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

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 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.

1 Introduction

In recent years some secure group communication systems (GCS) have been developed [4, 6, 8, 9, 11]. However, GCS were designed to be highly efficient in local (wired) networks and do not consider mobility, temporary disconnections and real time constraints. 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 [12], a state-of-the-art GCS, uses key establishment protocols that stall all communication (at the application level), while the key is generated, and relies 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.

2 Formal Methodology

The general methodology we employ for system design and analysis is based on an executable specification language called Maude [5]. Its theoretical foundation is rewriting logic [2], 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 [10] with its EVS semantics. The more constrained VS semantics is provided by a specification of Flush Spread [3] on top of this. Independently, a specification of the Cliques toolkitinstantiated to the GDH protocol [7] 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 [12].

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.

2.1 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, $Sort_0 \ldots Sort_k$ 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, $Cond_1 \ \wedge \ \ldots \wedge \ Cond_k$ represent the set of conditions that must hold. We then give a set R of rewrite rules to specify state transitions as follows.

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

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.

2.2 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 [1].

3 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 [11] 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 [11] 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.

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 with 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 it impossible to provide different degrees of synchrony using the same specification. Therefore, we 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. *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. 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. As an example, we show the modified version of the receive rule 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))
```

4 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.

[1] The solution presented in [11] also has this problem.

Both approaches are *generic*, that is independent of the underlying protocol, and can be composed to further improve system performance without sacrifying 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.

4.1 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 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 [14]. The relevant properties for key establishment algorithms are of purely computational nature [13]: *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 [12]: *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 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 3) so that this extension does not cause any additional overhead.

4.2 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 temporarily 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 4.4).

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.

4.4 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:

- 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.

5 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.

6 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. Although existing network simulation frameworks provide realistic scenarios to evaluate the performance of our approach under varying degrees of change, they do not offer the capability of verifying the correctness of system properties. Therefore, 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 a thin hookup client in NS2. The hookup client schedules the simulation of received messages from the Maude engine, keeps track of message overheads and reports network events (network partition and merge, node failures, link availability, etc) to the Maude engine via a configuration change. This gives us the opportunity to keep the system specification and algorithms in Maude, while being able to specify and plugin different network topologies, mobility and client connectivity models in NS2. Consequently, we can quickly explore design decisions and validate the behavior of the system interactively and then, obtain initial performance measures without the error-prone and duplicated task of creating a prototype implementation for the network simulation framework.

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.

6.1 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 it 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.

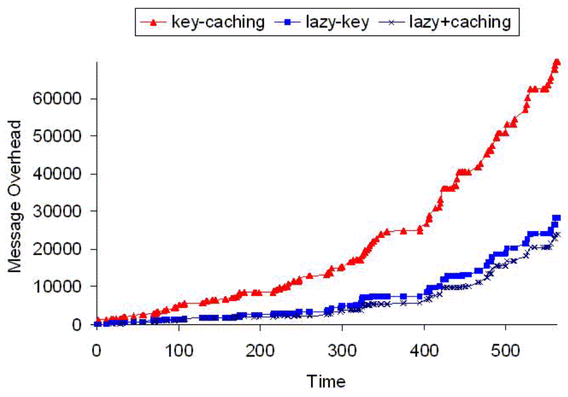


Figure 3. Scalability of generic optimizations: Message overhead for key establishment in a dense network using the Gauss-Markov model.

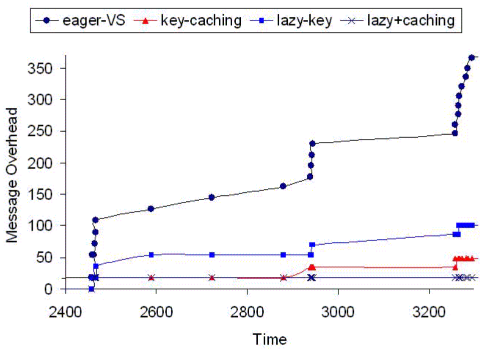
7 Concluding Remarks

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. 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. 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] C. Talcott, M.-O. Stehr and G. Denker. Towards a For-



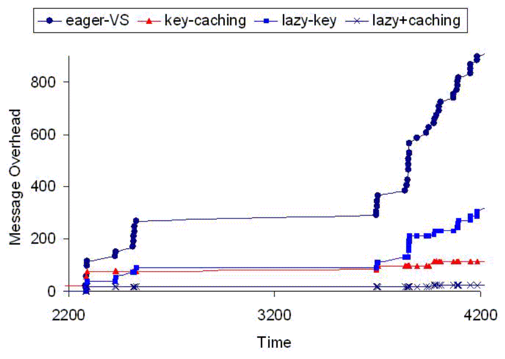


Figure 1. Message Overhead for key establishment and generic optimizations. Left: Gauss-Markov model in a sparse network. Right: Random-Waypoint model in a medium density network.

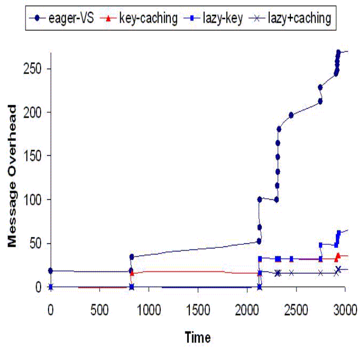
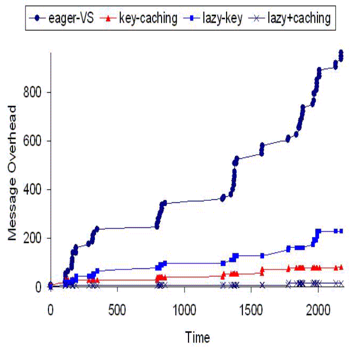
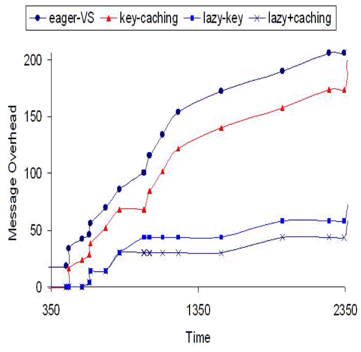




Figure 2. Message Overhead for key establishment and generic optimizations in prone to failure scenarios. Left: Gauss-Markov model in a medium density network. Center: Random-Waypoint model in a medium density network. Right: Gauss-Markov model in a sparse network.

mal Specification of the Spread Group Communication System. Website: `http://formal.cs.uiuc.edu/stehr/spread_eng.html`, 2004.

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

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

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

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

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

[8] O. Rodeh, K. Birman, M. Hayden, Z. Xiao and D. Dolev. Ensemble Security. Technical Report TR98-1703, Cornell University, 2000. Department of Computer Science.

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

[10] Y. Amir and J. Stanton. The Spread Wide Area Group Communication System. Technical Report Technical Report CNDS-98-4, Johns Hopkins University, 1998.

[11] 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*, 2(3), 2005.

[12] 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.

[13] 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.

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