Towards Adaptive Secure Group Communication: Bridging the Gap between Formal Specification and Network Simulation

Sebastian Gutierrez-Nolasco,
Nalini Venkatasubramanian
School of Inf. and Comp. Science,
University of California Irvine,
Irvine, CA 92697, USA
{seguti,nalini}@ics.uci.edu

Mark-Oliver Stehr,
Carolyn Talcott
Computer Science Laboratory,
SRI International,
Menlo-Park, CA 94025, USA
stehr@csl.sri.com, clt@cs.stanford.edu

***Abstract*— Traditionally, adaptability in communication frameworks has been restricted to predefined choices without taking into consideration tradeoffs between them and the application requirements. In this paper we extend an executable specification of a state-of-the-art secure group communication subsystem to explore two dimensions of adaptability, namely security and synchrony under crash-recovery and intermittent connectivity scenarios. In particular, we relax the traditional requirement of virtual synchrony (a well-known bottleneck) and propose various generic optimizations, while preserving essential security guarantees. In order to evaluate how practical and effective our generic optimizations are, we integrate the specification into ns2, bridging the gap between formal specification and classical network simulation.**

I. Introduction

In recent years some secure group communication systems (GCS) have been developed [1]–[5] and several useful techniques have been proposed to deal with scalability, performance and security in peer groups with dynamic membership and decentralized control [6]–[8]. However, GCS were designed to be highly efficient in local (wired) networks, assume a relatively small group size (up to few hundred), and do not consider mobility, temporary disconnections and real time constraints. In particular, scalability and high performance are both currently achieved via the light-weight/heavy-weight model [9], [10], where powerful servers (daemons) residing in each host execute relatively expensive distributed protocols and several clients can connect to a server to share the GCS services on each host.

The next generation of adaptable GCS is driven by constantly changing application requirements, real-time data delivery, intermittent membership changes due to temporary disconnections and mobility patterns, performance requirements and non-uniform security and fault tolerance levels. Due to the high computational overhead of public key cryptography, symmetric keys are commonly used to encrypt the data. To fully exploit the multicasting nature, a shared group key is typically considered to be the most efficient solution. Consequently, the main problem now becomes the efficient establishment and management of keys. Secure Spread [11], a state-of-the-art GCS, uses key establishment protocols that stall all communication (at the application level), while the key is generated and rely on strong synchronization guarantees to assure that no member can receive and decrypt messages after he left the group (*forward secrecy*) and no new member can receive and decrypt messages sent before he joined the group (*backward secrecy*). However, in many applications, disconnections are common and expected and data in transit must not only be protected against unauthorized users, but also must be delivered in a timely manner, so that decisions can be made from accurate and fresh data. Triggering a blocking rekey after every join or leave (to preserve forward and backward secrecy) may preclude timely delivery of sensitive information and even may lead to potential denial of service attacks if a trusted member is compromised and joins and leaves the group intermittently. In this case, it would be desirable to employ a less constrained GCS that does not require the generation of a new key after every join or leave, but still maintains a certain degree of security. In fact, we believe that an application should be able to tailor the secure GCS according to its needs not only in terms of security but also synchrony, timeliness and reliability, because there is no one-size-fits-all solution.

In this paper we study two dimensions of adaptability, namely security and synchrony in the presence of intermittent failures, formalize adaptation rules and establish key ordering and security properties. Furthermore, we experimentally evaluate the overhead and cost of adaptation in secure group communication and our quantitative results illustrate how formal prototyping and classical network simulation complement each other. Our starting point is a formal prototype of the Secure Spread GCS, which we have generalized along various lines to support secure communication with fewer synchronization constraints and adaptability along several new dimensions. In particular, our approach opens a spectrum of new security guarantees, which are weaker than in the synchronized case, but still sufficient for many applications. Thanks to the use of abstract APIs, our generalizations are to a large degree independent of the group communication system and the key establishment algorithm, and hence can be combined with improvements along other dimensions, such as the choice of specific group communication protocols and key establishment protocols. The use of formal prototyping techniques based on the executable specification language Maude enabled us to explore and validate design decisions without the need to carry out a full-fledged implementation.

II. State of the Art in GCS

After a brief explanation of the relevant group communication system semantics, this section gives an overview of a state-of-the-art group communication system (Spread) and a framework for key establishment protocols (Cliques), and discusses how these components are assembled to provide a secure group communication architecture (Secure Spread). In this paper we use Spread and Secure Spread without further qualification to refer to the publicly released versions that can be found at `http://www.spread.org/`.

A. Semantics of Group Communication

The most well-known group communication model is the *virtual synchrony semantics* (VS semantics) [9] which was originally developed for Isis/Horus [12], a primary component GCS, but later extended to partitionable GCS. One of these extensions is the *extended virtual synchrony semantics* (EVS semantics) [13], a model that extends the virtual synchrony model of Isis to support continued operation in all components of a partitioned network. The central concept of group communication is that of a *view*, *i.e.* a snapshot of membership in a group. In each execution of a partitionable GCS, views and transitions between them form a partial order. Both, the VS and the EVS semantics, share the key property of *virtual synchrony*, namely that every two processes that participate in the same two consecutive view changes, deliver the same set of messages between the two changes.

Virtual synchrony, however, is only one property of the VS semantics. The VS semantics furthermore ensures that messages are delivered in the same view they were sent in (sending view delivery). To accomplish this, an extra round of acknowledgment messages is needed every time before a view change, preventing applications to send other messages until the next view is installed. Furthermore, the VS semantics is a closed group semantics, allowing only current members of the group to send messages to the group.

The EVS semantics, on the other hand, allows message delivery in a different view than it was sent in, as long as the message is delivered in the same view to all members (same view delivery). Consequently, the synchronization phase which allows the application to be aware of the sending view is not needed in the EVS semantics. The EVS semantics also allows open groups, where non-members of the group can send messages to a group.

B. Spread

The Spread group communication system [14] emerged from the work on Transis [15] and Totem [16] and has been designed to cope with node failure and network partitions. Spread supports the EVS semantics and provides different levels of service with different reliability and ordering guarantees: Messages can be reliable, fifo, causally ordered, totally ordered (also called agreed), or safe, where the later means that messages are only delivered if it is known that everybody in the group has actually received it.

The Spread architecture consists of two layers, which are correspondingly reflected in our formal specification: the heavy-weight group layer and the light-weight group layer. The heavy-weight group layer provides extended virtual synchrony semantics at the level of the *physical group*, *i.e.* the group of hosts (servers). Due to changing network connectivity, we are really concerned with snapshots of group membership, which are called *configurations*. This layer provides services to multicast data messages which should be sent ideally to every host and to retrieve messages that have been delivered to the application, which can be either application data messages or messages that represent configuration change events.

The primary mode of operation is to deliver messages to all hosts which are part of the most recently established regular configuration. According to the EVS semantics all messages should be delivered at each of these hosts in the same regular configuration or the following transitional configuration (see below). This delivery is furthermore subject to ordering constraints that depend on the service level that was requested when the message was sent. In the case of safe messages, it is also subject the constraint that every host in the configuration has received this message, and hence can deliver it unless it crashes. If a change in the connectivity is detected, two different configuration change events are generated: First, there is an event to introduce a transitional configuration, which is a reduced configuration in which certain messages can be delivered that could not be delivered in the previous regular configuration. After this transitional phase, a new regular configuration is introduced which reflects the new connectivity of the network. The light-weight group layer provides EVS semantics at the level of logical groups, *i.e.* groups of agents (clients), simply called groups in the following. Groups are identified by names and the different snapshots of group membership are called views. The API is similar to that at the heavy-weight group layer, except that messages and changes refer to groups instead of configurations, but in addition the API offers two new services at this level: A client can request to join or leave a group, and in response Spread generates corresponding group change events when the actual transition to the new view has occurred.

It is worth to emphasize that in the EVS semantics the application cannot determine or even know the view in which the message is sent by the GCS. The application passes messages to the GCS where they can be buffered. Hence, the most recently established view at the time when the application sends the message is not necessarily the view in which the message is sent out by the GCS, let alone the view when the message is delivered to the receiving application.

C. Secure Spread

Secure Spread [11] provides secure group communication for closed groups and can operate with different protocols that establish a single key shared by all members of the current view. Secure Spread is built on top of Flush Spread [17] and the Cliques toolkit [8]. Flush Spread has a similar functionality as Spread but provides the stronger virtual synchrony semantics, which requires acknowledgments by all members for each view change. In [17] it is explained how VS semantics can be implemented using the weaker EVS semantics. The Flush Spread implementation is essentially a refinement of these ideas. The Cliques toolkit provides a generic API and implementations of various group key agreement protocols, among them the Group-Diffie-Hellman protocol (GDH) [8] and a tree-based variant (TGDH). Authentication is not provided by the key agreement protocol, but instead all messages are authenticated using digital signatures. An interesting feature of GDH and its variants is that they are contributory, which means that every member contributes a key share, but the entire key is never transmitted over the channel (not even in encrypted form). However, this leads to the essential requirement that all members actively participate in the key agreement.

Secure Spread simply uses the underlying Flush Spread to exchange the messages required and produced by the Cliques toolkit, whenever a group change occurs. If the key agreement is itself interrupted by a new group change the Cliques protocol is restarted. Furthermore, Secure Spread implements some optimizations allowing several subsequent joins and leaves to be batched into a single call of the delete/merge subprotocol.

III. Formal Methodology

The general methodology we employ for system design and analysis is based on an executable specification language called Maude [18]. Its theoretical foundation is rewriting logic [19], a logic with an operational as well as a model-theoretic semantics. Formal prototyping is a key ingredient of our methodology, which allows us to experiment with an abstract mathematical but executable specification of the system early in the design phase. Our experience indicates that the combination of mathematical rigor with execution and analysis tools such as Maude leads to better understanding of the system and often pinpoints potential problems.

To employ this methodology in the exploration of adaptive secure group communication, we build upon abstract executable specifications of all relevant components of Secure Spread. This includes the physical and logical group layers, providing the functionality of Spread [14] with its EVS semantics. The more constrained VS semantics is provided by a specification of Flush Spread [20] on top of this. Independently, a specification of the Cliques toolkitinstantiated to the GDH protocol [8] has been developed. On top of all these components an executable specification of Secure Spread has been built, more precisely the basic algorithm described in [11].

The starting point for our use of formal prototyping techniques in this paper is a formal specification of the Spread GCS. We model its distributed state as a multiset of local state elements (hosts, agents, messages) that behave according to a set of local rules formalizing the evolution of individual elements. Thus, we can visualize the distributed state of the GCS as a *space* in which all state elements float and interact with each other. Due to the complexity and highly nondeterministic nature of the GCS, we first explain how the state elements are axiomatized in rewriting logic, then how each layer (configuration, group, flush, secure) is specified in rewriting logic as an individual component with a public (API) and a private (structure) part. The modular structure of the specification naturally leads to a modular structure of testing, analysis, and mathematical proofs.

A. Modeling in Rewriting Logic

In general, a rewrite theory is a triple $\mathcal{R} = (\Sigma, E, R)$ with (Σ, E) an equational specification with signature of operators Σ and a set of equational axioms E, and a collection of rewrite rules R. The equational specification describes the static structure of the GCS as an algebraic data type and is a purely functional part, while the dynamics is described by the rules in R representing local state transitions that can occur in the system axiomatized by $\mathcal{R}$ and that are applied modulo the equations E. In Maude, an equational specification is made up of declarations of the following kinds

$var\ VarName\ :\ Sort\ .$

$op\ OpName\ :\ Sort_0 ... Sort_k\ \longrightarrow\ Sort\ [OpAtt]\ .$

$eq\ Term_0\ =\ Term_1\ [StAtt]\ .$

$ceq\ Term_0\ =\ Term_1\ if\ Cond_1\ \wedge\ ...\ \wedge\ Cond_k\ .$

A term is a variable (*var*) or the well-formed application of an operator (*op*) to a list of argument terms. In the fragment above, *VarName* is a variable name of type *Sort*, *OpName* is the operator name,$Sort_0 \dots Sort_k$ is the list of sorts for its arguments, *Sort* the sort of its result optionally followed by attribute declarations (*OpAtt*), which allow us to specify structural equations like associativity, commutativity, idempotency and identity. $Term_0$ and $Term_1$ are equivalent only if they belong to the same equivalence class as determined by the equations (*eq*) or conditional equations (*ceq*). In the particular case of conditional equations, $Cond_1\ \wedge\ \dots \wedge\ Cond_k$ represent the set of conditions that must hold. We then give a set R of rewrite rules to specify state transitions as follows.

$\mathbf{rl}\ Term_0\ \Longrightarrow\ Term_1\ .$

$\mathbf{crl}\ Term_0\ \Longrightarrow\ Term_1\ if\ Cond_1\ \wedge\ ...\ \wedge\ Cond_k\ .$

The keywords *rl* and *crl* introduce a rule and a conditional rule respectively. $Term_0$ and $Term_1$ are terms and

$Cond_1 \ \wedge \ \ldots \wedge \ Cond_k$ are rule conditions. Intuitively, rules (conditional and unconditional) describe local, potentially concurrent state transitions. States are represented as terms of the equational theory.

Let us begin to axiomatize the distributed state of the GCS by assuming the multiset structure described above. Therefore, we can view the distributed state as built up by a binary union operator, which we can represent using empty syntax as

```
op _ _ : State State -> State [assoc comm id: eState]
```

Following the conventions of Maude's mix-fix notation, we use underscore symbols (_) to indicate argument positions and the multiset union operator is declared to satisfy the laws of associativity (assoc) and commutativity (comm), and to have identity *empty state* (eState). Thus, complex distributed states are generated from singleton state elements by multiset union.

B. Formal Specification

We tried to keep the formal specification as abstract as possible by omitting several optimizations of the actual implementation while preserving the observable behavior. This allows us to reduce the complexity of states as well as the complexitiy of the state space, compared with the concrete implementation. Our specification is modular in the sense that each layer is specified as a component with a clearly defined API and each component takes the role of an application from the viewpoint of the component below and the role of a service for the component above. We will not discuss the formal details of the specifications of each component in this paper, but the interested reader can find all the components on the web [21]. For sake of brevity we have omitted many details, in particular sorts for sets and lists.

Configuration Layer

The configuration layer specification does not imply the use of a particular synchronization protocol (such as token ring or hop). The choice of a particular protocol could have certain bottom-up effects that upper layers should not rely on. Additionally, we expressed the principle of best effort delivery in the most direct way, namely in each situation we allow the delivery of all possible messages under the given delivery constraints.

Configuration Layer State: A **host** encapsulates its local state (operational in normal conditions, failed if crashed, transitional if the EVS algorithm is being executed) and local configuration (which can be either a regular configuration or a transitional one, which is inserted by the EVS algorithm when a network change occurs), as well as a configuration index and (optionally) the previous configuration information. Note that members of a configuration form a connected component of the network, such that each member can communicate with every other member. Each host also holds the set of received messages (input buffer), the set of delivered messages (to the client), a history of messages delivered since the last crash, the set of acknowledged messages, sequence numbers locally generated, sequence numbers known to each host and the set of messages sent (output buffer). Thus, we model a physical host (*proc*) using the following state elements:

```
op proc : String -> Proc .
op operational : Proc -> State .
op failed : Proc -> State .
op evs-start : Proc ProcSet Bool -> State .
op regconf : ProcSet Nat -> Conf .
op transconf : ProcSet Nat Conf -> Conf .
op localconf : Proc Conf -> State .
op received : Proc MessageSet -> State .
op delivered : Proc MessageList -> State .
op alldelivered : Proc MessageList -> State .
op acked : Proc MessageSet -> State .
op localmsgs : Proc Conf NatSet -> State .
op knownmsgs : Proc Conf NatSet -> State .
op sent : Proc Proc' BroadcastSet -> State .
```

In order to have a consistent distributed state of the GCS, we force each host to be in exactly one of the local states described above and we explicitly keep a list of all existing hosts and a set of global counters used to generate new configuration indices and message sequence numbers. Each configuration uses a pair of sequence numbers to ensure fifo and causality constraints, respectively, and holds the causal and total order delivery constraints to be enforced (the eventlist component of totalorder is simply a trace of all delivery events)

```
op network : ProcSet -> State .
op freshconf : Nat -> State .
op freshseq : Nat -> State .
op causalorder : Conf ConstraintSet -> State .
op totalorder : Conf EventList -> State .
```

Configuration Layer Messages: Each **data message** has a type (data, transitional, configuration, internal acknowledgment), a sequence number and mode (reliable, fifo, causal, agreed, safe), which defines its reliability and delivery constraints.

```
op datamsg : Proc Mode Conf Nat NatSet Data -> Message .
op transmsg : Conf -> Message .
op confmsg : Conf ProcSet -> Message .
op ackmsg : Proc Conf Nat Nat -> Message .
op broadcast : ProcSet Message -> Broadcast .
op reliable : -> Mode .
op fifo : -> Mode .
op causal : -> Mode .
op agreed : -> Mode .
op safe : -> Mode .
```

Since data messages are identified by their sequence number, we use the standard definition of Lamport's causal order, taking all messages into account, and carefully adding the message' sequence number to *localmsgs* and *knownmsgs* of the sender host when a multicast request is handled and to the receiver host's *knownmsgs* when the message is delivered. Then we require that before a message can be delivered, all messages in its past cone must have been delivered, except for those messages sent by a member outside of the current configuration, *i.e.* we allow gaps in the causal order for messages where the sender is not with us anymore. Similarly, we keep the total order of delivered messages for each configuration and allow gaps in the total order for messages where the sender is not longer with us. Due to space limitation we cannot discuss the extended virtual synchrony algorithm, and we have to constrain ourselves to give only a flavor of the formal specification.

Configuration Layer Rules: Below we have selected two key rules. The first rule formalizes the processing of an incoming multicast request (from a higher protocol layer) in the normal operational mode (*i.e.* without disruptions). The effect is that local knowledge and causal order delivery constraints are updated, the message is broadcast on the network, and the processing of the request is acknowledged.

```
rl operational(proc)
   network(everybody)
   localconf(proc,conf)
   localmsgs(proc,conf,localmsgs)
   knownmsgs(proc,conf,knownmsgs)
   freshseq(seq)
   delivered(proc,messagelist)
   alldelivered(proc,messagelist')
   causalorder(conf,constraints)
   sent(proc,broadcastset)
   multicast-req(proc,mode,data)
   =>
   operational(proc)
   network(everybody)
   localconf(proc,conf)
   localmsgs(proc,conf,localmsgs sNatSet(seq))
   knownmsgs(proc,conf,knownmsgs sNatSet(seq))
   freshseq((s seq))
   delivered(proc,messagelist)
   alldelivered(proc,messagelist')
   causalorder(conf,(addConstraints(constraints,proc,
     localmsgs,knownmsgs,seq,mode)))
   sent(proc,broadcastset sBroadcastSet(broadcast(everybody,
     datamsg(proc,mode,conf,seq,knownmsgs,data))))
   multicast-ack(proc) .
```

The delivery of a non-safe message is formalized by the following rule. If a message is received from the network and the delivery constraints are met (see condition), the local knowledge and the total order is updated, and the message is put into the delivery buffer, where if can be accessed by a higher protocol layer. We define *deliverable* as a predicate that allows us to check if a message can be delivered under the given constraints.

```
crl operational(proc)
    localconf(proc,conf)
    delivered(proc,delivered)
    localmsgs(proc,conf,localmsgs)
    knownmsgs(proc,conf,knownmsgs)
    received(proc,(sMessageSet(message) received))
    alldelivered(proc,alldelivered)
    causalorder(conf,constraints)
    totalorder(conf,events)
    =>
    operational(proc)
    localconf(proc,conf)
    localmsgs(proc,conf,localmsgs)
    knownmsgs(proc,conf,knownmsgs knownmsgs(message)
      sNatSet(seq(message)))
    received(proc,received)
    delivered(proc,(delivered sMessageList(message)))
    alldelivered(proc,(alldelivered sMessageList(message)))
    causalorder(conf,constraints)
    totalorder(conf,addEvent(events,src(message),
      seq(message),mode(message)))
    if deliverable(proc,conf,received,alldelivered,
       message,constraints,events) /\
       not(safe(mode(message))) .
```

Group Layer

We assume the use of light-weight groups and a simplified one-to-one mapping of agents to hosts, which avoids certain bottom-up effects that do not represent guaranteed behavior, such as the fact that messages between agents on the same host would never be lost.

Group Layer State: An **agent** holds information regarding its local state, such as the list of the groups it belongs to (there is always an implicit private group it belongs to) and its view, whose value is the associated configuration, group name, group members and a view index. We added the following state elements to model *agents*, *groups*, and *views*:

```
op agent : String -> Agent .
op group : String -> Group .
op view : Conf Group AgentList Nat -> View .
```

Additional state elements and message types were added to keep track of group layer states of hosts (operational, transitional and gather), all connected clients, update group views and process group messages.

```
op goperational : Proc -> State .
op gtrans : Proc ProcSet -> State .
op ggather : Proc ProcSet ProcSet -> State .
op ggather' : Proc ProcSet ProcSet GMessageList -> State .
op gclients : AgentList -> State .
op gjoined : Proc GroupList -> State .
op gview : Proc ViewSet -> State .
op glocalconf : Proc Conf -> State .
op gdelivered : Proc GMessageList -> State .
```

Group Layer Rules: Following the Spread implementation, join and leave events are realized as agreed messages, and weaker messages (e.g. reliable, fifo) do not have to respect them, *i.e.* can be delivered earlier or later depending on the host. The following rule shows how join, leave, or disconnect events, which have been sent by the group layer as agreed messages and are now delivered by the underlying configuration layer back to the group layer, are passed on to the next higher layer under normal conditions.

```
crl gstate(goperational,gtrans,ggather,ggather')
    gjoined(proc,grouplist)
    gview(proc,viewset)
    glocalconf(proc,conf)
    delivered(proc,(sMessageList(message) messagelist))
    gdelivered(proc,gmessagelist')
    =>
    gstate(goperational,gtrans,ggather,ggather')
    gjoined(proc,grouplist')
    gview(proc,viewset')
    glocalconf(proc,conf)
    delivered(proc,messagelist)
    gdelivered(proc,(gmessagelist' gmessagelist''))
    if contains(goperational gtrans,proc) /\
       isdata(message) /\ gmessage := data(message) /\
       (isgjoin(gmessage) or (isgleave(gmessage) /\
       sender(gmessage) =/= proc) or (isgdisconnect(gmessage)
       /\ sender(gmessage) =/= proc)) /\
       grouplist' := update(proc,grouplist,gmessage) /\
       viewset' := update(proc,conf,viewset,gmessage) /\
       gmessagelist'' := mkGMessages(proc,grouplist',
         viewset',gmessage) .
```

It uses a (partial) function *mkGMessages* which translates each such message into a multiset of messages, one for each affected group. Some cases of its equational specification are given below:

```
op mkGMessages : Proc GroupList ViewSet GMessage ->
                 GMessageList .
```

```
eq mkGMessages(proc,eGroupList,viewset',gmessage) =
       eGMessageList .
ceq mkGMessages(proc,sGroupList(group) grouplist',viewset',
       gmessage) = sGMessageList(gjoinmsg(sender,group,
       get(viewset',group))) mkGMessages(proc,grouplist',
       viewset',gmessage)
    if gjoinmsg(sender,group,noview) := gmessage .
ceq mkGMessages(proc,sGroupList(group) grouplist',viewset',
       gmessage) = mkGMessages(proc,grouplist',viewset',
       gmessage)
    if gjoinmsg(sender,group',noview) := gmessage  /\
       group' =/= group .
```

Flush Layer

Our flush layer specification mainly follows [20], but we have omitted several optimizations, such as special treatment for non-vulnerable messages (non-vulnerable messages are messages that can never be delivered too early and hence do not need to be tagged with the sending view), extra *flush-recv* messages after a *flush-ok* and drop of unprocessed membership changes if they become too old. These optimizations obscure the algorithm and are not relevant at the specification level, *i.e.* from an observational point of view.

Flush Layer State: Additional state elements were added to keep track of agent's installed and pending views (pending views are views that the agent has not yet installed), as well as how pending configuration messages are handled.

```
op fiview : Agent ViewSet -> State .
op fpview : Agent ViewSet -> State .
op fpending : Agent GroupOpSet -> State .
op fstate : Agent GroupSet GroupSet GroupSet -> State .
```

Flush Layer Rules: The following rule formalizes the situation where a join operation is pending for a group and an application requests the next message using a flush layer receive request. If the condition, which requires that *flusk-ok* messages have been received from all current members (as a reply to an earlier *flush-req* generated by a different rule), is satisfied, the join message is passed on to the application (as a reply to its receive request).

```
crl fstate(client,steady,authorize,agree)
    fpending(client,pending)
    fpview(client,viewset)
    fiview(client,viewset')
    f-receive-req(client)
    fbuffer(client,gmessagelist)
    =>
    fstate(client,add(steady,group),authorize,
      rm(agree,group))
    fpending(client, rm(pending,group))
    fpview(client,viewset''')
    fiview(client,viewset'')
    fbuffer(client,removeallfoks(client,group,members(view),
      gmessagelist))
    f-receive-ack(client,fjoinmsg(sender(get(pending,
      group)),group,view))
    if sGroupSet(group) groupset := receivedallfoks(client,
         agree,viewset,gmessagelist) /\
       joining(pending,group) /\ view := get(viewset,group)
       /\ viewset'' := update(viewset',group,view) /\
       viewset''' := rm(viewset,group) .
```

Secure Layer

Our secure layer specification uses the basic algorithm presented as a finite state machine in [11] and includes the GDH2 specification obtained by a reverse engineering and abstraction process from the Cliques toolkit source code.

Secure Layer State: In order to support secure communication, we equipped each agent with a secure group context information (session random number, partial group shared key, group members, keyid), which is modeled using the following state elements

```
op context : KeyShare PartialKey GroupMemberList -> Context .
op groupcontext : Group Context -> GroupContext .
op scontext : Agent GroupContextSet -> State .
```

Secure Layer Rules: To begin with one of the simpler rules, we show below how a secure multicast request is formalized. The secure multicast request for a given group encrypts the application data with the corresponding group key and triggers a flush layer multicast request. The condition expresses that this rule can only apply if the group is in a secure state, *i.e.* a key has been established for the current view.

```
crl sstate(client,secure,cm,pt,ft,fo,kl)
    scontext(client,groupcontextset)
    ssp-multicast-req(client,mode,group,sdata)
    =>
    sstate(client,secure,cm,pt,ft,fo,kl)
    scontext(client,groupcontextset)
    ssp-multicast-req'(client,mode,group,sdata)
    f-multicast-req(client,mode,group,fdata)
    if contains(secure,group) /\
       key := groupsecret(get(groupcontextset,group)) /\
       fdata := enc(key,sdata) .
```

Following [11], a membership change triggers the key establishment, which deterministically chooses an initial member of the new view (forming a singleton clique) and merges all remaining members into the clique using the merge subprotocol. Below we show the first two rules of this process. The first rule formalizes the creation of a singleton clique; while the second rule formalizes the the creation of a new cliques' user, which will be eventually merged with the singleton clique (rules not given here).

```
crl ssp-cm-cases(client,group,fmessage)
    snewcontroller(client,newcontroller)
    =>
    ssp-cm-not-alone-wait-for-first-user(
      client,group,fmessage,mergingagents)
    snewcontroller(client,newcontroller')
    clq-first-user-req(client,group)
    if card(agentset(members(view(fmessage)))) > 1 /\
       client==first(members(view(fmessage))) /\
       mergingagents:=rm(members(view(fmessage)),client)
       /\ newcontroller':= rm(newcontroller, group) .


crl ssp-cm-cases(client,group,fmessage)
    snewcontroller(client,newcontroller)
    =>
    ssp-cm-not-alone-wait-for-new-user(client,
      group,fmessage)
    snewcontroller(client,newcontroller')
    clq-new-user-req(client,group)
    if card(agentset(members(view(fmessage)))) > 1 /\
       client=/=first(members(view(fmessage))) /\
       newcontroller':=if client==last(members(
         view(fmessage))) then add(newcontroller,group)
         else rm(newcontroller,group) fi .
```

In the case of cascaded membership changes, *i.e.* a membership change occurs while the key agreement is still in progress, the algorithm is restarted. Once the key agreement is completed, the new group key is used for future communication until a new key has been established.

IV. High-Level Adaptability

As we briefly explained in Section I, the application should be able to tailor the secure GCS according to its needs in terms of synchrony and security. In order to provide this level of adaptability, we need to identify what assumptions need to be relaxed, what are the tradeoffs between these different levels and what parameters can be adjusted to tune the performance.

A. Adaptable Synchrony

Secure Spread implements security on top of Flush Spread, a layer providing the VS semantics, which guarantees that messages are sent and delivered in the same view. This synchronization makes it easier to implement the key establishment protocol because every message is encrypted with the same key as the receiver believes is current when the message is delivered. In order to provide security on top of EVS semantics, the secure GCS can no longer assume that the received message was encrypted with the current key. The paper [1] proposes a solution to this problem based on two levels of keys used by the heavy-weight and the light-weight layer, respectively. In the present paper we use the idea of [1] to maintain a history of keys indexed by key identifiers (keyids), but we stick to the use of light-weight group keys without assuming underlying heavy-weight keys. This enables us to study the interaction between security and EVS semantics in its pure form and makes the solution independent of the implementation of Spread. Furthermore, given that we already have a specification of Secure Spread, it makes it easy to obtain an integrated solution which can be adapted to both, the original VS-based security, exactly as implemented in Secure Spread, and to the new EVS-based security.

Hence, we have modified the formal prototype of Secure Spread as follows: First, for EVS groups (we added VS and EVS group synchrony modes as adaptation parameters) we removed the synchronization constraints imposed by the Flush Spread layer. Second, every key generated is associated with a keyid, every message is tagged with the corresponding keyid of the key used to encrypt the message, and every member of the group keeps a list of (possibly old) keys and their associated keyids. Thus, every time a message is received its keyid is checked and the corresponding key is fetched from the list so it can be properly decrypted. Thus members can move from one view to another one and rekey asynchronously. Every rekey phase adds the current key to the list of older keys and the newly generated key is used as the current key.

Obviously, the dynamics of this approach is far less constrained than in the VS case. Specifically, we observed the following difficulties: Although keyids allow to decrypt messages sent in previous views, they do not guarantee that every message received can be decrypted and delivered to the application. In particular, it may be possible that a new member receives an old message sent in a previous view. If he joined the group very recently, he does not have the key required to decrypt [1]. One possibility would be to drop the message, but this would violate the EVS semantics (only a network change can justify dropping a message). We have addressed this issue by introducing the concept of a *nondecryptable* message, *i.e.* a message with content that is not accessible, to inform the application of this situation. However, there is also the possibility that the new member can find a key in his list associated with the keyid of the message, but it is not the keyid associated with the new view. In this case, we say that the message was encrypted under an *old keyid*, and we tag the message as *delayed* to inform the application of this situation. Security on top of EVS allows us to increase concurrency and hence performance by providing non-blocking (application level) communication that uses the most recently established key to send messages, while the key establishment for the new view is in progress. However, this new added flexibility relaxes the degree of consistency in the system and eliminates some security guarantees.

B. Adaptable Security

The choice of the key establishment protocol is a natural dimension of adaptability in secure group communication. However, even with the most efficient key establishment protocols, network connectivity changes and membership changes can cascade while the key establishment is in progress, causing a restart of the key establishment protocol from scratch. Thus, delaying the execution of the key establishment protocol and carefully avoiding its execution in certain situations can improve system performance while preserving forward and backward secrecy. We have explored two approaches to reduce the number of key establishment phases. The first approach is based on key caching and the second one is based on lazy key establishment, that is delaying key establishment until the key is really needed. Both approaches are *generic*, that is independent of the underlying protocol, and can be composed to further improve system performance without sacrificing security guarantees. As an important by-product, key caching allows us to deal efficiently with temporary disconnections (as opposed to voluntary join/leave events), which are quite common in groups with mobile participants and their consequences are similar to network connectivity changes. Interestingly, the decision to (partially) relax virtual synchrony has opened a variety of new possibilities, which includes not only the possibility to perform lazy key establishment but also new secure delivery modes.

1) Key Establishment Protocols: One of the most important security guarantees is data confidentiality, which protects data from being eavesdropped. The way the secret shared group key is computed, how often, and when it is computed are critical for the security of the GCS. There are two basic approaches to generate a secret shared key in GCS. In the centralized approach, one member (typically a group leader) chooses the group key and distributes it to all group members (*group key distribution*); while in the contributory approach

[1]The solution presented in [1] also has this problem.

every member contributes to the creation of the secret shared key (*group key agreement*). Although the centralized approach works reasonably well for static (possibly large) groups, it turns out that the contributory approach is more robust for non-hierarchical (mid-size) groups with dynamically changing memberships [7]. The relevant properties for key establishment algorithms are of purely computational nature [22]: *Cryptographic forward secrecy* guarantees that a passive adversary who knows a contiguous subset of old group keys cannot discover subsequent group keys. *Cryptographic backward secrecy* guarantees that a passive adversary who knows a contiguous subset of group keys cannot discover preceding group keys. In a GCS like Secure Spread that supports the VS semantics, tightly synchronizing view changes with key establishment phases, backward and forward secrecy are immediate consequences of cryptographic forward and cryptographic backward secrecy, respectively [11]: *Forward secrecy* guarantees that nobody should be able to read messages sent to a group after he left this group (assuming he will not become a member of the group in the future). *Backward secrecy* guarantees that nobody should be able to read messages sent to a group before he joined this group (assuming he was not a member of the group in the past). However, to be precise, we need to define what are the join/leave events referenced in these definitions. It obviously would not make sense to take them to be the events of requesting a join/leave at the GCS. These events would be of no use for the client applications. They are not (immediately) observable for the applications, because the processing of such requests can be delayed. This suggests to define leave/join events to be the events where the GCS delivers leave/join (with the new view) to the application which sends the message. Similarly, we have to be precise about what the send event in these definitions refers to. Since a message carries sensitive data, we should adopt the most conservative definition, namely the event when the application requests the GCS to send a message.

Forward secrecy under the EVS semantics is fairly straightforward: Assume a member A leaves the group G, the GCS delivers a new view to B, and B sends a message M to G. The new view can have only been delivered after successful completion of a key establishment phase between the members of the new view. Since M is encrypted with the resulting key that A does not know, forward secrecy is guaranteed.

Backward secrecy under the EVS sematics, however, does not hold, as the following counterexample shows: Assume A requests the GCS to send a message M to a group G, but the processing of this request is delayed. In the meantime B joins G, and the GCS delivers the new view $\{A, B\}$ to A. Now the GCS processes the send request in the new view, which means that the message is encrypted using using the key associated with this view. Hence, B can decrypt the message, which is a violation of backward secrecy.

To solve this problem we have adopted the following solution: We add the view in which we would like to send the message (*requested sending view*) as an argument to the multicast service. This view determines the key to be used for encryption. Even if the message is sent out in the new view, the key of the requested sending view should be used. Note that there are two possibilities for a member of the new view. If it was a member of the earlier sending view it can decrypt the message. If it was not a member of the earlier sending view it just joined the group and will not be able to decrypt in accordance with backward secrecy. In this case, the message is delivered but as *nondecryptable*. The possibility to specify a requested sending view is optional, so that if backward secrecy is not a concern the original implementation can be used.

The high-level rationale for this solution is the following: The EVS semantics leads to a loss of sending view awareness at the application, but the benefits of sending view awareness can be recovered by always sending messages with a *requested sending view*, which prevents members joining unexpectedly to decrypt messages not intended for them. The drawback is that we have to internally keep track of former keys, and some messages received will be *nondecryptable*. Both of these mechanisms, however, were already added when we moved from the VS to the EVS semantics (see Section IV-A) so that this extension does not cause any additional overhead.

2) Key Caching: Frequent network connectivity changes may trigger patterns of membership changes, where new views tend to have the same members as earlier views. Current implementations of secure GCS generate a new key for each view. Thus, if a subset of members of a group becomes temporary isolated due to a network partition, the key establishment protocol will be invoked for each new partition, and again when the partitions merge together. No member has left/joined the group, but several new keys have been generated. Obviously, this is unnecessary, because the group membership has not changed in the end. Ideally, the key establishment protocol should be executed only if the current set of members has not shared a secret key before; otherwise, a previously agreed upon key can be used instead. Since the reuse of keys increases the vulnerability to crypto-analysis attacks, key caching like all forms of key reuse need to be carefully constrained. To this end, keys can be equipped with an expiration or some other attribute limiting key reuse, and they are removed from the list when this limit is reached.

In detail we have made the following modifications to our formal prototype to accommodate for key caching:

1) Every member keeps a list of keys and the associated set of members that share that key. The list is updated whenever a new key is generated.
2) If a membership change or network connectivity change happens, every member receives a message with the updated membership.
3) Every member checks its list of keys and if the updated membership shared a key before, the key is retrieved and used as the current key; otherwise the key establishment is triggered and a new key is generated.

Forward and backward secrecy are still satisfied, but *key freshness*, *i.e.* the property that each view uses a fresh key to encrypt messages, is given up. Therefore, a new group security mode (*fresh secure*) is added to enforce freshness if the application requests this level of security. If the group security mode is fresh secure, a normal key establishment is triggered even if the members shared a secret key before. It is important to point out that a keyid associated with a nonfresh key should not be confused with an old keyid, *i.e.* a keyid associated with a previous view, and hence it does not imply that the message is delivered as delayed (see Section IV-B.4).

3) Lazy Key Establishment: Current GCS have been designed under the assumption that network connectivity changes occur rarely and that members exchange a considerable amount of messages between membership changes. However, membership changes (due to unpredictable network connectivity changes or join/leave operations) may occur quite frequently in certain environments (wireless, mobile), and with many view changes taking place it is highly unlikely that messages are sent in every intermediate view. Under these circumstances, delaying the execution of the key establishment protocol until a message needs to be sent will avoid unnecessary key establishment phases. We say that a key establishment phase is unnecessary if a key is generated but not used because no message is sent before a new key is generated.

As a possible solution we explored *delayed key establishment*. Instead of a synchronized initiation of the key establishment algorithm by a view change event, the member who wants to send a message triggers the key establishment asymmetrically. Our formal prototype is modified as follows:

1) Any membership change or network connectivity change is treated normally and the membership is updated, but the key establishment protocol is not executed.
2) When a member needs to send a message, it checks if a current key exists and if it is up to date, *i.e.* belongs to the most recently established view.
3) If the key is up to date, then the message is encrypted and sent normally.
4) If the key does not exist or is not up to date:
    a) The member starts the key establishment protocol, notifies the other group members and stalls the message till the new key is generated.
    b) Members are notified and each one of them starts the key establishment protocol, which proceeds normally.
    c) If another member wants to send a message, the key establishment has been triggered by some other member and no view change has been triggered, the message is stalled until the new key is generated and the member continues with the normal key establishment execution (*i.e.* the key algorithm is not restarted).
    d) If a view change event is triggered at any time, the membership is updated and the key establishment protocol is restarted.
    e) Once the key has been generated, the current key is updated, the up-to-date flag is set and members proceed to encrypt and send messages normally.

4) Secure Delivery Modes: Traditionally, secure delivery in GCS has been restricted to the delivery of an encrypted message, assuming that all members of the group are able to decrypt the message using the unique shared group key. When we relax the virtual synchrony semantics, messages encrypted with different group keys may be received at any time and we can no longer assume that the receiver is able to decrypt every message using the most recent key or even to decrypt the message. As a result, EVS semantics leads to a new variety of secure delivery modes based on key freshness and an extended concept of *safe messages* as follows:

- Non-secure: Message is sent and received in clear-text.
- Secure: Message is encrypted and can be decrypted with any (possibly old) known key; otherwise delivered as *nondecryptable*.
- Strongly secure: Message is encrypted and must be decrypted with the most recent known key; otherwise delivered as *nondecryptable*.
- Safe-secure: Message is encrypted and can be decrypted with any (possibly old) known key, but can only be delivered if everybody else received and decrypted the message using any (possibly old) known key.
- Strongly safe-secure: Message is encrypted and must be decrypted with the most recent known key, but can only be delivered if everybody else received and decrypted the message using the most recent known key.

V. Formal Prototyping

In this section we show how we generalize the formal specification of Secure Spread to support the high level adaptability discussed in Section IV.

A. Relaxing Synchrony

Since the configuration and group layers already provide EVS semantics, our first thought was to remove the flush layer and let the group and secure layers communicate to each other. However, the key establishment protocol requires a synchronized initialization (*i.e.* all members must be aware that a new key is going to be generated) that then would have to be added to the secure layer to ensure proper execution, making impossible to provide different degrees of synchrony using the same specification. Therefore, we modified the flush layer to incorporate group types (a group type identifies the semantics of the group) and remove synchronization constraints accordingly; *i.e.* groups using VS-semantics use the full-fledged flush layer; while groups using EVS-semantics update flush layer state, but avoid the expensive and time consuming *flush acknowledgment* as well as *data blocking*. This allows us to model and support both VS groups and EVS groups at the same time, and explore their possible coexistence. In particular, we added the following state elements to the flush layer to keep track of the semantics of each group and modified the rules accordingly.

```
op group-vs :  -> GroupType .
op group-evs :  -> GroupType .
op fgrptype : Agent GroupSet GroupSet -> State .
```

As an example, we show the modified version of the joining operation rule presented in Section III-B for the specific case of EVS groups. That is, a join message for an EVS group updates the flush state, but does not require *flush acknowledgments* and the message is immediately passed on to the application.

```
crl fstate(client,steady,authorize,agree)
    fgrptype(client,vsgrp,evsgrp)
    fpending(client,pending)
    fpview(client,viewset)
    fiview(client,viewset')
    f-receive-req(client)
    =>
    fstate(client,add(steady,group),authorize,
      rm(agree,group))
    fgrptype(client,vsgrp,evsgrp)
    fpending(client,rm(pending,group))
    fpview(client,viewset''')
    fiview(client,viewset'')
    f-receive-ack(client,fjoinmsg(sender(get(pending,
      group)),group,view))
    if sGroupSet(group) groupset := agree /\
       joining(pending,group) /\ view := get(viewset,group)
       /\ viewset'' := update(viewset',group,view) /\
       viewset''' := rm(viewset,group) /\
       contains(evsgrp,group) .
```

Similarly, at the secure layer, we added to the **agent** declaration the semantics of the group (group type), the associated keyid-key list and keyid-membership set information, the level of laziness of the key establishment protocol (eager, key caching, lazy till a message needs to be sent, or a combination of both) and extended the secure group context information by including the keyid associated to the current key

```
op keyid : Nat -> KeyId .
op scontext : Agent KeyId GroupContextSet -> State .
op sgrptype : Agent GroupSet GroupSet -> State .
op slaziness : Agent GroupSet GroupSet GroupSet -> State .
op sgroupkeylist : Agent GroupAssocKeyidListSet -> State .
op sgroupmembersetlist : Agent GroupAssocMembSetListSet ->
                         State .
```

Furthermore, we enhanced the multicast request with a *req-sending-view* parameter that allows us to request a specific view in which the message should be sent, a *freshkey* parameter that indicates if a new key was generated for the current view and the capability to tag every outgoing message with the keyid corresponding to the key used to encrypt the message. The following rule shows the modified version of the multicast request rule described is Section III-B. Note that the restriction to perform a multicast only in a secure state is not present anymore, allowing us to encrypt and send messages using the most recently established key while a rekey is in progress.

```
crl sstate(client,secure,cm,pt,ft,fo,kl)
    scontext(client,currkeyid,groupcontextset)
    sgroupkeylist(client,groupassockeylistset)
    sgroupmembersetlist(client,groupassocmembsetlistset)
    slaziness(client,lazyst,lazyct,eager)
    siview(client,viewset)
    ssp-multicast-req(client,mode,group,sdata,
      req-sending-view)
    =>
    sstate(client,secure,cm,pt,ft,fo,kl)
    scontext(client,currkeyid,groupcontextset)
    sgroupkeylist(client,groupassockeylistset)
    sgroupmembersetlist(client,groupassocmembsetlistset)
    slaziness(client,lazyst,lazyct,eager)
    siview(client,viewset)
    ssp-multicast-req'(client,mode,group,sdata)
    f-multicast-req(client,mode,group,fdata)
    if viewkeyid := keyid(lookup((
         get(groupassocmembsetlistset,group)),
         agentset(members(get(viewset,group))))) /\
       key := partialkey(lookup((get(groupassockeylistset,
         group)),viewkeyid))  /\
       fdata := enc(viewkeyid,key,sdata) .
```

Preserving EVS Semantics

Although the idea of providing security on top of EVS was mentioned in [1], no actual system implementation exists and subtle effects that may be observed naturally when the system is running were easily overlooked. The following example illustrates a simple scenario overlooked in the literature, but discovered in our formal prototype via symbolic execution.

Let us assume that A and C belong to the same secure group G, which has a current shared group key k. Let C send an agreed message $m1$ encrypted with k. Just after sending $m1$, B joins G and triggers the key establishment protocol. Let us assume that $m1$ is delayed so that it is received in the new view containing $\{A, B, C\}$, *i.e.* after the key establishment is completed. A and C deliver this message because they both have access to the key k (they use the keyid in $m1$ to locate the key in their key history). B receives $m1$ but can not decrypt and deliver $m1$ to the application, because B does not know the key k used to encrypt $m1$. Finally, let's assume that B sends an agreed message $m2$ with the new key k' and A and C deliver this message. But what about B ?

The messages $m1$ and $m2$ were sent in agreed mode and the EVS semantics guarantees that agreed messages are delivered in the same order. In our example, $m1$ must be delivered before $m2$ because this is the order chosen by A and C. Furthermore, due to the self-delivery property $m2$ needs to be delivered by B at some point, but only after $m1$ according to the constraint just mentioned. Thus, B can not drop $m1$ without creating a gap in the ordering and thus violating EVS semantics. On the other hand, B cannot deliver the message in the usual way, because it cannot decrypt it.

In order to preserve the EVS semantics we introduced the concept of *nondecryptable message* and extended the receive rules to tag the message nondecryptable if the keyid used to encrypt is not found in the history of keys for that particular member (see condition).

```
crl siview(client,viewset)
  sstate(client,secure,cm,pt,ft,fo,kl)
  scontext(client,currkeyid,groupcontextset)
  ssp-receive-req'(client)
  sbuffer(client,smessagelist)
  f-receive-ack(client,fmessage)
  sgroupkeylist(client,groupassockeylistset)
  =>
  siview(client,viewset)
  sstate(client,secure,cm,pt,ft,fo,kl)
  scontext(client,currkeyid,groupcontextset)
  sbuffer(client,smessagelist sSMessageList(smessage))
  ssp-receive-req(client)
  sgroupkeylist(client,groupassockeylistset)
  if isidata(fmessage) /\  group := group(fmessage) /\
     keyid' := keyid(data(fmessage)) /\
     not(contains((get(groupassockeylistset,
       group)),keyid')) /\
     smessage := sdatamsg(client,group,
      nondecryptable(data(fmessage)),get(viewset,group)) .
```

B. Applying Generic Optimizations

In order to support key-caching and lazy key establishment, we modified the *key-list* and *cascading-membership* states of the finite state machine. First, we allow the *key-list* state to update the associated keyid-key list (*groupassockeylistset*) and its corresponding membership information (*groupassocmembsetlistset*) every time a new key is successfully generated. Then, we modified the behavior of the *cascading-membership* state (which in spite of its name is also used for a single non-cascaded view change) to avoid rekey in case we specified lazy key establishment or to cache a previous agreed key, if key caching has been specified and a previous agreed key exists; otherwise, the full key establishment protocol will be executed.

In VS semantics, a membership change notification is delivered to the application along with the new secure view, allowing to delay and discard membership notifications while the key establishment is being executed and still preserve ordering constraints since no messages are sent or received. However, in EVS semantics, any membership change notification must be delivered to the application as soon as it is received in order to preserve ordering constraints, since we allow messages to be sent and received while the key establishment is in progress. Thus, the *key-list* state will deliver a membership change to the application if and only if the group uses VS semantics and let the *cascading-membership* state to deliver a membership change to the application if the group uses EVS semantics. Below we have selected the most representative rules to illustrate these modifications. The first rule formalizes a membership change operation when *key caching* is selected and the updated membership already shared a key in the near past (see condition).

```
crl ssp-cm-check-members(client,group,fmessage)
    sstate(client,secure,cm,pt,ft,fo,kl)
    scontext(client,currkeyid,groupcontextset)
    snewmembmsg(client,newmembmsgset)
    svsset(client,vsset)
    snotfirstcm(client,notfirstcm)
    sbuffer(client,smessagelist)
    sgroupkeylist(client,groupassockeylistset)
    sgroupmembsetlist(client,groupassocmembsetlistset)
    sfreshness(client,freshkey)
    skainprogress(client,kainprogress)
    =>
    sstate(client,secure',cm',pt,ft,fo,kl)
    scontext(client,currkeyid',groupcontextset)
    snewmembmsg(client,newmembmsgset)
    svsset(client,vsset)
    snotfirstcm(client,notfirstcm')
    sbuffer(client,smessagelist sSMessageList(smessage))
    sgroupkeylist(client,groupassockeylistset')
    sgroupmembsetlist(client,groupassocmembsetlistset')
    sfreshness(client,add'(freshkey,group))
    skainprogress(client,rm(kainprogress,group))
    ssp-receive-req(client)
    if contains((get(groupassocmembsetlistset,group)),
         agentset(members(view(fmessage)))) /\
       card(agentset(members(view(fmessage)))) > 1  /\
       keyid':=keyid(lookup((get(groupassocmembsetlistset,
         group)),agentset(members(view(fmessage))))) /\
       key:=partialkey(lookup((get(groupassockeylistset,
         group)),keyid')) /\
       assockeylist:=get(groupassockeylistset,group) /\
       assockeylist':=add'(assockeylist,(assockey(key,
         currkeyid'))) /\ currkeyid' := keyid' /\
       groupassockeylistset':=update(groupassockeylistset,
         group,assockeylist') /\
       assocmembsetlist:=get(groupassocmembsetlistset,group)
       /\ assocmembsetlist' := add'(assocmembsetlist,
         (assocmembset(agentset(members(view(fmessage))),
         currkeyid'))) /\ groupassocmembsetlistset':=update(
         groupassocmembsetlistset,group,assocmembsetlist')
       /\ smessage := update-transset(get(newmembmsgset,
            group),get(vsset,group)) /\
       notfirstcm' := rm(notfirstcm,group) /\
       cm':=rm(cm,group) /\ secure':=add(secure,group) .
```

If the updated membership is a reoccuring one, its membership information is used to get (via the associated keyid) the previously agreed key. The cached key is then used as the current key, a new associated keyid is generated and both associated keyid-key and membership information lists are updated. Finally, the membership change notification is delivered to the application and the finite state machine moves from the *cascading-membership* state directly to the *secure* state without triggering the key establishment protocol.

Similarly, the following rule formalizes a membership change operation when the *lazy key establishment* is selected, avoiding the creation of a new key and the execution of the key establishment protocol. As we can see in the condition, the membership is updated, the membership change notification is delivered to the application and the finite state machine moves from the *cascading-membership* state directly to the *secure* state without triggering the key establishment protocol.

```
crl ssp-cm-avoid-rekey(client,group,fmessage)
    sstate(client,secure,cm,pt,ft,fo,kl)
    scontext(client,currkeyid,groupcontextset)
    snewmembmsg(client,newmembmsgset)
    svsset(client,vsset)
    snotfirstcm(client,notfirstcm)
    sshadowmessage(client,shadowmessage)
    sbuffer(client,smessagelist)
    =>
    sstate(client,secure',cm',pt,ft,fo,kl)
    scontext(client,currkeyid,groupcontextset)
    snewmembmsg(client,newmembmsgset)
    svsset(client,vsset)
    snotfirstcm(client,notfirstcm')
    sshadowmessage(client,add'(shadowmessage,fmessage))
    sbuffer(client,smessagelist sSMessageList(smessage))
    ssp-receive-req(client)
    if card(agentset(members(view(fmessage)))) > 1 /\
       smessage := update-transset(get(newmembmsgset,
         group),get(vsset,group)) /\
       notfirstcm' := rm(notfirstcm,group) /\
       cm' := rm(cm,group) /\
       secure' := add(secure,group) .
```

C. Key Properties

Since security on top of VS has been studied in [11], we constrain ourselves to properties of EVS groups (recall that our formal prototype integrates both approaches).

Property 1 All messages received at the secure layer are delivered to the application.

Proof Sketch All data messages received at the secure layer are encrypted and tagged with a keyid that identifies the key used to encrypt the message. For each message, if the key associated to its keyid is found in the associated keyid-key list, the message is decrypted and delivered to the application; otherwise the message is tagged nondecryptable and delivered to the application. All membership and configuration change messages received are delivered to the application.

Property 2 All message ordering constraints are preserved.

Proof Sketch All received messages at the secure layer are delivered to the application without delay and in the same order they were received, regardless of the current state of the finite state machine.

Regarding security on top of EVS, we prove that forward and backward secrecy are preserved and that generic optimizations can regain key freshness if fresh secure mode is used.

Property 3 Security on top of EVS semantics provides forward secrecy.

Proof Sketch Since we defined a leave event as an event where the GCS delivers the new secure view to the application, a message send in this new view will be encrypted with a new key, which may be recently generated or a previously agreed key. In either case, the key is only known by the remaining members of the group.

Property 4 Security on top of EVS semantics provides backward secrecy if messages are always sent using the current view as the requested sending view.

Proof Sketch Messages encrypted using the requested sending view parameter will be encrypted with the key that only the members of the specified (possibly past) view have and therefore, only they can decrypt the message. Obviously, the members of the specified view must have joined before the message is sent.

Property 5 When fresh secure mode is used to send a message, key freshness is guaranteed.

Proof Sketch In fresh secure mode, a message is always encrypted with a recently generated key for the current view. If no new key was generated for the current view due to key caching or lazy key establishment, the key establishment protocol is triggered and the message is stalled until a new key is generated.

Although in our formal specification a message already encrypted may sit in the GCS for an unbounded amount of time, in practice, this time is bounded and relatively short compared to the key expiration time. Hence, we have the following property in such an implementation.

Property 6 A delayed message can never be tagged as non-decryptable, assuming that a maximal transit time (max. time between multicast and delivery) exists and is smaller than the key expiration time.

Proof Sketch Under the given assumption it is not possible to receive an old message encrypted with a key that has been removed from the keyid-key association list.

D. Symbolic Execution

Usually, abstract specifications are axiomatic and not executable, but the distinguishing feature of rewriting logic from many other specification languages is that it allows us to use axiomatic specification techniques at a reasonably high-level while still maintaining executability, allowing us to apply symbolic execution to: (i) validate our specification against our understanding of the system, and (ii) find violations of key properties of the formal model and hence of the implementation. In order to deal with the complexity and high degree of concurrency and nondeterminism of a typical GCS, we partially constrain the behavior of the system by composing it with an environment that acts as a controller. By defining a controller language based on sequential and parallel composition of actions and associating each action to a rule in our specification, we steer the system into critical states that confirm or validate the properties of interest.

We present two running examples. The first one represents the nondecryptable example given in Section V-A and the second one illustrates the execution of the lazy key establishment protocol. Below, we show a code excerpt where we define an initial state containing three hosts (*a*,*b*,*c*). This initial state is shared by the two examples.

```
eq allprocs = sProcSet(agent("a")) sProcSet(agent("b"))
              sProcSet(agent("c")) .
op a : Nat -> State .
op b : Nat -> State .
op c : Nat -> State .
op init : -> State .
eq init =
    network(allprocs)
    mkinitialconf(allprocs)
    mkinitialprocs(allprocs)
    mkinitialagents(allprocs)
    fresh(0)       --- for cliques
    freshkeyid(0)  --- for keyId
    ssp-receive-req(agent("a"))
    ssp-receive-req(agent("b"))
    ssp-receive-req(agent("c"))
    sdelivered(agent("a"),eSMessageList)
    sdelivered(agent("b"),eSMessageList)
    sdelivered(agent("c"),eSMessageList)
    sdelivered'(agent("a"),eSMessageList)
    sdelivered'(agent("b"),eSMessageList)
    sdelivered'(agent("c"),eSMessageList)
```

Preserving EVS Semantics: Following the scenario described in Section V-A, the agent located at host *c* sends a message in *agreed mode* to the group *G*. Before the message is received, an agent located at host *b* joins the group and triggers the key establishment protocol. After the successful completion of the key establishment protocol the message is received by the members of the group and a new message also in *agreed mode* is sent by the agent at host *c*, and immediately received by all.

```
PERFORM(MULTICAST-TEST) ;
PERFORM(MULTICAST(agent("c"),group("G"))) ;
PERFORM(SEND(agent("c"),2)) ;  --- message sent
PERFORM(JOIN(agent("b"),group("G"))) ;
PERFORM(SEND(agent("b"),3)) ;
PERFORM(RECEIVE(agent("a"),3)) ;
PERFORM(RECEIVE(agent("b"),3)) ;
PERFORM(RECEIVE(agent("c"),3)) ;
...
PERFORM(DELIVERCHANGE(agent("a"))) ;
PERFORM(DELIVERCHANGE(agent("b"))) ;
PERFORM(DELIVERCHANGE(agent("c"))) ;
PERFORM(RECEIVE(agent("a"),2)) ; --- message received
PERFORM(RECEIVE(agent("b"),2)) ;
PERFORM(RECEIVE(agent("c"),2)) ;
PERFORM(MULTICAST-TEST) ;
PERFORM(MULTICAST(agent("c"),group("G"))) ;
PERFORM(SEND(agent("c"),10)) ;
PERFORM(RECEIVE(agent("a"),10)) ;
PERFORM(RECEIVE(agent("b"),10)) ;
PERFORM(RECEIVE(agent("c"),10)) ;
```

Without the nondecryptable tag, the agent at host *b* can not decrypt and deliver the received message to the application, since it was encrypted and sent before he joined the group. Furthermore, dropping or queuing the message create a gap in the ordering that will eventually stall the system when another agreed message is received, as we can see in the following excerpt of the system, where the message *m2* can not be delivered even if it has been received by all clients. Let us recall that *sdelivered'* is the history of messages delivered to the application. Configuration and transitional messages have been removed from *sdelivered'*) for better readability.

```
network(sProcSet(agent("a")) sProcSet(agent("b"))
        sProcSet(agent("c")))
freshconf(1)
...
freshseq(10)
...
fresh(0)
freshkeyid(2)
ssp-receive-req'(agent("a"))
ssp-receive-req'(agent("b"))
ssp-receive-req'(agent("c"))
...
localconf(agent("a"),regconf(sProcSet(agent("a"))
          sProcSet(agent("b")) sProcSet(agent("c")),7))
localconf(agent("b"),regconf(sProcSet(agent("a"))
          sProcSet(agent("b")) sProcSet(agent("c")),7))
localconf(agent("c"),regconf(sProcSet(agent("a"))
          sProcSet(agent("b")) sProcSet(agent("c")),7))
...
sdelivered'(agent("a"),...
  sSMessageList(sdatamsg(agent("a"),group("G"),
  sdata("m1"),view(regconf(sProcSet(agent("a"))
  sProcSet(agent("b")) sProcSet(agent("c")),7),group("G"),
  sAgentList(agent("c")) sAgentList(agent("a")),0)))

sdelivered'(agent("b"),...

sdelivered'(agent("c"),...
  sSMessageList(sdatamsg(agent("c"),group("G"),
  sdata("m1"), view(regconf(sProcSet(agent("a"))
  sProcSet(agent("b")) sProcSet(agent("c")),7),group("G"),
  sAgentList(agent("c")) sAgentList(agent("a")),0)))
...
```

Using the nondecryptable tag, the example runs to completion and the final state of the system (see below) can be summarized as follows: the agent located at host *a* left the group, but is still alive, the agent located at host *b* disconnected and the agent located at host *c* remains as a singleton group with the state machine in secure state.

```
result State: network(sProcSet(agent("a"))
              sProcSet(agent("b")) sProcSet(agent("c")))
freshconf(8)
...
freshseq(40)
...
fresh(8)
freshkeyid(7)
ssp-receive-req'(agent("a"))
ssp-receive-req'(agent("c"))
CONTROLLER(eController)
localconf(agent("a"),regconf(sProcSet(agent("a"))
          sProcSet(agent("b")) sProcSet(agent("c")),7))
localconf(agent("b"),regconf(sProcSet(agent("a"))
          sProcSet(agent("b")) sProcSet(agent("c")),7))
localconf(agent("c"),regconf(sProcSet(agent("a"))
          sProcSet(agent("b")) sProcSet(agent("c")),7))
...
sdelivered'(agent("a"),...
  sSMessageList(sdatamsg(agent("a"),group("G"),
  sdata("m1"),view(regconf(sProcSet(agent("a"))
  sProcSet(agent("b")) sProcSet(agent("c")),7),group("G"),
  sAgentList(agent("c")) sAgentList(agent("a")),0)))
  ...
  sSMessageList(sdatamsg(agent("c"),group("G"),
  sdata("m2"),view(regconf(sProcSet(agent("a"))
  sProcSet(agent("b")) sProcSet(agent("c")),7),group("G"),
  sAgentList(agent("a")) sAgentList(agent("b"))
  sAgentList(agent("c")),0))))
  ...
sdelivered'(agent("b"),...
  sSMessageList(sdatamsg(agent("b"),group("G"),
  nondecryptable(sdatamsg(keyid(1),idPartialKey,
  sdata("m1"))),view(regconf(sProcSet(agent("a"))
  sProcSet(agent("b")) sProcSet(agent("c")),7), group("G"),
  sAgentList(agent("c")) sAgentList(agent("a")),0)))
  ...
  sSMessageList(sdatamsg(agent("c"),group("G"),
  sdata("m2"), view(regconf(sProcSet(agent("a"))
  sProcSet(agent("b")) sProcSet(agent("c")),7),group("G"),
  sAgentList(agent("a")) sAgentList(agent("b"))
  sAgentList(agent("c")),0))))
  ...
sdelivered'(agent("c"),...
  sSMessageList(sdatamsg(agent("c"),group("G"),
  sdata("m1"),view(regconf(sProcSet(agent("a"))
  sProcSet(agent("b")) sProcSet(agent("c")),7),group("G"),
  sAgentList(agent("c")) sAgentList(agent("a")),0)))
  ...
  sSMessageList(sdatamsg(agent("c"),group("G"),
  sdata("m2"),view(regconf(sProcSet(agent("a"))
  sProcSet(agent("b")) sProcSet(agent("c")),7),group("G"),
  sAgentList(agent("a")) sAgentList(agent("b"))
  sAgentList(agent("c")),0)))
...
sstate(agent("a"),eGroupSet,eGroupSet,eGroupSet,
       eGroupSet,eGroupSet,eGroupSet)
sstate(agent("c"),sGroupSet(group("G")),eGroupSet,
       eGroupSet,eGroupSet,eGroupSet,eGroupSet)
```

Using Lazy Key: In our second example, we select the lazy key establishment protocol and create the group *G*={*a*,*b*,*c*}. We then steer the system by forcing a partition that divides group *G*={*a*,*b*,*c*} in two subgroups *G1*={*a*} and *G2*={*b*,*c*} and triggers the EVS algorithm. Once new views have been delivered and no key has been created, we steer the system again by forcing the merge of both subgroups into *G*={*a*,*b*,*c*}, again triggering the EVS algorithm, but avoiding the key establishment protocol. Finally, the agent at host *c* requests to send a message to the group.

```
PERFORM(CHANGE(agent("a"),sProcSet(agent("a")))) ;
PERFORM(CHANGE(agent("b"),(sProcSet(agent("b"))
                           sProcSet(agent("c"))))) ;
PERFORM(CHANGE(agent("c"),(sProcSet(agent("b"))
                           sProcSet(agent("c"))))) ;
( PERFORM(EVS-START(agent("a"),true)) ||
  PERFORM(EVS-START(agent("b"),true)) ||
  PERFORM(EVS-START'(agent("c"),true))
) ;         --- a,b,and c finish successfully
...
PERFORM(CHANGE(agent("a"),(sProcSet(agent("a"))
                           sProcSet(agent("b"))
                           sProcSet(agent("c"))))) ;
PERFORM(CHANGE(agent("b"),(sProcSet(agent("a"))
                             sProcSet(agent("b"))
                           sProcSet(agent("c"))))) ;
PERFORM(CHANGE(agent("c"),(sProcSet(agent("a"))
                            sProcSet(agent("b"))
                           sProcSet(agent("c"))))) ;
( PERFORM(EVS-START(agent("a"),true)) ||
  PERFORM(EVS-START(agent("b"),true)) ||
  PERFORM(EVS-START'(agent("c"),true))
) ;         --- a,b,and c finish successfully
PERFORM(SENDGROUPMSG(agent("a"))) ;
PERFORM(SEND(agent("a"),13)) ;  --- a sends group message
PERFORM(RECEIVE(agent("a"),13)) ;
PERFORM(RECEIVE(agent("b"),13)) ;
PERFORM(RECEIVE(agent("c"),13)) ;
PERFORM(SENDGROUPMSG(agent("b"))) ;
PERFORM(SEND(agent("b"),14)) ;  --- b sends group message
PERFORM(RECEIVE(agent("a"),14)) ;
PERFORM(RECEIVE(agent("b"),14)) ;
PERFORM(RECEIVE(agent("c"),14)) ;
PERFORM(SENDGROUPMSG(agent("c"))) ;
PERFORM(SEND(agent("c"),15)) ;  --- c sends group message
PERFORM(RECEIVE(agent("a"),15)) ;
PERFORM(RECEIVE(agent("b"),15)) ;
PERFORM(RECEIVE(agent("c"),15)) ;
PERFORM(DELIVERCHANGE(agent("a"))) ;
PERFORM(DELIVERCHANGE(agent("b"))) ;
PERFORM(DELIVERCHANGE(agent("c"))) ;
PERFORM(MULTICAST-TEST) ;    ---no key has been generated
PERFORM(MULTICAST(agent("c"),group("G")))
```

Since no key has been generated, the agent at host *c* triggers the key establishment protocol asynchronously. However, the other members of the group are not aware of it and thus the agent at host *c* blocks indefinely. Furthermore, any other member that tries to send a message will also block and no membership change will alleviate this situation, as we can see in the following excerpt.

```
result State: ENABLED(MULTICAST(agent("b"),
              privategroup(agent("c"))))
network(sProcSet(agent("a")) sProcSet(agent("b"))
        sProcSet(agent("c")))
freshconf(5)
...
fresh(0)
freshkeyid(0)
...
CONTROLLER(WAITFOR(MULTICAST(agent("b"),
           privategroup(agent("c")))));
...
```

The blocking is caused by the lack of a synchronized initiation of the key agreement protocol (GDH2). Usually the membership change message serves as a synchronization barrier that forces everyone to update the membership and generate (or retrieve) a key for the new view, but in case lazy key establishment is selected, we need a way to tell everyone that a new key needs to be build (or retrieved) immediately. Furthermore, it is possible that members try to send a message almost at the same time and try to trigger more than once the key establishment protocol. We solved these issues by broadcasting a *force-key* message everytime a new or cached key is needed for the current view. Once a member receives this message, it moves from the *secure* state to *cascading-membership* state and the key establishment protocol execution proceeds normally. In order to avoid multiple *force-key* messages sent by different members almost at the same time, we allow only one multicast of a *force-key* message in a view. The following excerpt shows the final state of the system. After successfully generating a new key, the message was sent and then received by all members. Similarly to our previous example, the agent at host *b* disconnects, the agent at host *a* left the group and the agent at host *c* is a singleton group. It is noteworthy to mention that only one key was generated in 7 view changes, as we can witness by inspecting the local configuration and the associated membership-keyid list.

```
result State: network(sProcSet(agent("a"))
              sProcSet(agent("b")) sProcSet(agent("c")))
freshconf(8)
operational(agent("a"))
operational(agent("b"))
operational(agent("c"))
freshseq(35)
sp-receive-req(agent("a"))
sp-receive-req(agent("c"))
f-receive-req'(agent("a"))
f-receive-req'(agent("c"))
fresh(4)
freshkeyid(1)
ssp-receive-req'(agent("a"))
ssp-receive-req'(agent("c"))
CONTROLLER(eController)
a(7)
b(7)
c(3)
localconf(agent("a"),regconf(sProcSet(agent("a"))
          sProcSet(agent("b")) sProcSet(agent("c")),7))
localconf(agent("b"),regconf(sProcSet(agent("a"))
          sProcSet(agent("b")) sProcSet(agent("c")),7))
localconf(agent("c"),regconf(sProcSet(agent("a"))
          sProcSet(agent("b")) sProcSet(agent("c")),7))
...
sgroupmembsetlist(agent("a"),sGroupAssocMembSetListSet(
  groupassocmembsetlist(group("G"),
  sAssocMembSetList(assocmembset(sAgentSet(agent("a"))
  sAgentSet(agent("b")) sAgentSet(agent("c")),keyid(0))))))
sgroupmembsetlist(agent("c"),sGroupAssocMembSetListSet(
  groupassocmembsetlist(group("G"),
  sAssocMembSetList(assocmembset(sAgentSet(agent("a"))
  sAgentSet(agent("b")) sAgentSet(agent("c")),keyid(0))))))
...
```

VI. Experimental Evaluation

Although symbolic execution allows us to investigate certain scenarios that might be hard to generate in an experimental setting, quantitative information is needed to evaluate how practical and effective our generic optimizations are. In order to measure the overhead of adaptation,without re-implementing the specification, we developed a Maude API to systematically translate Maude commands into OTcl/C++ and enhanced the NS2 simulation framework to take commands from the Maude engine via the API. This gives us the opportunity to specify and plugin different network topologies, mobility and client connectivity models in NS2, without modifying the specification (and high-level algorithms) in Maude. More precisely, a thin hookup client in NS2 reports configuration changes to Maude via an action token. Since every action is associated to a rule in the specification, received tokens steer the symbolic execution, allowing the specification to react to the underlying network changes. Therefore, the integration of classical network simulation and formal specifications allows us to formally model, observe and evaluate different types of dynamic peer group systems.

Simulation Environment Setup

In our model, we assume that clients communicate with each other as long as they do not move out of each other's transmission range and that only one group is active in the network. We model join and leave events using a negative exponential distribution with parameters j-λ and l-λ, respectively. Similarly, we use a Poisson distribution with parameter m-η to model message requests from the application and a Weilbull failure rate function with parameters α and β to model node failures.

The simulations were performed using three network densities (sparse, medium and dense) and two commonly used mobility models (Random-Waypoint and Gauss-Markov). The network densities consists of 29, 149 and 271 nodes, respectively. All the scenarios share the following input parameters: j-$\lambda = 10$, l-$\lambda = 10$, m-$\eta = 2$, $\alpha = 1$ and $\beta = 2$. Initially, 20 members using an IEEE 802.11 radio and MAC model with random transmission ranges between 150m and 250m and 1Mb of available bandwidth are randomly distributed in a simulation space of 1024x1024 square meters. The node velocities are chosen from the interval 5-10 m/s and a speed standard deviation of 0.5. The nodes change speed and direction every 2.5s and the standard deviation for the angle is $\pi/4$. For the Random-Waypoint model, 7 attraction points were added. Each point had an attraction intensity taken from the interval 5-10, with a standard deviation of 20 and a uniform probability of pausing at the attraction point of 0.75. In order to obtain meaningful results, we repeated each scenario 50 times, aggregated and averaged the obtained results.

Experimental Strategy and Results

We start by measuring the the performance of our generic optimizations using EVS semantics as compared to the synchronized (VS semantics) version used by Secure Spread under failure-free scenarios. Next, we allow node failure and recovery, as well as intermittent disconnections, which create patterns of membership changes. While the system keeps track of every group member, we report partitions and merges on an aggregate basis to study the behavior and dynamics of the group as a whole. This allows us to evaluate the overall messaging overhead produced by the key establishment protocol and the total number of keys generated.

Basic Results: Figure 1 compares the overall messaging overhead produced by the execution of the key establishment protocol in failure-free scenarios in four different modes: (1) eager-VS: full-blown rekey on every membership or configuration change using VS semantics, (2) key-caching using EVS semantics, (3) lazy-key establishment using EVS semantics and (4) lazy+caching: a combined use of key caching and lazy key establishment using EVS semantics. In general, we observe that the EVS semantics has better messaging performance than VS semantics since does not require synchronization barriers in the form of flush acknowledgements. This reduces both the number of messages sent and delays induced by waiting time.

Impact of Failures and Transient Partitions: Figure 2 shows the overall messaging overhead produced by the four different modes in prone to failure scenarios. Not surprisingly, the Random-Waypoint model in a medium network density shows a high level of burstiness in membership changes due to repeated disconnections and failures at the attraction points (center figure). The same pattern of failures and disconnections in the Gauss-Markov model produces a high messaging overhead in the VS mode, but does not impact the performance of the EVS modes (left figure). However, the same pattern of failures and disconnections in a sparse network scenario creates a short lived subgroups. Thus, the difference of performance between key caching and eager-VS are the flush messages.

Exploring Scalability: Figure 3 shows our initial scalability results in a dense network using the Gauss-Markov model. We observe that achieving scalability is difficult due to frequent configuration changes caused by temporary partitions and merges.

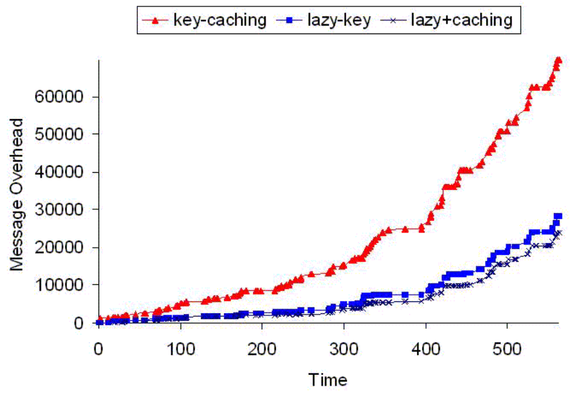
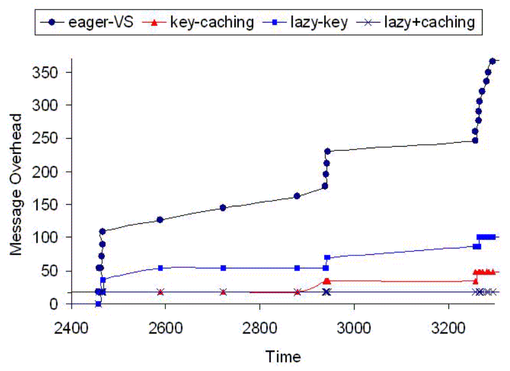


Fig. 3. Scalability of generic optimizations: Message overhead for key establishment in a dense network using the Gauss-Markov model.

VII. Concluding Remarks

Our effort to develop a formal specification of a secure GCS was twofold. First, we obtained a mathematically satisfactory and concise description that models the behavior of the system. In addition, we found subtle bugs that broke message ordering guarantees. Second, the formal specification was integrated into a classical network simulator and used as a tool for testing alternative designs, semantic guarantees, and extensions in functionality without the need to carry-out full-fledged implementations.

In this paper we have focussed on two dimensions of high-level adaptability in group communication, namely synchrony and security, as opposed to low-level adaptability of the underlying communication protocols, which we leave as future work. We have explored several solutions and built a formal prototype to validate our ideas and explore the properties of the new design. We have emphasized adaptability, because there is no one-size-fits-all solution given the diversity of application requirements that we are concerned with. We developed adaptation parameters that allow us to tailor (dynamically) the communication framework to specific application requirements. In the synchrony dimension, groups with different degrees of synchrony can coexist given that every group specifies its synchrony (VS or EVS), members can participate in several groups with different synchrony modes simultaneously. In the security dimension, each group specifies the degree of laziness of the key establishment protocol, which is not entirely independent of the degree of synchrony selected: (i) eager keying will trigger a rekey after every membership change; (ii) key caching will reuse previous cached keys accordingly; and (iii) lazy keying will delay rekeying until a message needs to be send. It is noteworthy that our approach is entirely generic in the sense that it is independent of the key



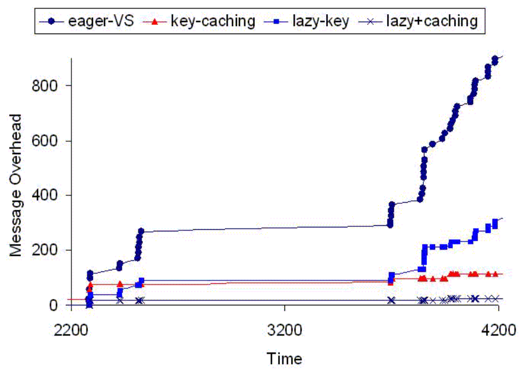
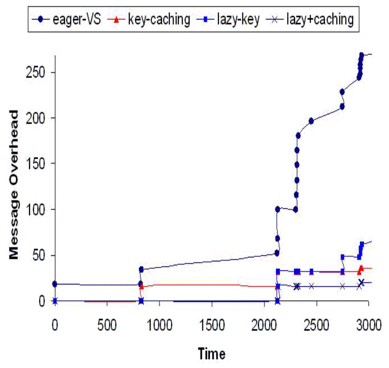
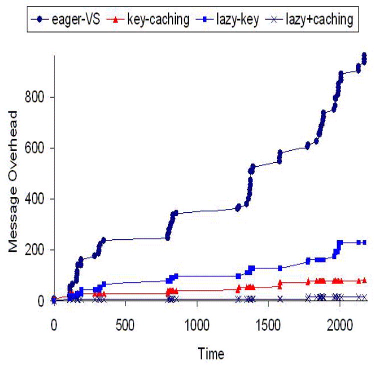


Fig. 1. Message Overhead for key establishment and generic optimizations (**Left:** Gauss-Markov model in a sparse network, **Right:** Random-Waypoint model in a medium density network).





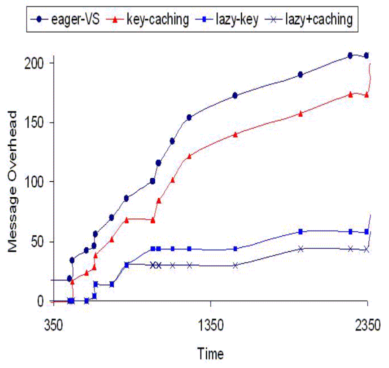


Fig. 2. Message Overhead for key establishment and generic optimizations in prone to failure scenarios (**Left:** Gauss-Markov model in a medium density network, **Center:** Random-Waypoint model in a medium density network, **Right:** Gauss-Markov model in a sparse network).

establishment protocol and the implementation of the group communication system.

Possible directions for future work include further generic optimizations for key management and secure multicasting, dynamic access control for a high-level enforcement of security requirements, adaptability to support group communication in mobile environments, and adaptability to QoS requirements such as timeliness constraints.

Acknowledgments

This research was supported by the Office of Naval Research under MURI Research Contract N00014-02-1-0715. We would like to thank Grit Denker for her collaboration in the specification of Secure Spread, and Yair Amir, John Schultz, and Gene Tsudik for various discussions in this context.

References

[1] Y. Amir, C. Nita-Rotaru, J. Stanton and G. Tzudik, "Secure Spread: An Integrated Architecture for Secure Group Communication," *IEEE Transactions on Dependable and Secure Computing*, vol. 2(3), 2005.

[2] O. Rodeh, K. Birman, M. Hayden, Z. Xiao and D. Dolev, "Ensemble Security," Cornell University, Tech. Rep. TR98-1703, 2000, Department of Computer Science.

[3] P. McDaniel, A. Prakash and P. Honeyman, "Antigone: A Flexible Communication for Secure Group Communication," in *Proceedings of the 8th USENIX Security Symposium*, 1999.

[4] K.P. Kihlstrom, L.E. Moser and P.M. Melliar-Smith, "The SecureRing Protocols for Securing Group Communication," in *IEEE 31st Hawaii International Conference on System Sciences*, 1998.

[5] M. K. Reiter, "Secure Agreement Protocols: Reliable and Atomic Group Multicast in Rampart," in *2nd ACM Conference on Computer and Communications Security*, 1994.

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

[7] Y. Kim, A. Perrig and G. Tsudik, "Communication-efficient Group Key Agreement," in *IFIP SEC 2001*, 2001.

[8] M. Steiner, G. Tsudik and M. Waidner, "Key Agreement in Dynamic Peer Groups," in *IEEE Transactions on Parallel and Distributed Systems*, 2000.

[9] A. Fekete, N. Lynch and A. Shvartsman, "Specifying and using a Partitionable Group Communication Service," in *16th Annual ACM Symposium on Principles of Distributed Computing*, 1997.

[10] S. Floyd,V. Jacobson, C. Liu, S. McCanne and L. Zhang, "A Reliable Multicast Framework for Light-weight Session and Application Level Framing," in *IEEE/ACM Transactions on Networking, (5):784-803*, 1997.

[11] Y. Amir, Y. Kim, C. Nita-Rotaru, J. Schultz, J. Stanton and G. Tsudik, "Secure Group Communication Using Robust Contributory Key Agreement," in *IEEE Transactions on Parallel and Distributed Systems*, 2004.

[12] R. van Renesse, K. Birman and S. Maffeis, "Horus: A Flexible Group Communication System," *Communication of the ACM*, vol. 39(4):76-83, 1996.

[13] L. E. Moser, Y. Amir, P. M. Melliar-Smith and D. A. Agarwal, "Extended Virtual Synchrony," in *14th International Conference on Distributed Computing Systems*, 1994.

[14] Y. Amir and J. Stanton, "The Spread Wide Area Group Communication

System," Johns Hopkins University, Tech. Rep. Technical Report CNDS-98-4, 1998.

[15] Y. Amir, D. Dolev, S. Kramer and D Malki, "Transis: A Communication Subsystem for High Availability," in *22nd International Symposium on Fault-Tolerant Computing Systems*, 1992.

[16] Y. Amir, L. E. Moser, P. M. Melliar-Smith, D. Agarwal and P. Ciarfella, "The Totem Single-Ring Ordering and Membership Protocol," in *ACM Transactions on Computer Systems*, 1995.

[17] 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 *20th International Conference on Distributed Computing Systems*, 2000.

[18] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer and C. Talcott, "The Maude 2.0 System," in *Rewriting Techniques and Applications (RTA 2003)*, ser. Lecture Notes in Computer Science, Robert Nieuwenhuis, Ed. Springer-Verlag, June 2003, pp. 76–87.

[19] J. Meseguer, "Conditional Rewriting Logic as a Unified Model of Concurrency," in *Theoretical Computer Science 96(1):73-155*, 1992.

[20] J. Schultz, "Partitionable Virtual Synchrony Using Extended Virtual Synchrony," Master Thesis, Department of Computer Science, Johns Hopkins University, 2001.

[21] C. Talcott, M.-O. Stehr and G. Denker, "Towards a Formal Specification of the Spread Group Communication System," 2004, website: http://formal.cs.uiuc.edu/stehr/spread_eng.html.

[22] Y. Kim, A. Perrig and G. Tsudik, "Simple and Fault Tolerant Key Agreement for Dynamic Collaborative Groups," in *7th ACM Conference on Computer and Commmunications Security*, 2000.