [12, 28, 26]. Most of such protocols are centralized key distribution schemes.
- Protocols designed to support medium size tightly-coupled dynamic peer groups, with a many-to-many communication paradigm and strong security requirements [19, 49]. Both distributed group key distribution and group key agreement methods are applicable to such settings.

Many protocols of the first type are being developed in the context of IETF/IRTF: Group Key Management Protocol (GKMP) [28], Multicast Key Management Protocol (MKMP) [27], Scalable Multicast Key Distribution [11], the Intra-domain Group Key Management work of [26], One-way Function Trees [10], Group Secure Association Key Management Protocol (GSAKMP)[30], GSAKMP-light [29], Group Domain of Interpretation (GDOI) [13], while [39] 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 group key agreement schemes [49, 48, 19, 36, 50, 37] extend the well-known Diffie-Hellman key exchange [21] method to groups of n parties. Steer et al. proposed a group key agreement protocol [48] 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 [19] proposed an efficient protocol that takes only three rounds and three modular exponentiations per member to generate a group key. This protocol allows all members to re-compute the group key for any membership change with a constant small CPU cost. However, it requires $2n$ broadcast messages which can be expensive on a wide area network. Tzeng and Tzeng also proposed an elegant authenticated key agreement scheme based on secure multi-party computation [50]. Their protocol is optimized in terms of communication rounds, but also uses $2n$ simultaneous broadcast messages. The resulting group key does not provide PFS which represents a major drawback.

Steiner et al. address dynamic membership issues [49] in group key agreement and propose a family of 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 key agreement. The entire protocol suite has been proven secure with respect to both passive and active attacks. Follow-on work yielded more efficient protocols in either communication or computation [36, 37].

Dynamic group key distribution methods are also amenable to dynamic peer groups. Centralized Key Distribution (CKD) is a simple example of distributed key distribution (see Section 6.4), 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 [46] or taking advantage of efficient data structures such as AVL trees [45]. Although they have some of the disadvantages of key distribution schemes, the communication and computation costs are appreciably lower than those in CKD.

8.2 Reliable Group Communication

Reliable group communication in LAN environments has a long history beginning with ISIS [15], and more recent systems such as Transis [3], Horus [44], Totem [5], and RMP [51]. These systems explored several different models of Group Communication such as Virtual Synchrony [14] and Extended Virtual Synchrony [42]. More recent work in this area focuses on scaling group membership to wide-area networks [8, 33]. Research on securing group communication is fairly new. The only implementations of GCS-s that focus on security (in addition to ours) are the SecureRing [34] project at UCSB, the

Horus/Ensemble work at Cornell [46] and the Rampart system at AT&T [43].

Some GCS-s (Rampart and SecureRing) address Byzantine failures. They suffer from limited performance since they use costly protocols and rely intensively on public key cryptography. Rampart builds the group multicast protocols over a secure group membership protocol, while SecureRing system protects a low-level ring by authenticating each transmission of the token and data message received.

The Ensemble work is state-of-the-art in secure reliable group communication. It allows application-dependent trust models and optimizes certain aspects of the group key generation and distribution protocols. Ensemble achieves data confidentiality by using a shared group key obtained by means of group key distribution protocols. In comparison with our approach, although efficient, the scheme does not provide forward secrecy, key independence and PFS.

Some other approaches focus on building highly configurable dynamic distributed protocols. Cactus [31] 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 [32].

Another toolkit that can be used to build secure group oriented applications is Enclaves [25]. It provides group control and communication (both unicast and multicast) and data confidentiality. The system uses 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 group members. The main drawback of the system is that it does not address failure recovery when the leader of the group fails.

Antigone [40] is a framework that provides mechanisms which allow flexible application security policies. 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 problems, distributing the same key when the group changes breaks PFS, while the session rekeying mechanism does not recover from the leader's failure.

9 Conclusions

In this paper, we showed that although difficult, it is possible to harden security protocols to make them robust to asynchronous network events. In particular, we demonstrated how robust contributory key agreement protocols can be designed, by taking advantage of group communication services. We presented two such robust protocols based on the GDH key protocol suite and the Virtual Synchrony group communication semantics. We also showed how such protocols can be used to design secure group communication services, and argued that by integrating them with a GCS supporting Virtual Synchrony, group communication membership and ordering guarantees are preserved. We exemplified by presenting Secure Spread, a client library that uses Spread as its GCS and relies on a group key management protocol that is 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.

References

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

[2] AMIR, Y., ATENIESE, G., HASSE, D., KIM, Y., NITA-ROTARU, C., SCHLOSSNAGLE, T., SCHULTZ, J., STANTON, J., AND TSUDIK, G. Secure group communication in asynchronous networks with failures: integration and experiments. In *Proceedings of the 20th IEEE International Conference on Distributed Computing Systems* (April 2000), pp. 330–343.

[3] AMIR, Y., DOLEV, D., KRAMER, S., AND MALKI, D. Transis: A communication sub-system for high availability. *Digest of Papers, The 22nd International Symposium on Fault-Tolerant Computing Systems* (1992), 76–84.

[4] AMIR, Y., KIM, Y., NITA-ROTARU, C., SCHULTZ, J., STANTON, J., AND TSUDIK, G. Exploring robustness in group key agreement. In *Proceedings of the 21th IEEE International Conference on Distributed Computing Systems*, (April 2001), pp. 399–408.

[5] AMIR, Y., MOSER, L. E., MELLIAR-SMITH, P. M., AGARWAL, D., AND CIARFELLA, P. The Totem single-ring ordering and membership protocol. *ACM Transactions on Computer Systems 13*, 4 (November 1995), 311–342.

[6] AMIR, Y., NITA-ROTARU, C., AND STANTON, J. Framework for authentication and access control of client-server group communication systems. In *3rd International Workshop on Networked Group Communication* (November 2001).

[7] AMIR, Y., AND STANTON, J. The Spread wide area group communication system. Tech. Rep. 98-4, Johns Hopkins University, Center of Networking and Distributed Systems, 1998.

[8] ANKER, T., CHOCKLER, G. V., DOLEV, D., AND KEIDAR, I. Scalable group membership services for novel applications. In *Proceedings of the Workshop on Networks in Distributed Computing* (1998).

[9] ATENIESE, G., CHEVASSUT, O., HASSE, D., KIM, Y., AND TSUDIK, G. Design of a group key agreement API. In *DARPA Information Security Conference and Exposition (DISCEX 2000)* (January 2000).

[10] BALENSON, D., MCGREW, D., AND SHERMAN, A. Key management for large dynamic groups: One-way function trees and amortized initialization. Work in Progress, 2000. draft-irtf-smug-groupkeymgmt-oft-00.txt.

[11] BALLARDIE, T. Scalable multicast key distribution. RFC 1949, 1996.

[12] BALLARDIE, T., FRANCIS, P., AND CROWCROFT, J. Core based trees: An architecture for scalable interdomain multicast routing. In *Proceedings of ACM SIGCOMM'93* (1993), pp. 85–95.

[13] BAUGHER, M., HARDJONO, T., HARNEY, H., AND WEIS, B. The group domain of interpretation. Work in Progress, December 2002.

[14] BIRMAN, K. P., AND JOSEPH, T. Exploiting virtual synchrony in distributed systems. In *11th Annual Symposium on Operating Systems Principles* (November 1987), pp. 123–138.

[15] BIRMAN, K. P., AND RENESSE, R. V. *Reliable Distributed Computing with the Isis Toolkit*. IEEE Computer Society Press, March 1994.

[16] BONEH, D. The decision Diffie-Hellman problem. *Lecture Notes in Computer Science 1423* (1998), 48–63.

[17] BRESSON, E., CHEVASSUT, O., AND POINTCHEVAL, D. Provably authenticated group Diffie-Hellman key exchange — the dynamic case. In *Asiacrypt 2001* (2001), LNCS.

[18] BRESSON, E., CHEVASSUT, O., POINTCHEVAL, D., AND QUISQUATER, J.-J. Provably authenticated group diffie-hellman key exchange. In *8th ACM Conference on Computer and Communications Security* (Nov. 2001), ACM Press.

[19] BURMESTER, M., AND DESMEDT, Y. A secure and efficient conference key distribution system. *Advances in Cryptology – EUROCRYPT'94* (May 1994).

[20] CHOCKLER, G. V., KEIDAR, I., AND VITENBERG, R. Group communication specifications: A comprehensive study. *ACM Computing Surveys*, 4 (December 2001), 427–469.

[21] DIFFIE, W., AND HELLMAN, M. E. New directions in cryptography. *IEEE Trans. Inform. Theory IT-22* (Nov. 1976), 644–654.

[22] FEKETE, A., LYNCH, N., AND SHVARTSMAN, A. Specifying and using a partitionable group communication service. *ACM Transactions on Computer Systems 19*, 2 (May 2001), 171–216.

[23] Floyd, S., Jacobson, V., Liu, C., McCanne, S., and Zhang, L. A reliable multicast framework for light-weight sessions and application level framing. *IEEE/ACM Transactions on Networking 5*, 6 (December 1997), 784–803.

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

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

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

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

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

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

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

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

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

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

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

[35] Kim, Y., Mazzocchi, D., and Tsudik, G. Admission control in collaborative groups. In *2nd IEEE International Symposium on Network Computing and Applications* (April 2003).

[36] Kim, Y., Perrig, A., and Tsudik, G. Simple and fault-tolerant key agreement for dynamic collaborative groups. In *7th ACM Conference on Computer and Communications Security* (November 2000), pp. 235–244.

[37] KIM, Y., PERRIG, A., AND TSUDIK, G. Communication-efficient group key agreement. In *IFIP SEC* (June 2001).

[38] KIM, Y., AND TSUDIK, G. Membership control in peer groups. In *Workshop on New Directions on Scalable Cyber-Security* (March 2003).

[39] M.BAUGHER, CANETTI, R., DONDETI, L., AND LINDHOLM, F. Group key management architecture. Work in Progress, 2002. draft-irtf-smug-gkmarch-02.txt.

[40] MCDANIEL, P., PRAKASH, A., AND HONEYMAN, P. Antigone: A flexible framework for secure group communication. In *Proceedings of the 8th USENIX Security Symposium* (August 1999), pp. 99–114.

[41] MENEZES, A., VAN OORSCHOT, P., AND VANSTONE, S. *Handbook of Applied Cryptography*. CRC Press, 1996.

[42] MOSER, L. E., AMIR, Y., MELLIAR-SMITH, P. M., AND AGARWAL, D. A. Extended virtual synchrony. In *Proceedings of the IEEE 14th International Conference on Distributed Computing Systems* (June 1994), pp. 56–65.

[43] REITER, M. K. Secure agreement protocols: reliable and atomic group multicast in Rampart. In *Proceedings of the 2nd ACM Conference on Computer and Communications Security* (November 1994), ACM, pp. 68–80.

[44] RENESSE, R. V., K.BIRMAN, AND MAFFEIS, S. Horus: A flexible group communication system. *Communications of the ACM 39* (April 1996), 76–83.

[45] RODEH, O., BIRMAN, K., AND DOLEV, D. 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.

[46] RODEH, O., BIRMAN, K., AND DOLEV, D. The architecture and performance of security protocols in the Ensemble Group Communication System. *ACM Transactions on Information and System Security 4*, 3 (August 2001), 289–319.

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

[48] STEER, D., STRAWCZYNSKI, L., DIFFIE, W., AND WIENER, M. A secure audio teleconference system. *Advances in Cryptology – CRYPTO'88* (August 1990).

[49] STEINER, M., TSUDIK, G., AND WAIDNER, M. Key agreement in dynamic peer groups. *IEEE Transactions on Parallel and Distributed Systems* (August 2000).

[50] TZENG, W.-G., AND TZENG, Z.-J. Round-efficient conference-key agreement protocols with provable security. In *Advances in Cryptology – ASIACRYPT '2000* (December 2000), LNCS.

[51] WHETTEN, B., MONTGOMERY, T., AND KAPLAN, S. A high performance totally ordered multicast protocol. In *Theory and Practice in Distributed Systems, International Workshop* (September 1994), LNCS, p. 938.

A Basic Algorithm: Pseudo-Code and Correctness Proof

The pseudo-code describing our basic algorithm is presented in Algorithms 2, 3, 4, 5, 6, 7, and 8. Each Algorithm describes the actions specific to a state. There are two types of actions: group communication operation (message delivery, message unicast, message broadcast, or send a flush acknowledgment) and GDH key agreement specific operation (computation on the token or access to GDH state information).

We use several simple procedures:

- *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 GDH 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 chooses a member on the list 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.

GCS-s 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[5].

Algorithm 2 Initialization of global variables

```
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
VS_set := EMPTY
First_transitional := TRUE
VS_transitional := FALSE
First_cascaded_membership := TRUE
Wait_for_sec_flush_ok := FALSE
KL_got_flush_req := FALSE
Event := NULL
Clq_ctx := NULL
Group_key := NULL
State := WAIT_FOR_CASCADING_MEMBERSHIP
/* for opt. Alg., replace the above line with:
State := WAIT_FOR_SELF_JOIN */
```

Every process executes the algorithm for a specific group and maintains a list of global variables (see Algorithm 2): *Group_name* is the name of the group for which the algorithm is executed; *Group_key* is the shared secret of the group; *Me* is the process executing the algorithm; *Event* is the current event being handled; *Clq_ctx* keeps 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* is the new membership that will be delivered; *VS_set* is used to compute the transitional set delivered to the application with a new membership. 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: *First_transitional*, *First_cascaded_membership*, *Wait_for_sec_fl_ok*, *VS_transitional* and *KL_got_flush_reg*. The names of all global variables are capitalized whereas all other

[5]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.

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 choose to use a more expensive service for the last broadcast to reduce the complexity of the algorithm and the proofs.

Algorithm 3 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
```

A.1 Correctness Proof

We now prove that the above algorithm preserves the Virtual Synchrony Model described in Section 3.1.

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 se-

Algorithm 4 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)
        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
```

cure view delivery event. The secure membership notification is built and saved in the CM state (see Algorithm 8). 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 A.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 A.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.*

Algorithm 5 Code executed in WAIT_FOR_KEY_LIST (KL) state

```
case Event is
Key_List:
    if (!VS_transitional)
        Clq_ctx := clq_update_ctx(Clq_ctx, key_list_msg)
        Group_Key := clq_extract_key(Clq_ctx)
        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
```

We now prove the following lemmas.

Lemma A.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 joining. 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

Algorithm 6 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
```

in the CM state and does not leave it until it receives a membership event.

Lemma A.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 A.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 Sending View Delivery. 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

Algorithm 7 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
```

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 A.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 Algorithm 8, Marks 1 and 2), which, by Lemma A.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

Algorithm 8 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)
            merge_set := memb_msg.mb_set - Me
            partial_token_msg := clq_update_key(Clq_ctx, merge_set)
            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)
            State := WAIT_FOR_PARTIAL_TOKEN
        endif
    else /* alone */
        clq_destroy_ctx(Clq_ctx)
        Clq_ctx := clq_first_member(Me)
        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
```

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.

A.1.1 Self Inclusion

Theorem A.1 *When process p installs a secure view, the view includes p.*

Proof: This holds due to Lemma A.5.

A.1.2 Local Monotonicity

Theorem A.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 A.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.

A.1.3 Sending View Delivery

Theorem A.3 *A message is delivered by the GKA in the secure view that it was sent in.*

Proof: By Lemma A.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 A.5), so by 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 membership by sending a flush_ok message (see Algorithm 3). 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.

A.1.4 Delivery Integrity

Theorem A.4 *If 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 A.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.

A.1.5 No Duplication

Theorem A.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.

A.1.6 Self Delivery

Theorem A.6 *If 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.

A.1.7 Transitional Set

Theorem A.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 A.6 *If 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 A.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 A.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 GCS Transitional Set property number two.

Theorem A.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 A.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 that p is not in q's transitional set by 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. 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.

A.1.8 Virtual Synchrony

Theorem A.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 A.4) and VS membership notifications are received by the GKA only in the CM state (Lemma A.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 $v2$ [6]. Therefore, by the GCS Virtual Synchrony, 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.

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

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 $v1\_sec$.

A.1.9 FIFO, Causal, Agreed and Safe Delivery

Lemma A.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 A.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 A.7.

Theorem A.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 A.7.

Theorem A.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 A.7 and because the secure transitional set is the intersection of all the VS transitional sets.

Theorem A.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 A.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.

A.1.10 Transitional Signal

Theorem A.15 *Each process delivers exactly one transitional signal per view.*

Proof: GCS Transitional Signal property number one 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 Algorithms 3, 5, 4, 6, 7, 8).

B Optimized Algorithm: Pseudo-Code and Correctness Proof

The pseudo-code corresponding to the state machine from Figure 3 is presented in Algorithms 2, 3, 4, 5, 6, 7, 8, 9 and 10.

The description of the protocol we use two additional fields (*merge_set* and *leave_set*) of the membership notification to determine the cause of the group view change. In addition, we use a modified version of the procedure *clq_update_key* that can handle combined network events.

Algorithm 9 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)
            merge_set := memb_msg.merge_set
            partial_token_msg := clq_update_key(Clq_ctx, merge_set)
            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)
            State := WAIT_FOR_PARTIAL_TOKEN
        endif
    else
        Clq_ctx := clq_first_member(Me)
        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
```

B.1 Correctness Proof

The proof that the optimized algorithm described above provides the virtual synchrony semantics presented in Section 3.1 is very similar to the proof we provided for the basic algorithm. There are some differences in the optimized algorithm: 1) secure memberships can be installed in three states, CM, SJ and M; 2) application messages are delivered in the S and M states; 3) membership notifications are received from the GCS in the CM, SJ, and M states; 4) 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,

Algorithm 10 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_update_key(Clq_ctx,leave_set,merge_set)
                    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)
                State := WAIT_FOR_PARTIAL_TOKEN
            endif
        endif
    else /* alone */
        Clq_ctx := clq_first_member(Me)
        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
```

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 B.1 *The only states where VS membership notifications are received are the SJ, CM and M states.*

Lemma B.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 3.1 can be proven by using the above lemmas and the properties provided by the underlying GCS. 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.

B.1.1 Virtual Synchrony

Theorem B.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 (Lemma B.2) and VS membership notifications are received only in the SJ, CM and M states (Lemma B.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 $v1\_sec$.