Exploring Robustness in Group Key Agreement [*]

Yair Amir [†] Yongdae Kim [‡] Cristina Nita-Rotaru [†] John Schultz [†] Jonathan Stanton [†]
Gene Tsudik [§]

Abstract

Secure group communication is crucial for building distributed applications that work in dynamic environments and communicate over unsecured networks (e.g. the Internet). Key agreement is a critical part of providing security services for group communication systems. Most of the current contributory key agreement protocols are not designed to tolerate failures and membership changes during execution. In particular, nested or cascaded group membership events (such as partitions) are not accommodated.

In this paper we present the first robust contributory key agreement protocols resilient to any sequence of events while preserving the group communication membership and ordering guarantees.

1 Introduction

The explosive growth of the Internet has increased both the number and the popularity of applications that require a reliable group communication infrastructure, such as voice- and video-conferencing, white-boards, distributed simulations, and replicated servers of all types.

Secure group communication is crucial for building distributed applications that work in dynamic network environments and communicate over insecure networks such as the global Internet. Key management is the base for providing common security services (data secrecy, authentication and integrity) for group communication. There are several approaches to group key management.

One approach relies on a single, centralized entity, to generate keys and distribute them to the group. In this case, a so-called key server maintains long-term shared keys with each group member in order to enable secure two-party communication for the actual key distribution. A specific form of this solution uses a fixed trusted third party (TTP) as the key server. This approach has two problems: 1) the TTP must be constantly available and 2) a TTP must exist in every possible subset of a group in order to support continued operation in the event of network partitions. The first problem can be addressed with fault-tolerance and replication techniques. The second, however, is impossible to solve in a scalable and efficient manner. We note, however, that centralized approaches work well in a one-to-many multicast scenario since a TTP (or a set thereof) placed at, or very near, the source of communication can support continued operation within an arbitrary partition as long as it includes the source. (Typically, one-to-many settings only aim to offer continued operation within a single partition that includes the source; whereas, many-to-many environments must offer the same in an arbitrary number of partitions.)

Another key management approach involves dynamically selecting – in some deterministic manner – a group member charged with the task of generating

[*]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.

[†]Department of Computer Science, Johns Hopkins University, Baltimore, MD 21218, USA. Email: {yairamir, crisn, jschultz, jonathan}@cs.jhu.edu

[‡]Computer Networks Division, USC Information Sciences Institute, Marina Del Ray, CA 90292-6695, USA. Email: yongdaek@isi.edu

[§]Information and Computer Science Department, University of California, Irvine Irvine, CA 92697-3425, USA. Email: gts@ics.uci.edu

keys and distributing them to other group members. This approach is robust and more amenable to many-to-many type of group communication since any partition can continue operation by electing a temporary key server. The drawback here is that, as in the TTP case, a key server must establish long-term pairwise secure channels with all current group members in order to distribute group keys. Consequently, each time a new key server comes into play, significant costs must be incurred to set up these channels. Another disadvantage, again as in the TTP case, is the reliance on a single entity to generate good (i.e., cryptographically strong, random) keys.

In contrast to the above, contributory key management asks each group member to contribute an equal share to the common group key (computed as a function of all members' contributions). This approach avoids the problems with the single points of trust and failure. Moreover, some contributory methods do not require the establishment of pairwise secret channels among group members. However, current contributory key agreement[1] protocols are not designed to tolerate failures and group membership changes during execution. In particular, nested (or cascaded) failures, partitions and other group events are not accommodated. This is not surprising since most multi-round cryptographic protocols do not offer built-in robustness with the notable exception of protocols for fair exchange [1].

The main goal of this paper is to demonstrate how provably secure, multi-round group key agreement protocols can be combined with reliable group communication services to obtain provably **fault-tolerant** group key agreement solutions. More precisely, we present two robust contributory key agreement protocols which are resilient to any sequence (even cascaded) of events while preserving group communications membership and ordering guarantees. Both protocols are based on Cliques GDH contributory key agreement that generalizes on the two-party Diffie-Hellman [2] key exchange. Our first protocol utilizes membership information provided by the group communication system in order to appropriately re-start Cliques GDH key agreement in an agreed-upon manner every time the group changes. The second protocol optimizes the performance of common cases at the cost of a more sophisticated protocol state machine.

The rest of the paper is organized as follows. The remainder of this section focuses on our motivation in pursuing this work and overviews related work. We then present Secure Spread, a secure group communication system which utilizes our key agreement protocols. The two subsequent sections present two robust key agreement protocols and prove their correctness. Finally, we summarize our work and discuss some future directions.

1.1 Motivation

As mentioned earlier, a prominent challenge encountered in securing group communication is in developing robust, reliable and fault-tolerant group key management mechanisms that perform well in practice. While the motivation for security services (key management, in particular) in a tightly-coupled group communication setting is fairly intuitive, the need for reliable group communication services by the group key management is less obvious. We claim that reliable and sequenced message delivery is important (and even crucial) for cryptographic group protocols. Asynchronous network behavior must be handled by the underlying group communication layer, which prompts the need for a highly reliable group communication service.

This dependence is both natural and mutual. It is natural since secure dynamic peer groups always require certain communication guarantees. (Best-effort datagram service is not usually a viable option, whereas, it may suffice for one-to-many type groups encountered in Internet multicast settings.) It is mutual since reliable group communication systems are of limited utility in open networks without strong security services and guarantees. Thus, we have interdependence among reliable group communication and group key management protocols.

Cryptographic protocols designers are primarily concerned with security and typically assume that protocol robustness is handled by the particular application or by the underlying communication layer. This is reasonable in two-party protocols where communication failures are relatively easy to handle and recover

[1]We use the term "agreement," as opposed to "distribution", to emphasize the contributory nature of the key management.

from. The picture changes dramatically in group protocols where the behavior model is richer.

Multi-round group key management protocols cannot be expected to run to completion without being possibly interrupted by various group membership events: joins, leaves, disconnects, partitions, merges or any combination thereof.

Our previous work [3] focused on the performance evaluation in the scenario with no network faults or cascaded events and provided a good insight of the overall cost of high security in a group communication environment. The present work goes into the details of a complete solution that handles every possible combination of group membership events. The contribution of this paper, therefore, is the design, and the proof of correctness of, a robust contributory key agreement algorithm.

1.2 Related Work

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

1.2.1 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 (GKM) protocol to provide methods for generating new group keys and updating existing keys. GKM protocols generally fall into two classes:

- Protocols designed for large-scale (e.g., IP Multicast) applications with a one-to-many communication paradigm and relatively weak security requirements.

- Protocols designed to support tightly-coupled dynamic peer groups with modest scalability requirements, a many-to-many communication paradigm and strong security requirements.

GKM protocols of the first type are being developed in the context of IETF/IRTF. One example is the Group Key Management Protocol (GKMP) [4] which provides key dissemination using a dedicated group controller. Another is the Multicast Key Management Protocol (MKMP) [5] which assumes a number of trusted “key distributors” exist throughout the network. (MKMP provides a way for a group member to probe for the nearest distributor in order to get a copy of a group key.) Some GKM protocols leverage off particular IP Multicast routing protocols. The Scalable Multicast Key Distribution [6] approach uses the Core Based Trees [7] multicast routing protocol state and structure to authorize members and disseminate keys. Although it provides an efficient method of key dissemination, this method is limited to domains that use CBT for multicast routing.

Some key management approaches targeting IP Multicast use hierarchical key distribution. For example, the Iolus system [8], partitions the multicast tree into subgroups; each sub-group has a different group key and nodes on the borders of sub-groups perform re-encryption of multicast data in real time. The work of [9] and the Intra-domain Group Key Management Protocol advance this concept by allowing each subgroup to be a separate domain with independent control over what group keying protocol is used. Another hierarchical approach makes the group key itself hierarchical, usually with a tree-based structure. In [10], a tree-oriented key structure allows each leaf to represent a number of nodes and some membership changes to only require $\log(n)$ key changes.

A number of GKM protocols supporting abstract peer groups have been developed in the last decade [11], [12], [13], [14], [15], [16], [17]. All, except [17], extend the well-known Diffie-Hellman key exchange [2] method to group of n parties. These protocols vary in degrees of protection from hostile attacks and in their performance characteristics. (For an in-depth comparison, see [16].) In this paper, we make use of the CLIQUES toolkit which implements – among other methods – a suite of protocols, called generic Group Diffie-Hellman (GDH). GDH offers contributory authenticated group key agreement and handles dynamic membership changes [15, 16]. The entire protocol suite has been proven secure with respect to both passive and active attacks.

1.2.2 Reliable Group Communication

Reliable group communication in LAN environments have a well-developed history beginning with ISIS

[18], and more recent systems such as Transis [19], Horus [20], Totem [21], and RMP [22]. These systems explored several different models of Group Communication such as Virtual Synchrony [23] and Extended Virtual Synchrony [24]. More recent work in this area focuses on scaling group membership to wide-area networks [25], [26].

Research in securing group communication is fairly new. The only actual implementations of group communication systems that focus on security (in addition to ours), are SecureRing [27] project at UCSB, and the Horus/Ensemble work at Cornell [28]. The SecureRing system protects a low-level ring protocol by using cryptographic techniques to authenticate each transmission of the token and each data message received. The Ensemble security work is the state-of-the-art in secure reliable group communication and addresses problems as group keys and re-keying. It also allows application-dependent trust models and optimizes certain aspects of group key generation and distribution protocols. In comparison with our approach, Ensemble uses a different group key structure that is not contributory and provides a different set of security guarantees.

Recent research on Bimodal-Multicast, Gossip-based protocols [29] and the Spinglass system has largely focused on increasing the scalability and stability of reliable group communication services in more hostile environments such as wide-area and lossy networks by providing probabilistic guarantees about delivery, reliability, and membership.

2 A Secure Group Communication Environment

The work discussed in this paper has involved integrating the Spread wide-area group communication system with the group key agreement protocols in the Cliques GDH protocol suite. In this section we overview both Spread and Cliques toolkits.

2.1 Spread Toolkit

Spread [30], [31] is a group communication system for wide and local area networks. It provides all the services of traditional group communication systems, including: unreliable/reliable delivery, FIFO, causal, total ordering, and membership services with strong semantics.

Spread creates an overlay network that can impose an arbitrary network configuration (such as point-to-multi-point, tree, ring, tree-with-subgroups or any combination thereof) to adapt the system to different network environments. The Spread architecture allows multiple protocols to be used on links both between and within sites. The Spread toolkit is very useful for applications that need traditional group communication services (such as causal and total ordering, membership and delivery guarantees) but also need to operate over wide-area networks.

The system consists of a long-running daemon and a library linked with the application. This architecture has many benefits, the most important for wide-area settings being the ability to pay the minimum possible price for different causes of group membership changes. A simple join or leave of a process translates into a single message, while a daemon disconnection or connection requires a full membership change. Luckily, there is a strong inverse relationship between the frequency of these events and their cost in a practical system. The process and daemon memberships correspond to the more common model of light-weight and heavy-weight groups.

Spread scales well with the number of groups used by the application without imposing any overhead on network routers. Group naming and addressing is not a shared resource (as in IP multicast addressing) but rather a large space of strings which is unique to a collaboration session.

The toolkit can support a large number of different collaboration sessions, each of which spans the Internet but has only a moderate number of participants. This is achieved by using unicast messages over the wide-area network, routing them between Spread nodes on the overlay network.

The Spread system provides two different semantics: Extended Virtual Synchrony [24, 32] and View Synchrony [33]. In this paper, and for our implementation we only use the View Synchrony semantics of Spread.

The Spread toolkit is available publicly and is being used by several organizations for both research and practical projects. The toolkit supports cross-platform applications and has been ported to several Unix platforms as well as Windows and Java environments.

2.2 Cliques Toolkit

Cliques [16, 15, 34] is a cryptographic toolkit providing key management services for dynamic peer groups. Cliques includes several protocol suites:

- GDH: based on group extensions of the 2-party Diffie-Hellman key exchange [15, 16]; provides fully contributory authenticated key agreement. GDH is fairly computation-intensive requiring $O(n)$ cryptographic operations upon each key change. It is, however, bandwidth-efficient.
- CKD: centralized key distribution with the key server dynamically chosen from among the group members. A key server uses pairwise Diffie-Hellman key exchange to distribute keys. CKD is comparable to GDH in terms of both computation and bandwidth costs.
- TGDH: tree-based group Diffie-Hellman [34]; TGDH is more efficient than the above in terms of computation as most operations require $O(\log n)$ cryptographic operations. (The security of TGDH is slightly weaker and it lacks several other features not germane in this context.)
- BD: a protocol based on Burmester-Desmedt [13] variation of group Diffie-Hellman. BD is computation-efficient requiring constant number of exponentiations upon any key change. However, communication costs are significant with two rounds of n-to-n broadcasts.

All Cliques protocol suites offer key independence, perfect forward secrecy and resistance to known key attacks. (See [35, 16] for precise definitions of these properties.)

In this paper, we focus only on the GDH protocol suite within the Cliques toolkit. As mentioned earlier, our specific goal is to take a provably secure, multi-round group key agreement protocol (GDH) and, by combining it with the reliable group communication service (Spread), obtain a provably **fault-tolerant** group key agreement solution.

Cliques GDH API [36] is the implementation of the GDH protocol suite. It contains GDH cryptographic primitives while assuming the existence of a reliable communication platform for transporting protocol messages. GDH assigns a special role to the last member to join a group. This role, referred to as the group controller, floats as group membership changes. A group controller is charged with initiating key updates following membership changes.[2] The following operations trigger a key update:

- join: add a single new member to the group (handled as a special case of merge).
- merge: add multiple members to the group.
- leave: one member voluntarily leaves the group (handled as a special case of partition).
- partition: multiple members leave the group due to expulsion or a network event.

3 System Model

In this section we specify the failure and the group communication models used in this paper.

3.1 Failure Model

We consider a *distributed system*, a group of processes executing on one or more computers and coordinating actions by exchanging messages ([37]). The message exchange is achieved via asynchronous multicast and unicast messages. Messages can be lost.

The system is subject to process crashes and recoveries. A crash of any component of the process such as the key-agreement layer, the Cliques library, or the group communication system is considered a process crash. It is assumed that the crash of one of these components is detected by all the other components and is treated as a process crash.

Also, the system is prone to partitions which may result a network being split into disconnected subnetworks. When such a partition is fixed, the disconnected components merge into a larger connected component. While processes are in separate disconnected components they cannot exchange messages.

We assume that message corruption is masked by a lower layer. Byzantine failures are not considered.

[2]GDH API also allows a key refresh operation which may be initiated only by the current controller.

Our intruder model takes into account only outside intruders, both passive and active. An outsider is anyone who is not a current group member. (Of course, any former and future member, is an outsider according to this definition.) We do not consider insider attacks since our threat model concentrates on the secrecy of group keys and the integrity of the group membership (i.e., the inability to spoof authenticated membership). Consequently, insider attacks are not relevant because a malicious insider can always reveal the group key and/or its own private key thus allowing for fraudulent membership authentication.

Passive outsider attacks involve eavesdropping with the aim of discovering the group key(s). This attack type has been proven to be computationally infeasible in [15]. Active outsider attacks involve injecting, deleting, delaying and modifying protocol messages. Some of these attacks aim to cause denial of service; we do not address these denial of service attacks. Attacks with the goal of impersonating a group member are prevented by the use of public key-based signatures. (All protocol messages are signed by the sender and verified by all receivers.) Other, more subtle, active attacks aim to introduce a known (to the attacker) or old key. These are prevented by the combined use of: timestamps, unique protocol message identifiers and sequence numbers (identifying the particular protocol run) in each protocol message.

3.2 Group Communication Model

A group communication system usually provides fundamental services such as membership as well as dissemination, reliability and ordering of messages. The membership service notifies the upper-level application with a list of group members each time the group changes. This notification-of-membership service is called a *view*. Every process that is part of the group communication system runs the membership algorithm and decides on the new view in agreement with other connected processes. Once this decision is made, the view is installed and the upper-level application is notified.

Several different sets of membership properties have been defined in the literature. Each provides a different set of semantic guarantees to the application, and are usually called Virtual Synchrony semantics or some variant on the name. The many variations of virtual synchrony are all based on the property that processes moving together from one view to another deliver the same set of messages in the former membership view.

Some group communication systems have been built [20], [22], [26] that approximate the virtual synchrony model along with some related properties. However, each system does not provide the exact same set of properties, and to the best of our knowledge a canonical "Virtual Synchrony model" of an entire system has not been defined in the literature. A good survey describing many of the variations of different properties for virtual synchrony semantics can be found in [38].

Virtual synchrony strengthens the shared state of the system by delivering messages in the same membership as they were sent in. This enables the use of a shared key to encrypt data, since the receiver is guaranteed to have the same membership view as the sender and therefore the same key (ignoring for now some constraints on rekeying).

This work assumes that the group communication system supports virtual synchrony semantics as they are defined below. The description of the properties is largely based on the survey [38] and the description of the Extended Virtual Synchrony semantics [24].

Note that we define that some event occurred in view v at process p if the most recent view installed by process p 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 the identifier id of v is greater than the identifier id' of 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 not sent twice. A message is not delivered twice 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) 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.
   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 is included in q's transitional set for this view.

8. *Virtual Synchrony*
   Two processes that move together[3] through two consecutive views deliver the same set of messages in the former.

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

10. *Agreed Delivery*
    1) Agreed delivery maintains causal delivery guarantees.
    2) If agreed messages m and m' are delivered at process p in this order, and m and m' are delivered by process q, then m' is delivered by q after it delivers m.
    3) If agreed messages m and m' are delivered by process p in view v in this order, and m' is delivered by process q in v before a transitional signal, then q delivers m. If messages m and m' are delivered by process p in view v in this order, 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.

11. *Safe Delivery*
    1) Safe delivery maintains 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.

[3]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.

4 A Basic Robust Algorithm

This section discusses the details of a basic robust key agreement algorithm. We describe the algorithm and prove its correctness, i.e. that the algorithm preserves the virtual synchrony semantics presented in Section 3.2. Throughout the remainder of the paper, we mean by the group communication system (GCS), a group communication system providing the virtual synchrony semantics.

4.1 Algorithm Description

Our basic algorithm is based on the Cliques GDH IKA.2 protocol. Briefly, this protocol works as follows (see [15] for a complete description):

When an additive group view change happens (a join or a merge) the current group controller generates a new key token by refreshing its contribution to the group key and passes the token to one of the new members. When that new member receives this token, it adds its own contribution and passes the token to the next new member[4]. Eventually, the token reaches the last new member. This new member, who is slated to become the new group controller, broadcasts the token to the group without adding its contribution. Upon receiving the broadcast token, each group member (old and new) factors out its contribution and unicasts the result (called a factor-out token) to the new controller. The new controller collects all the factor-out tokens, adds its own contribution to each of them, builds a list of partial keys and broadcasts the list to the group. Every member can then obtain the group key by factoring in its contribution. (This is actually performed with modular exponentiation.)

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

When some members leave the group, the group controller (who, at all times, is the most recent group member) removes their corresponding partial keys from the list of partial keys, refreshes each partial key in the list and broadcasts the list to the group. Each remaining member can then compute the shared key.

The algorithm described above 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 on this claim, consider what happens if a subtractive (leave or partition) group membership event occurs while the above protocol is in progress, for example, while the group controller is waiting for individual unicasts from all group members. Since the Cliques protocol is unaware of the membership change (which is ”visible” only to the group communication system), the group controller will not proceed until all factor-out tokens (including those from former members) are collected. Therefore, the system will block. Similar scenarios are also possible, e.g., if one of the new members crashes while adding its contribution to a group key. In this case, the token will never reach the new group controller and the protocol will, once again, simply block.

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

As the above examples illustrate, the protocol does not function correctly in the face of cascaded subtractive membership events. This behavior is not acceptable for reliable group communication systems that aim to provide a high degree of robustness and fault-tolerance.

A natural and correct solution to this problem is as follows: every time a group view change occurs, the group deterministically chooses a member (say, the oldest) and runs the Cliques GDH protocol with the chosen member initializing it. Note that this approach costs twice in computation and $O(n)$ more in the number of messages for the common case with no cascading membership events. This will be rectified in the second protocol described in Section 5.

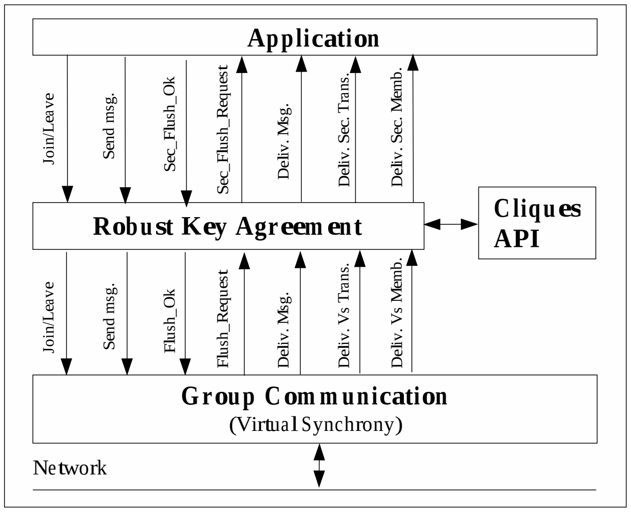


Figure 1. Secure group communication model

When the key-agreement protocol is integrated with a group communication system and virtual synchrony semantics must be preserved, extra care must be taken in order to provide all its guarantees to the application, including delivery of the correct views, transitional signal and transitional sets. We will elaborate on these issues later. Figure 1 presents the architecture of a secure group communication system. The system uses the following types of messages: Cliques messages (final_token_msg, partial_token_msg, key_list_msg, fact_out_msg), which are specific to the key agreement protocol (see [36]); membership notification messages (memb_msg); transitional signal messages (trans_signal_msg); application messages (data_msg); flush mechanism messages (flush_request_msg, flush_ok_msg).

To satisfy *Sending View Delivery* without discarding messages from live and connected members, a group communication system must block the sending of messages before the new membership is installed. In order to implement *Sending View Delivery* the group communication system sends a message (flush_request_msg) to the client asking for permission to install a new membership before actually creating the membership. The application responds with an acknowledgement message (flush_ok_msg) which follows all the messages sent by the application in the old view. After sending the acknowledgement message, the application is not allowed to send any messages until the new view is delivered. In Figure 1, the key-agreement algorithm interacts with both the application and GCS. The key-agreement algorithm implements the blocking mechanism transparently. When a flush_request_msg message is received from GCS, it is delivered to the user application. When the application acknowledgement message is received it is sent down to GCS.

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

The specification of the algorithm is defined in terms of the following received events which are associated with a specific group:

- Partial_Token: a partial token message (partial_token_msg) was received by the key-agreement algorithm from the GCS.
- Final_Token: a final token message (final_token_msg) was received by the key-agreement algorithm from the GCS.
- Fact_Out: a factor out message (factor_out_msg) was received by the key-agreement algorithm from the GCS.
- Key_List: a key list message (key_list_msg) was received by the key-agreement algorithm from the GCS.
- User_Message: a data application message (data_msg) was received by the key-agreement algorithm from the application. The user can send messages using broadcast or unicast services.
- Data_Message: a data application message (data_msg) was received by the key-agreement algorithm from the GCS.
- Transitional_Signal: a transitional signal message (trans_signal_msg) was received by the key-agreement algorithm from the GCS.
- Membership: a membership message (memb_msg) was received by the key-agreement algorithm from the GCS.
- Flush_Request: a flush request message (flush_request_msg) was received by the key-agreement algorithm from the GCS.
- Secure_Flush_Request: a flush request message (flush_request_msg) was received by the application from the key-agreement algorithm.
- Secure_Flush_Ok: a flush acknowledge message (flush_ok_msg) was received by the key-agreement algorithm from the application.

Note that the same type of message can be associated with different events, depending on the source of the message. For example, both Flush_Request and Secure_Flush_Request events are associated with a flush_request_msg message, but in the first case the message is received by the key-agreement algorithm from the application, while in the second case the message is received by the application from the key-agreement algorithm.

The algorithm consists of a state machine having the following states:

- SECURE (S): in this state the secure group is functional, all of the members have the group key and can communicate securely; the possible events are Data_Message, User_Message, Secure_Flush_Ok, Flush_Request, and Transitional_Signal; getting a Secure_Flush_Ok without receiving a Flush_Request is illegal; all other events are not possible.
- WAIT_FOR_PARTIAL_TOKEN (PT): in this state the process is waiting for a partial_token_msg message; the possible events are Partial_Token, Flush_Request and Transitional_Signal; User_Message and Secure_Flush_Ok are illegal; all other events are not possible.

- WAIT_FOR_FINAL_TOKEN (FT): in this state the process is waiting for a final_token_msg message; the possible events are Final_Token, Flush_Request and Transitional_Signal; User_Message and Secure_Flush_Ok are illegal; all other events are possible.

- COLLECT_FACT_OUTS (FO): in this state the process is waiting for $N-1$ fact_out_msg messages (where N is the size of the group); the only possible events are Fact_Out, Flush_Request, and Transitional_Signal; User_Message and Secure_Flush_Ok are illegal; all other events are not possible.

- WAIT_FOR_KEY_LIST (KL): in this state the process is waiting for a key_list_msg message; the possible events are Key_List, Flush_Request and Transitional_Signal; User_Message and Secure_Flush_Ok are illegal; all other events are not possible.

- WAIT_FOR_CASCADING_MEMBERSHIP (CM): in this state the process is waiting for are membership and transitional signal messages (memb_msg and trans_signal_msg); the possible events are Membership, Transitional_Signal, Data_Message (possible only the first time the process gets in this state), Partial_Token, Final_Token, Fact_Out and Key_List (they correspond to Cliques messages from a previous instance of the key agreement protocol when cascaded events happen); User_Message and Secure_Flush_Ok are illegal; all other events are not possible.

A process handles an event by performing two types of actions. The first type of action is a group communication operation and can be either a message delivery, or a message send such as unicast, broadcast, or send_flush_ok. The second type of action is a key agreement specific action. This translates into either computation (clq_first_member, clq_new_member, clq_update_ctx, clq_update_key, clq_factor_out, clq_merge) or access to Cliques state information (clq_destroy_ctx, clq_get_secret, clq_new_gc, clq_next_member). These primitives are part of the Cliques GDH API specification and are described in detail in [36]. We make use of a few trivial procedures:

- *alone*: given a membership notification for a group, which contains 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, FALSE otherwise;

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

- *is_in*: given an item and a set, returns TRUE if the set contains the item, FALSE otherwise;

- *empty*: given a set, returns TRUE if the set is empty, FALSE otherwise;

- *choose*: given a set, deterministically choose a member and returns that member;

- -: this is the subtraction operator for sets;

We also make use of 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 set of all the members of this view;

- *vs_set*, the transitional set associated with this notification;

- *merge_set*, the members from the new view that are not in the transitional set of the new view;

- *leave_set*, the members from the previous view that are not in the transitional set of the new view.

Group communication systems usually provide only the first three pieces of information in a membership notification. By using the membership set of the previous membership notification, and the current membership notification, the merge_set and leave_set can be computed by either the key-agreement algorithm or

```
New_membership.vs_set := EMPTY
New_membership.mb_set := Me
New_membership.merge_set := EMPTY
New_membership.leave_set := EMPTY
New_membership.mb_id := 0
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
```

Figure 3. Initialization of global variables

the GCS. To simplify the presentation of the pseudo-code of the algorithm we assume that the *merge_set* and *leave_set* are provided by the group communication system as part of the membership notification. The *Cliques_ctx* data structure is part of the Cliques GDH API specification, described in [36].

Every process executes the algorithm for a specific group and maintains a list of global variables. *Group_name* is the name of the group for which the algorithm is executed, *Group_key* is the shared secret of the group, while *Me* is the process executing the algorithm. The *Event* variable represents the current event handled. *Clq_ctx* keeps all the cryptographic context required by the Cliques API. *New_Membership* is the new membership that will be delivered, and *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*.

All global variables are written with capital letter, while all other variables are assumed to be local. Figure 3 shows the initialization of the global variables.

A diagram of the state machine is presented in Figure 2 and the corresponding pseudo-code in Figures 4, 5, 6, 7, 8, 9.

4.2 Correctness Proof

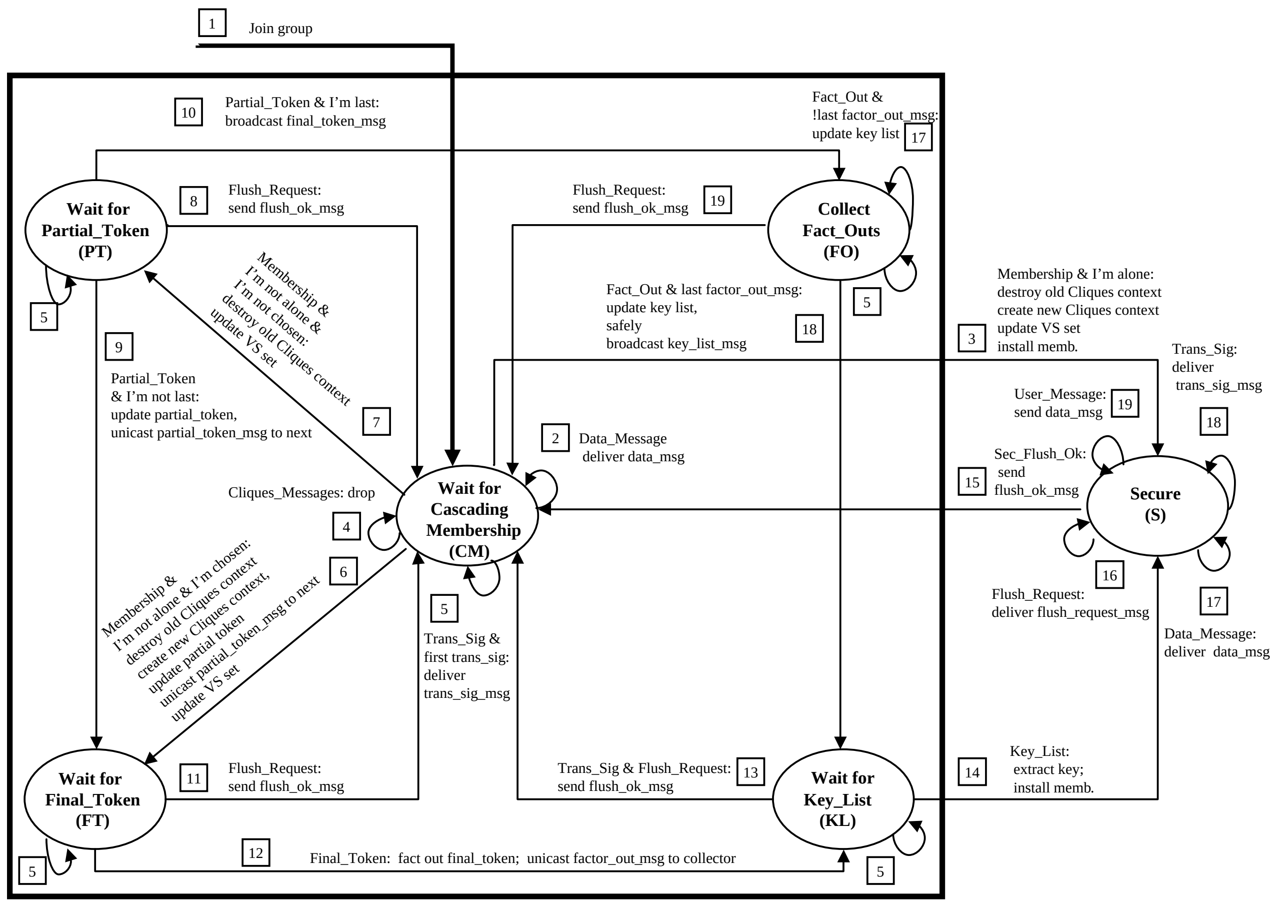
In this section we prove that the basic robust algorithm preserves the Virtual Synchrony Model described in Section 3.2. We assume that the underlying group communication layer provides the Virtual Syn-

```
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(Group_name)
     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:

 3→deliver(trans_signal_msg)
   First_transitional := FALSE
   VS_transitional := TRUE

All other events:

   not possible
```

Figure 4. Code executed in SECURE state

```
case Event is

Final_Token:

   fact_out_msg := clq_factor_out(Clq_ctx,
                                  final_token_msg)
   new_gc := clq_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(Group_name)
   State := WAIT_FOR_CASCADING_MEMBERSHIP

Transitional_Signal:

   if(First_transitional)
 3→  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
```

Figure 5. Code executed in WAIT_FOR_FINAL_TOKEN state


1 Join group
10 Partial_Token & I'm last: broadcast final_token_msg
Fact_Out & !last factor_out_msg: update key list 17
Wait for Partial_Token (PT)
8 Flush_Request: send flush_ok_msg
Flush_Request: send flush_ok_msg 19
Collect Fact_Outs (FO)
5
Membership & I'm not alone & I'm not chosen: destroy old Cliques context update VS set
Fact_Out & last factor_out_msg: update key list, safely broadcast key_list_msg 18
5
Membership & I'm alone: destroy old Cliques context create new Cliques context update VS set install memb. 3
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
11 Flush_Request: send flush_ok_msg
Trans_Sig & Flush_Request: send flush_ok_msg 13
14 Key_List: extract key; install memb.
Wait for Final_Token (FT)
Wait for Key_List (KL)
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
: A process can leave the group in any state

Figure 2. Basic algorithm

chrony Model. The Cliques protocol was proven to be correct in [16]. We also note that, as evident from the state machine in Figure 2, the Cliques GDH protocol remains intact, i.e., all of its protocol messages are sent and delivered in the same order as specified in [15]. Therefore, the basic robust key agreement algorithm provides the same security guarantees as the Cliques GDH protocol.

A secure membership notification is defined as a notification delivered by the key-agreement algorithm, and a VS membership notification is a notification delivered by the group communication system to the key-agreement algorithm. A secure view is a view installed by the key-agreement algorithm and a VS view is a view installed by the group communication system.

The following two lemmas are straightforward, but they are defined to clarify the proof. The first lemma is part of the VS Model provided by the group communication system. The second lemma is enforced by the key-agreement algorithm.

Lemma 4.1 *Every VS Membership event is preceded by the process sending a flush_ok_msg, with the exception of the case when a process joins a group. For a joining process, no flush_ok_msg message is sent and the membership notification is the first message delivered to it.*

Lemma 4.2 *A process is not allowed to send messages while it is performing the key agreement (this is between the time it sends a flush ok message until the time it receives a secure membership notification).*

Some useful observations can be made about membership notifications and application messages. The key-agreement algorithm discards VS membership events,

```
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,Group_name)
     State := COLLECT_FACT_OUTS
   endif

Flush_Request:

   send_flush_ok(Group_name)
   State := WAIT_FOR_CASCADING_MEMBERSHIP

Transitional_Signal:

   if(First_transitional)
3 →  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
```

Figure 6. Code executed in WAIT_FOR_PARTIAL_TOKEN state

not every VS view delivery event has a corresponding secure view delivery event. The secure membership notification is built and saved in the CM state (see Figure 9). 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_membership* variable. The only state in which a membership from the group communication system is received, is CM.

User messages are delivered immediately as they are received, they are not delayed or reordered. The only states which deliver user messages are S and CM.

We now prove the following lemmas.

Lemma 4.3 *The only state where VS membership notifications are received by the key-agreement algorithm is CM.*

Proof: By Lemma 4.1, 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

```
case Event is

Key_List:

   if(!VS_transitional)
     Clq_ctx := clq_update_ctx(Clq_ctx,
                               key_list_msg)
     Group_Key := clq_get_secret(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(Group_name)
     State := WAIT_FOR_CASCADING_MEMBERSHIP
   endif
   KL_got_flush_req := TRUE

Transitional_Signal:

   if(First_transitional)
3 →  deliver(trans_signal_msg)
     First_transitional := FALSE
   endif
   if(KL_got_flush_req)
     send_flush_ok(Group_name)
     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
```

Figure 7. Code executed in WAIT_FOR_KEY_LIST 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(SAFE, key_list_msg, Group_name)
     KL_got_flush_req := FALSE
     State := WAIT_FOR_KEY_LIST
   endif

Flush_Request:

   send_flush_ok(Group_name)
   State := WAIT_FOR_CASCADING_MEMBERSHIP

Transitional_Signal:

   if(First_transitional)
3 →  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
```

Figure 8. Code executed in COLLECT_FACT_OUTS state

CM state and does not leave the CM state until it receives a Membership event. A joining process starts executing the algorithm in the CM state and does not leave it until it receives a membership event.

Lemma 4.4 *The only states where user messages are received by the key-agreement algorithm from the group communication system are S and CM.*

Proof: After receiving a VS membership notification in the CM state (by Lemma 4.3 this is the only state where membership notifications are received) the process moves to one of the states FT, PT, FO, KL, or S. The transition to state S installs a new secure view, so in that state the process can send and receive user messages. In any of the FT, PT, FO, KL or CM states the process is not allowed to send application messages. If an application message is received in any of the FT, PT, FO or KL states, two cases are possible: first, this is a message sent in the previous secure view in state S, or second, this is a message sent by a process that completed the key agreement before this process did and already installed the new view and sent messages.

The first case is not possible because it implies that the group communication system delivered a user

```
Case Event is

Data_Message:

  deliver(data_msg)

Transitional_Signal:

   if(First_transitional)
3 →  deliver(trans_signal_msg)
     First_transitional := FALSE
   endif
   VS_transitional := TRUE

Membership:

   if(First_cascaded_membership)
4 →  VS_set :=  New_memb_msg.mb_set
     First_cascaded_membership := FALSE
   endif
5 →VS_set := VS_set − memb_msg.leave_set
   if(!empty(memb_msg.leave_set) &&
       First_transitional)
3 →  deliver(trans_signal_msg)
     First_transitional := FALSE
   endif
1 →New_memb_msg.mb_id := memb_msg.mb_id
2 →New_memb_msg.mb_set := memb_msg.mb_set
   if(!alone(memb_msg.mb_set))
     if(choose(memb_msg.mb_set) == Me)
       clq_destroy_ctx(Clq_ctx)
       Clq_ctx := clq_first_member(Me,
                                   Group_name)
       merge_set := memb_msg.mb_set − Me
       partial_token_msg := clq_update_key(
                               Clq_ctx, merge_set)
       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_name)
     Group_key := clq_extract_key(Clq_ctx)
     New_memb_msg.vs_set := Me
     deliver(New_memb_msg)
     First_transitional := TRUE
     First_cascaded_membership := TRUE
     State := SECURE
   endif
   VS_transitional := FALSE

Partial_Token, Final_Token, Fact_out, Key_List:

   ignore

User_Message, Secure_Flush_Ok:

   illegal, return an error to the user

All other events:

   not possible
```

Figure 9. Code executed in WAIT_FOR_CASCADING_MEMBERSHIP state

message not in the view in which it was sent, which contradicts the Sending View Delivery property. In the second case, note that the key list message is broadcast as a safe message. 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. This contradicts the Causal Delivery property. Therefore the only states where a process can receive user messages are S and CM.

4.2.1 Self Inclusion

Theorem 4.1 *When process p installs a secure view, the view includes p.*

Proof: By the protocol, we update the view-to-be-installed only when a membership notification is received from GCS (i.e., when we update *New_membership.mb_set* in Figure 9, Mark 2). By Lemma 4.3, this occurs only in the CM state.

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

4.2.2 Local Monotonicity

Lemma 4.5 *The identifier of a secure view is the identifier of the most recently installed VS view.*

Proof: By the protocol, we update the view-to-be-installed only when a membership notification is received from GCS (i.e., when we update *New_membership.mb_id* in Figure 9, Mark 1). By Lemma 4.3, this occurs only in the CM state.

By the algorithm, there are two transitions that install secure views. The first transition corresponds to a Membership event received in the CM state, indicating that process p is alone. In this case, the secure membership notification is immediately delivered with the most recent VS identifier. The second transition corresponds to a Key_List event received in the KL state. In this case, when the secure view is delivered, it indicates the most recent VS identifier.

Theorem 4.2 *If process p installs a secure view v_sec after installing a view v_sec' then the identifier of v_sec is greater than the identifier of v_sec'.*

Proof: The algorithm does not create or change view identifiers. It only uses the identifiers provided by the VS membership notifications without reordering them. By Lemma 4.5, p always delivers a secure view with the same identifier as the most recent VS identifier, therefore, it delivers a subsequence of the sequence of VS identifiers. Because it delivers a subsequence of VS identifiers and because GCS provides Local Monotonicity, the key-agreement algorithm provides Local Monotonicity too.

4.2.3 Sending View Delivery

Theorem 4.3 *A message is delivered by the key-agreement algorithm in the secure view that it was sent in.*

Proof: By the algorithm, messages are delivered by the key-agreement algorithm only in the S and CM states. In the S state, the secure view is the most recent VS view (i.e., when we update *New_membership.mb_set* in Figure 9, Mark 2). By the Sending View Delivery property of GCS, the above claim is true.

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 Figure 4). Since the key-agreement algorithm delivers a message immediately after it was received and GCS provides Sending View Delivery, all the messages sent in view v will be delivered before the next VS view was received, and therefore, before a new secure view is installed.

4.2.4 Delivery Integrity

Theorem 4.4 *If process p delivers a message m in a secure view v, then there exists a process q that sent m in v causally before p delivered m.*

Proof: This proof contains two parts. First, we show that the key-agreement algorithm delivers message m causally after it was sent. This is true by transitivity since:

- The key-agreement algorithm sends m immediately after it was sent by the application.
- By the delivery integrity property of GCS, the group communication system delivers message m causally after it was sent.
- The key-agreement algorithm delivers m immediately after it was received from the group communication system.

Note that the user messages are not reordered, they are delivered as soon as they are received.

Second, we show that if process p delivers message m in v, then there exists a process q that sent m in v. This claim is true by Theorem 4.3.

4.2.5 No Duplication

Theorem 4.5 *A message is not sent twice using the key-agreement algorithm. The key-agreement algorithm does not deliver a message twice to the same process.*

Proof: By the algorithm, user messages are sent only in the S state, when the application is sending them, so a message is not sent twice.

By the algorithm, messages are delivered only in the S and CM states. They are delivered immediately upon receipt from the group communication system. Since GCS guarantees no duplication, it can not be that a message sent once to the group communication system is received twice. Note that the key-agreement algorithm generates Cliques messages, but these are never delivered to the application so they do not affect the No Duplication property.

4.2.6 Self Delivery

Theorem 4.6 *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 group communication system and the key-agreement algorithm never discards application messages and it delivers them immediately after receiving them. Since GCS provides Self Delivery, the theorem is true.

4.2.7 Transitional Set

Lemma 4.6 *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 a safe message, by the Safe Delivery properties of GCS, q must also receive the key_list_msg message before the transitional signal. Therefore, q must also have installed v_sec.

Theorem 4.7 *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, we only remove from this set members reported by VS membership notifications as not being in our VS transitional set (i.e. the *leave_set*), and we never add members to the transitional set. Due to this, if q is included in p's secure transitional set then q must have been included in all of p's VS transitional sets since the last secure view delivered at p. Additionally, p and q must have installed the same sequence of VS views prior to v_sec because they both installed the VS view corresponding to v_sec and because of the GCS transitional set property number two.

Therefore, by Lemma 4.6, q must have installed the same previous secure view as p. To show that q installed no intervening 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 4.8 *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: If p and q install the same secure view, and q is included in p's transitional set for this view, but p is not included in q's transitional set for this view, two cases are possible. First, q's previous secure view was not the same as p's secure view. In this case, by theorem 4.7, q is not included in p's transitional set, contradicting our assumption that q is included in p's transitional set.

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.

4.2.8 Virtual Synchrony

Theorem 4.9 *Two processes p and q that move together through two consecutive secure views, deliver the same set of messages in the former view.*

Proof: By Lemma 4.4, user messages are received by the key-agreement algorithm only in the S or CM states and as specified by the protocol, they are delivered as soon as they are received. Therefore, user messages are delivered only in the S and CM states.

By Lemma 4.3, VS membership notifications are received only in the CM state. By the protocol (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 $v2$ [5].

Therefore, by the Virtual Synchrony property guaranteed by GCS, 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 CM state and by Lemma 4.2, 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 $v1\_sec$.

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

4.2.9 Causal Delivery

Lemma 4.7 *All the messages delivered by the key-agreement algorithm, support the ordering properties with which they were delivered by the group communication system.*

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

Theorem 4.10 *If message m causally precedes message m', and both are sent in the same secure view, then any process q that delivers m' delivers m before m'.*

Proof: This is true by Lemma 4.7.

4.2.10 Agreed Delivery

Theorem 4.11 *If messages m and m' are delivered at process p in this order, and m and m' are delivered by process q then m' is delivered by q after m is delivered by q.*
If messages m and m' are delivered by process p in secure view $v1\_sec$ in this order, and m' is delivered by process q in secure view $v2\_sec$ and message m was sent by a process r which is a member of secure view $v2\_sec$, then q delivered m.

Proof: This is true by Lemma 4.7 and because the secure transitional set is the intersection of all the VS transitional sets.

4.2.11 Safe Delivery

Theorem 4.12 *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.*

Proof: By Lemma 4.7, key-agreement delivers messages with the same ordering guarantees with which they were delivered by the GCS.

By the algorithm, the first transitional signal received from GCS is delivered to the application (see Mark 3 in Figures 4, 7, 6, 5, 8, 9).

By the algorithm, the transitional set delivered with a new secure membership in calculated as follows, when a group change happens while the group is stable (state S), the transitional set is initialized to the current secure membership list (see Mark 4, in Figure 9) and then every time another membership happens before installing this secure membership, the members that left the group are removed from the transitional set, such that when the new secure membership is delivered, the transitional set is correct. Therefore, the safe delivery requirements are preserved.

5 An Optimized Robust Algorithm

In this section we show how the algorithm presented in the previous section can be optimized, such that the price paid for handling common, non-cascaded events is lower, while preserving the same set of group communication semantics 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 from the group communication system, the algorithm ignores all the previous key agreement information and starts the merge protocol choosing a member from the new group to initialize it. Therefore, this algorithm pays more than necessary for computing a group key in a regular case, because it does not distinguish between a membership that finished without being interrupted and a cascaded membership.

The algorithm described above can be optimized so that it distinguishes between these two cases. Every time the group view changes, the algorithm detects the cause of the group change (join, leave, partition, merge or a combination of partition and merge) and invokes the Cliques GDH specific protocol. For example, in the case where a leave occurred, the leave protocol is invoked. Computing a new key in the case that a leave or partition occurred, requires only one broadcast. Thus, leave events can be handled immediately with a lower communication and computation cost than the basic algorithm required.

In the optimized key-agreement algorithm the process still starts executing the state machine by invoking the *Join* primitive. Also, at any moment, a process can voluntarily leave the algorithm by invoking the *Leave* primitive.

The optimized algorithm utilizes the following two states in addition to those of the basic algorithm:

- WAIT_FOR_SELF_JOIN (SJ): this is the initial state in which a process that joined a group enters the state machine; the process is waiting for the membership message that notifies the group about its joining. In case a network event happens between the join request and the membership notification delivery, the GCS will report the cause of the group change as being a network event and the transitional set will contain only the joining member. The only possible event is a Membership. User_Message and Secure_Flush_Ok events are illegal. An error will be returned to the user if they are attempted. All other events are not possible.

- WAIT_FOR_MEMBERSHIP (M): in this state the process is waiting for a membership notification. The possible events are: Transitional_Signal, Data_Message and Membership. The membership notification can be caused by voluntarily events such as join or leave, or network events. User_Message and Secure_Flush_Ok events are illegal. An error will be returned to the user. All other events are not possible.

While a process starts the basic algorithm in the CM state, in the optimized algorithm a process starts the algorithm in state SJ. From the stable state (S state) if the group changed the process moves to the M state instead of moving to the CM state as in the basic algorithm. From here, depending on the cause of the group change, the merge or the leave Cliques GDH protocols are invoked. Also, a combined network event which includes both joins and leaves simultaneously can be handled by a modified version of the Cliques GDH merge protocol (as described in Section 5.2). If another group change happens before a key is computed, the process will move to the CM state and execute the basic algorithm.

The *merge_set* and *leave_set* fields of the membership notification can be used to determine the cause of the group view change. In addition, we use a modified version of the procedure *clq_update_key* procedure which can handle combined network events.

The diagram of the state machine of the algorithm is presented in Figure 12 and the corresponding pseudocode in Figures 4,5, 6, 7, 8, 9, 10, 11.

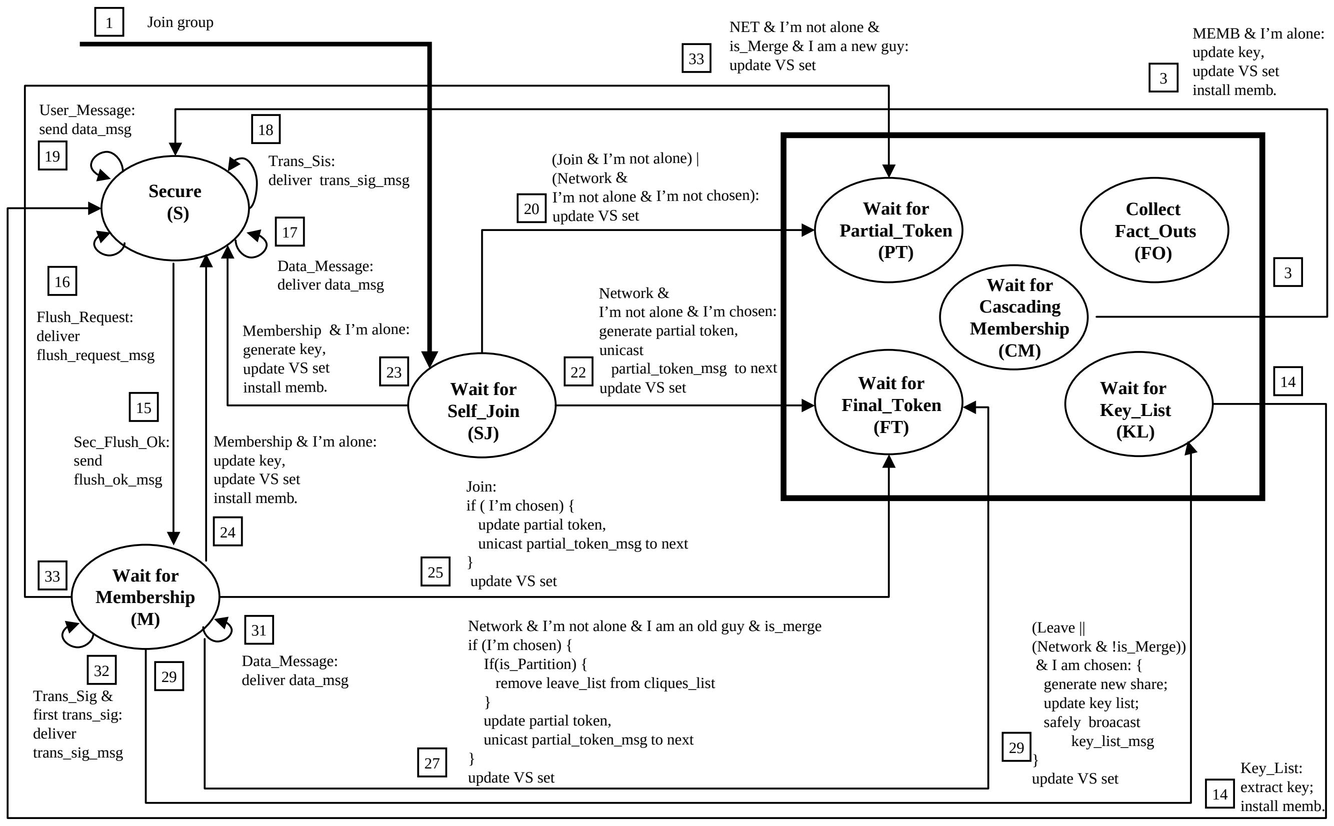
5.2 Handling Bundled Events

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

As mentioned above, Cliques provides two separate protocols that handle leave and merge events. Each of these protocols can trivially handle bundled events of the same type, i.e., the Cliques merge protocol can accommodate any combination of bundled merges and the Cliques leave protocol can do the same for any combination of leaves and partitions. A more interesting scenario occurs when a single membership event bundles merges/joins with leaves/partitions. One obvious way to handle this type of event is to first in-

```
Case Event is

Membership:
 4
──→VS_set :=  New_memb_msg.mb_set
 1
──→New_memb_msg.mb_id := memb_msg.mb_id
──→New_memb_msg.mb_set := memb_msg.mb_set
   First_cascaded_membership := FALSE
   if(!alone(memb_msg.mb_set))
     if(choose(memb_msg.mb_set) = Me)
       Clq_ctx := clq_first_member(Me,
                                    Group_name)
       merge_set := memb_msg.merge_set
       partial_token_msg := clq_update_key(
                             Clq_ctx,merge_set)
       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_name)
     Group_key := clq_extract_key(Clq_ctx)
     New_memb_msg.vs_set := Me
     deliver(New_memb_msg)
     First_cascaded_membership := TRUE
     State := SECURE
   endif
   VS_transitional := FALSE

User_Message, Secure_Flush_Ok:

   illegal, return an error to the user

All other events:

   not possible
```

Figure 10. Code executed in WAIT_FOR_SELF_JOIN state


1
Join group
33
NET & I'm not alone & is_Merge & I am a new guy: update VS set
3
MEMB & I'm alone: update key, update VS set install memb.
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
20
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
22
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.
24
Join: if (I'm chosen) { update partial token, unicast partial_token_msg to next } update VS set
25
33
Wait for Membership (M)
31
Data_Message: deliver data_msg
32
29
Trans_Sig & first trans_sig: deliver trans_sig_msg
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
27
(Leave || (Network & !is_Merge)) & I am chosen: { generate new share; update key list; safely broacast key_list_msg } update VS set
29
14
Key_List: extract key; install memb.

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 a safe message.
: A process can leave the group in any state

Figure 12. Optimized algorithm

voke Cliques leave to process all leaves/partitions and then invoke Cliques merge to process joins/merges. However, this is inefficient since the group would essentially perform two separate key agreement protocols where only one is truly needed. We can take advantage of the fact that both protocols in Cliques are initiated by the group controller. 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.

5.3 Correctness Proof

In this section we prove that the optimized algorithm described above provides the virtual synchrony semantic presented in Section 3.2.

We note that the optimized algorithm state machine utilizes two more states that can install secure memberships: SJ and M. In both of these states a secure view can be installed only in the special case when the group consists of only one member so the process can compute a key and install the secure memberships.

Unlike the basic algorithm where application messages were delivered only in states S and CM, in the optimized algorithm application messages are delivered in the S and M states. Membership notifications are received in the CM, SJ, and M states.

It can be noticed that for the optimized algorithm, a process starts the algorithm in the SJ state. Also, from the stable state (the S state), due to a group change notification that triggers the key-agreement algorithm, instead of moving to the CM state as in the basic algorithm, the process moves to the M state. A process can

```
Case Event is

Data_Message:

   deliver(data_msg)

Transitional_Signal:

   if(First_transitional)
3→   deliver(trans_signal_msg)
     First_transitional := FALSE
   endif
   VS_transitional := TRUE

Membership:

4→ VS_set :=  New_memb_msg.mb_set
5→ VS_set := VS_set - memb_msg.leave_set
1→ New_memb_msg.mb_id := memb_msg.mb_id
2→ 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(SAFE,key_list_msg,Group_name)
       endif
       State := WAIT_FOR_KEY_LIST
     else
       if(is_in(chosen(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)
           State := WAIT_FOR_FINAL_TOKEN
         endif
       else /* new member */
         clq_destroy_ctx(Clq_ctx)
         Clq_ctx := clq_new_member(Me)
         State := WAIT_FOR_PARTIAL_TOKEN
       endif
   else /* alone */
     Clq_ctx := clq_first_member(Me, Group_name)
     Group_key := clq_extract_key(Clq_ctx)
     New_memb_msg.vs_set := Me
     deliver(New_memb_msg)
     First_transitional := TRUE
     First_cascaded_membership := TRUE
     State := SECURE
   endif
   VS_transitional := FALSE

User_Message, Secure_Flush_Ok:

   illegal, return an error to the user

All other events:

   not possible
```

Figure 11. Code executed in WAIT_FOR_MEMBERSHIP state

move in the CM state only from the PT, FT, FO, and KL states. From the moment that the process moves to the CM state, it executes the basic algorithm.

For the rest of the proof, we make use of Lemmas 4.1 and 4.2 which are still valid. The first lemma is a property of the underlying group communication layer and the second one is enforced by the key-agreement algorithm. We also note that Lemma 4.2 which specifies that a process is not allowed to send user messages while performing the key-agreement algorithm, enforces that a process can not send user messages in any of the M, CM, PT, FT, FO, or KL states.

Lemma 5.1 *The only states where VS membership notifications are received are the SJ, CM and M states.*

Proof: By Lemma 4.1, 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 either the M or CM states and does not leave these states until it receives a Membership event. A joining process starts executing the algorithm in the SJ state and does not leave it until it receives a membership event.

Lemma 5.2 *The only states where user messages can be received are S and M.*

Proof: By the protocol, in the S state the process can send and/or receive messages. By Lemma 4.1, the first message that a process receives in the SJ state is the VS membership notification which triggers immediately a transition to another state, so no user messages are received in the SJ state.

After receiving a VS membership notification from GCS in any of the SJ, M or CM states (by Lemma 5.1 these are the only states 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 M, 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 CM, FT, PT, FO or KL states, two cases are possible: first, this is a message sent in the previous secure view in state S, or second, this is a message sent by a process that completed the key-agreement algorithm before this process did and already installed the new view and sent messages.

The first case is not possible because it implies that the group communication system delivered a user message not in the view in which it was sent, which contradicts the Sending View Delivery property. In the second case, note that the key list message is broadcast as a safe message. 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. This contradicts the Causal Delivery property. Therefore the only states where a process can receive user messages are S and M.

5.3.1 Self Inclusion

Theorem 5.1 *When process p installs a secure view, the view includes p.*

Proof: By the protocol, we update the view-to-be-installed only when a membership is received from GCS (i.e., when we update *New_membership.mb_set* in Figures 9, 11 and 10, Mark 2). By Lemma 5.1, this occurs only in the SJ, M and CM states.

By the algorithm, there are four transitions that install secure views. The first three transitions correspond to a Membership event received in the CM, M, or SJ states, indicating that process p is alone. In this case, the secure membership is immediately delivered with p (the only one) in it.

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

5.3.2 Local Monotonicity

The proof of this property is very similar to the one we gave for the basic algorithm. It is enough to show that Lemma 4.5 is still true for the optimized protocol, therefore, by Theorem 4.2, Local Monotonicity is provided by the key-agreement algorithm.

Lemma 5.3 *The identifier of a secure view is the identifier of the most recently installed VS view.*

Proof: By the protocol, we update the view-to-be-installed only when a membership is received from GCS (i.e., when we update *New_membership.mb_id* in Figures 9, 10, and 11, Mark 1). By Lemma 5.1, this occurs only in the CM, SJ, and M states.

By the algorithm, there are four transitions that install secure views. The first three transitions correspond to a Membership event received in one of the CM, SJ, or M states, indicating that process p is alone. In this case, the secure membership is immediately delivered with the most recent VS identifier.

The fourth transition corresponds to a Key_List event received in the KL state. In this case, when the view is delivered, it indicates the most recent VS identifier.

5.3.3 Sending View Delivery

Theorem 5.2 *A message is delivered by the key-agreement algorithm in the secure view that is was sent in.*

Proof: By the algorithm, messages are delivered only in the S and M states. In the S state, the secure membership is the most recent VS membership (i.e., when we update *New_membership.mb_set* in Figure 9, mark 2). By the GCS Sending View Delivery property, all the messages sent in the S state are delivered in the secure view that they were sent in.

By the algorithm, a process moves to the M state after the application agreed to close the membership by sending a flush_ok message (see Figure 4). The group communication system guarantees that before delivering the new VS view, it will deliver all the messages that were sent in the previous view. Since the key-agreement algorithm delivers a message immediately after it was received and GCS provides Sending View Delivery then all the messages sent in view v will be delivered before the next VS view was received, and therefore, before a new secure view is installed.

5.3.4 Delivery Integrity

Theorem 5.3 *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.*

Proof: The proof is identical to the proof in the case of the basic algorithm.

5.3.5 No Duplication

Theorem 5.4 *A message is not sent twice using the key-agreement algorithm. The key-agreement algorithm does not deliver a message twice to the same process.*

Proof: The proof is very similar to the one in the case of the basic algorithm.

5.3.6 Self Delivery

Theorem 5.5 *If process p sends a message m, then p delivers m unless it crashes.*

Proof: The proof is identical to the one we gave for the basic algorithm.

5.3.7 Transitional Set

We remark that in the optimized protocol, when a secure view changes, the first VS view notification is received in the M state while later cascaded VS memberships are received in the CM state. The computation of the secure transitional set is the same as for the basic algorithm. Therefore, the arguments we provided to prove Lemma 4.6 and Theorems 4.7 and 4.8 are still valid.

5.3.8 Virtual Synchrony

Theorem 5.6 *Two processes p and q that move together through two consecutive secure views, deliver the same set of messages in the former view.*

Proof: By Lemma 5.2, user messages are received by the key-agreement algorithm from GCS only in the S or M states and as specified by the protocol, they are delivered as soon as they are received. Therefore, user messages are delivered to the application only in the S and M states.

By Lemma 5.1, VS membership notifications are received only in the SJ, CM and M states. By the protocol (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]. 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 Virtual Synchrony property guaranteed by GCS, 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 by Lemma 4.2, 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 $v1\_sec$.

5.3.9 Causal Delivery

Theorem 5.7 *If message m causally precedes message m', and both are sent in the same secure view, then any process q that delivers m' delivers m before m'.*

Proof: The proof is identical to the one provided for the basic algorithm.

5.3.10 Agreed Delivery

Theorem 5.8 *If messages m and m' are delivered at process p in this order, and m and m' are delivered by process q then m' is delivered by q after m is delivered by q.*

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: The proof is identical to the one provided for the basic algorithm.

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

5.3.11 Safe Delivery

Theorem 5.9 *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 unless it crashes.*

Proof: By Lemma 4.7, the key-agreement algorithm delivers messages with the same ordering guarantees with which they were delivered by the group communication system.

By the algorithm, the first transitional signal received from the group communication system is delivered to the application (see Mark 3 in Figures 4, 7, 6, 5, 8, 9 and 11.

By the algorithm, the transitional set delivered with a new secure membership notification is calculated as follows, when a group change happens while the group is stable (state S), the transitional set is initialized to the current secure membership list (see Figures 10 and 11, Mark 4) and then every time another VS membership notification is received from GCS before installing this secure view, the members that left the group are removed from the transitional set (see Figure 9 and 11, Mark 5), such that when the new secure membership notification is delivered, the transitional set is correct. Therefore, the safe delivery requirements are preserved.

6 Conclusions

Implementing secure and robust handling of cascading group events, using an approach optimized for the most frequent events (join and leave), is crucial in order to have a complete secure group communication system. Hardening security protocols to make them robust to asynchronous network events although difficult is possible. This work provides two robust key agreement algorithms. We prove that by integrating them with a group communication systems supporting Virtual Synchrony, the group communication membership and ordering guarantees are preserved.

We intend to implement the optimized protocol in the Secure Spread system. In addition we intend to explore and experiment with robustness and recovery techniques for a spectrum of other group key management mechanisms, such as the centralized approach and the Burmester-Desmedt protocol.

Finally, several necessary services for a secure group communication could lead to interesting future work. They include services such as group member certification, intra-group authentication, private communication within a group and private communication between members and non-members of the group.

References

[1] N. Asokan, V. Schoup, and M. Waidner, "Optimistic fair exchange of digital signatures," *IEEE Journal on Selected Area in Communications*, 2000.

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

[3] Y. Amir, G. Ateniese, D. Hasse, Y. Kim, C. Nita-Rotaru, T. Schlossnagle, J. Schultz, J. Stanton, and G. Tsudik, "Secure group communication in asynchronous networks with failures: integration and experiments," in *Proceedings of the 20th IEEE International Conference on Distributed Computing Systems*, (Taipei, Taiwan), pp. 330–343, April 2000.

[4] H. Harney and C. Muckenhirn, "Group key management protocol (gkmp) specification," Tech. Rep. RFC 2093, IETF, July 1997.

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

[6] T. Ballardie, "Scalable multicast key distribution," Tech. Rep. RFC 1949, IETF, 1996.

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

[8] S. Mittra, "Iolus: A framework for scalable secure multicasting," in *Proceedings of the ACM SIGCOMM '97*, September 1997.

[9] T. Hardjono, B. Cain, and I. Monga, "Intradomain group key management protocol," tech. rep., draftietfipsecintragkm00.txt (Work in Progress), November 1998.

[10] C. K. Wong, M. G. Gouda, and S. S. Lam, "Secure group communications using key graphs," in *Proceedings of the ACM SIGCOMM '98*, pp. 68–79, 1998.

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

[12] A. Fiat and M. Naor, "Broadcast encryption," *Advances in Cryptology - CRYPTO'93*, August 1993.

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

[14] M. Just and S. Vaudenay, "Authenticated multiparty key agreement," *Advances in Cryptology - EUROCRYPT'96*, May 1996.

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

[16] G. Ateniese, M. Steiner, and G. Tsudik, "New multi-party authentication services and key agreement protocols," *IEEE Journal of Selected Areas in Communication*, vol. 18, March 2000.

[17] R. Poovendran, S. Corson, and J. Baras, "A shared key generation procedure using fractional keys," *IEEE Milcom 98*, October 1998.

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

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

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

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

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

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

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

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

[26] I. Keidar, K. Marzullo, J. Sussman, and D. Dolev, "A client-server oriented algorithm for virtually synchronous group membership in wans," Tech. Rep. CS99-623, Univ. of California, San Diego Tech Report, June 1999.

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

[28] O. Rodeh, K. Birman, M. Hayden, Z. Xiao, and D. Dolev, "Ensemble security," Tech. Rep.

TR98-1703, Cornell University, Department of Computer Science, September 1998.

[29] K. Birman, M. Hayden, O. Ozkasap, Z. Xiao, M. Budiu, and Y. Minsky, "Bimodal multicast," Tech. Rep. TR99-1745, Department of Computer Science, Cornell University, May 1999.

[30] Y. Amir and J. Stanton, "The spread wide area group communication system," Tech. Rep. 98-4, Johns Hopkins University Department of Computer Science, 1998.

[31] Y. Amir, C. Danilov, and J. Stanton, "A low latency, loss tolerant architecture and protocol for wide area group communication," in *Proceedings of the International Conference on Dependable Systems and Networks*, pp. 327–336, June 2000.

[32] Y. Amir, *Replication using Group Communication over a Partitioned Network*. PhD thesis, Institute of Computer Science, The Hebrew University of Jerusalem, Jerusalem, Israel, 1995.

[33] A. Fekete, N. Lynch, and A. Shvartsman, "Specifying and using a partionable group communication service," in *Proceedings of the 16th annual ACM Symposium on Principles of Distributed Computing*, (Santa Barbara, CA), pp. 53–62, August 1997.

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

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

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

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

[38] R. Vitenberg, I. Keidar, G. V. Chockler, and D. Dolev, "Group communication specifications: A comprehensive study," Tech. Rep. CS0964, Computer Science Department, the Technion, Haifa, Israel; Tech. Rep. MIT-LCS-TR-790,Massachusetts Institute of Technology, Laboratory for Computer Science; Tech. Rep. CS99-31, Institute of Computer Science, The Hebrew University of Jerusalem., 1999.