A study of Group Rekeying

Ohad Rodeh, Kenneth P. Birman, Danny Dolev *

March 16, 2000

Abstract

In this paper we study the key management problem, in the context of Group Communication Systems (GCS). GCSs are mid-sized systems, scaling up to 100 members. We present a side-by-side comparison of three ways of managing keys, studing bandwidth and latency.

1 Introduction

With the advent of the Internet, and wide-spread use of communicating applications, requirements for data privacy and integrity have increased. This study focuses on applications exhibiting group-patterns, where data-sharing frequently occurs in a medium sized group. Sharing typically occurs in applications such as: (1) VPNs: Group collaboration and conferencing applications that secure the "conference" (2) Administration and management of a LAN, where one would like security against infrastructure disruption (3) Remote learning or briefings (in military or business situations, these might need to be secure) (4) Various kinds of replicated services such as white pages (NIS).

Today, most such architectures are either rather static, simplifying management, or simply run without security at all. With better key management tools, we may eventually see these kinds of applications begin to routinely secure the application. A good tool should be tolerant of failures, or denial of service can easily occur, and should scale to group sizes of 100 or 200 participants. Our work explores a solution to group keying and focuses on an experimental question: how do the various options perform and how much overhead do they bring to the table?

Our experimental setting is our Group Communication System (GCS), Ensemble [18]. This system belongs to a family of such systems built around the world. Brevity precludes us from listing all such systems, the interested reader is refered to [1, 6]. Ensemble evolved from Isis [6], Horus [19], and Transis [21]. GCSs have been used for many different applications, such as: clustering [4, 20], air traffic control, stock exchanges [2], and more.

*The authors were supported in part by the Israeli Ministry of Science grant number 032-7892, and by DARPA/RADC grant f30602-99-1-6532.

A GCS provides reliable multicast and membership services to groups of processes. It monitors and connects member processes on the network, providing them with a consistent view of group membership.

A group of members (processes) can be efficiently protected using a single symmetric encryption key. This key is securely communicated to all group members, which subsequentally use it to encrypt/decrypt group messages. The group-key is securely switched whenever the group membership changes, thereby preventing old members from eavesdropping on current group conversations. This also prevents new members from tapping into past conversations. The challenge is to create an efficient and fast key-switch algorithm that can handle large groups and a high rate of membership changes.

A secure GCS ensures that all members in a view are authentic and authorized. It provides a secure key with which all group communication is protected. Members outside the group cannot listen in on group-communications. A partial list of secure GCSs include Antigone [15], and Spread [22]. Spread has been secured using the Cliques cryptographic toolkit [10] that provides strong security guaranties for group-keys. In fact, Cliques provides stronger guaranties than Ensemble, though stronger group-keys come at an unavoidable computational cost. Antigone has been used to secure video conferences over the web, using the VIC and VAT tools. However, to date, it has not been provided with a fault tolerance architecture. The Totem [9], and Rampart [16] systems can survive Byzantine faults, at the cost of a further degradation of performance.

The Ensemble security architecture [14] has evolved from seminal work by Reiter [12, 17] done in the context of the Isis and Horus systems. These results show how group keying can be integrated with a Group Membership Protocol (GMP) to support such functions as securely managing keys at the group members, securely rekeying, supporting *secure channels* between members (discussed below), HMAC-ing[1] messages and encrypting the data segments of messages.

We have implemented three different key-management schemes in the Ensemble system. Since the basic security architecture is shared amongst the protocols, differences occur when group members join and leave, and rekeying is required. In this paper we report on an experimental and analytical comparative study of these three schemes. We used a large set of machines in a LAN setting, to isolate and compare different protocol characteristics. We measure total system load, network bandwidth, and latencies that our protocols exhibit under different scenarios.

2 Model

The "universe" for the purposes of this paper is comprised of a set of machines connected through the Internet. Machines, or processes, can communicate with each other by passing messages through the network. The system is asynchronous: clock drifts are unbounded and messages may be arbitrarily delayed or lost in the network. We do not consider Byzantine failures. The network can split into several disjoint components allowing only machines in the same component to pass messages to each other. A GCS interposes between the network and the application, and

[1] An HMAC is a Keyed Hash [5] that can be based on any interactive cryptographic hash (e.g., MD5 or SHA-1). It is used to protect the integrity of a message.

transforms a world of unpredictable failures, delays, transient problems, and message loss, into a better behaved one [8].

A GCS provides reliable multicast and membership services to groups of processes. Processes may dynamically join and leave a group, and group components can merge through the GCS protocols and state-transfer. All processes, inside a group, have knowledge of the set of currently live and accessible members (the group *view*). Furthermore, a GCS guarantees agreement on the current view. This allows a group-leader to be easily chosen: it is the process with lexicographically "smallest" name.

To achieve fault-tolerance, GCSs require all members to actively participate in failure-detection, membership, flow-control, and reliability protocols. Therefore, such systems have inherently limited scalability. We have managed to scale Ensemble to 100 members per group, but no more. For a detailed study of this problem, the interested reader is refered to [7, 2]. In this paper, we do not discuss configurations of more than 100 members.

We assume processes in a group have access to trusted authentication and authorization services, as well as to a local key-generation facility. We also assume that the authentication service allows processes to open *secure channels*. A secure channel between a pair of processes allows the secure exchange of private information.

Ensemble allows the creation of secure-groups where all group members agree on a single symmetric key. Only trusted and authorized members are allowed into the group. Since all members use the same key to HMAC and encrypt their messages, no intruder can attack the group or purport to be part of it.

Since all group members have the same view of the membership, we number them lexicographically from p_1 to p_n. Implicitly, we shall refer to the "group G". This is our reference group, $p_1 \dots p_n \in G$. When we refer to the group leader, we implicitly refer to member number 1 (denoted p_1).

All members make use of a *secure channel* abstraction. Each process keeps a cache of secure channels. This cache is used for the exchange of private messages in G. For example, if member p needs to pass private message m to member q then p's cache is queried. If a secure channel to q with key K_{pq} already exists then m is encrypted with K_{pq} and sent pt-2-pt to q. If the channel does not exist, then a handshaking protocol, using a Diffie-Hellman exchange, is used to securely agree on a key K_{pq} between p and q. The channel is added to the cache, and m is encrypted with K_{pq} and sent to q.

Integer module exponentiations, needed for a Diffie-Hellman exchange, are expensive. We used a 500Mhz PentiumIII with 256Mbytes of memory, running the Linux2.2 OS for speed measurements. An exponentiation with a 1024bit key using the OpenSSL [11] cryptographic library was clocked at 40 milliseconds. Setting up a secure channel requires two messages containing 1024bit long integers. Hence, we view the establishment of secure channels as expensive, in terms of both bandwidth and CPU. One of the goals of our algorithms is to reduce the need of their use.

3 Algorithm description

We compare three different algorithms: Basic,Binary, and Dist [13].

The Basic algorithm uses a simple method to rekey a group, see Figure 1(a). The leader chooses a new key and disseminates it to the members using secure channels. The members send an acknowledgment back to the leader. Once the leader gets acknowledgments from all the members, it performs a view-change, and switches the group to the new key.

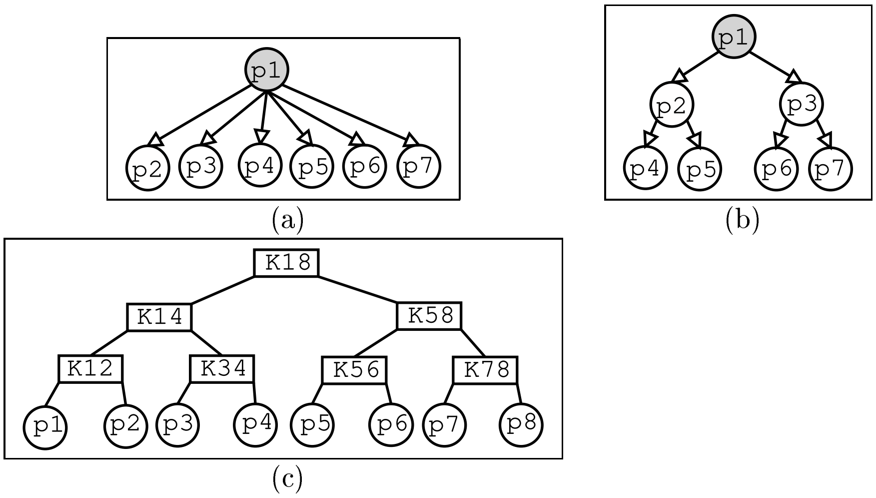

Figure 1: (a) The communication pattern of Basic. (b) The communication pattern of Binary. (c) An example of a graph of keys.

As group size grows, Basic will have a problem with *ack-implosion* at the leader. To avoid this situation, a fixed-degree tree is used to collect acknowledgments. When a tree leaf receives the new key, it sends an *ack* (short for acknowledgment) to its father. When the father receives acks from all its children, and it receives the key from the leader, it sends an ack up the tree. When the leader receives an ack from all its children, it decides that the algorithm is complete, and it performs a view change with the new key installed. Currently we use a tree of degree six in our system, though this is a settable parameter.

Algorithms Binary and Dist share the basic communication pattern with Basic. They perform key-dissemination, and need to collect acknowledgments to detect termination. Therefore, they all share the same tree-based algorithm for ack collection. We omit henceforth the ack collection stage from algorithm description.

The Binary protocol improves Basic. The major problem with the Basic protocol is that all load is concentrated at the leader. The leader needs to build secure channels to all the members. This is costly, as expensive integer exponentiations must be used.

The idea is to spread communication and computation load across the group members. The algorithm uses a binary tree of secure channels to disseminate the new key, see Figure 1(b). A binary-tree is laid over the group. The algorithm proceeds in three stages. (1) The leader chooses a new (random) group-key. (2) It sends it through secure channels to its children (3) The children pass it recursively to their children through secure channels.

Dist takes a completely different approach to the key-management problem. It is based on a centralized scheme suggested by Wong, Gauda, and, Lam (WGL) [3]. Briefly, using WGL, a graph of keys is laid over the group. In Figure 1(c) an example of a group of 8 members is shown. Each member knows all the keys on the route from itself to the root. The root, key K_{18}, is the group key. In WGL, a centralized server builds the key graph and has knowledge of all the keys. Using such a tree enables fast group rekeying when members join and leave, on the order of $log_2 n$ operations. In Dist, there is no centralized server, and the keygraph is built in a distributed manner.

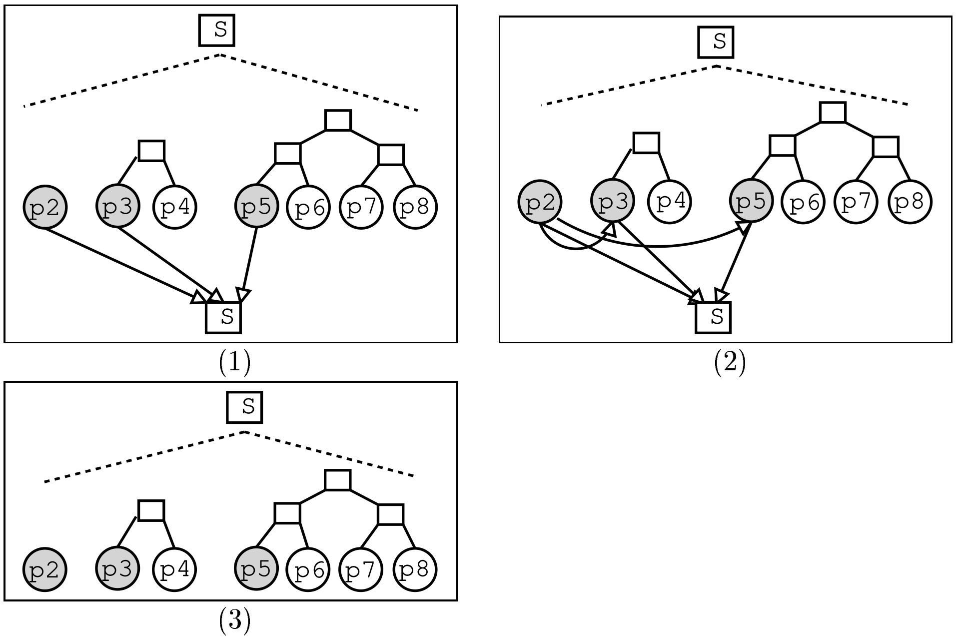

Figure 2: The communication pattern of Dist. In this figure we denote the leader as S ("server"), and separate it from the group. This is done for clarity, in fact, S is actually an alias for member p_2. (1) Subleaders send their key-graph information to the leader. The leader computes the merged key-graph. (2) Subleaders follow the leader's instructions. They choose and send new subkeys. (3) The leader merges together all pt-2-pt messages into one bundle and multicasts it. Members decrypt their key sets from it.

The protocol works as follows (see Figure 2):

I: Information is gathered from sub-leaders. A subleader is a member that is the leader of an independent key-tree. Each subleader sends to the leader the structure of its subtree. The leader performs a local computation and decides what the best method of merging the trees is. It multicasts the merged tree with a set of merge-instructions to the group.

II: Subleaders follow the leader's instructions. Using the instructions, Subleaders choose new subkeys. Some keys are sent directly to the leader, and some sent through secure-channels to other subleaders. A subleader receiving a key, encrypts it with its subtree key, and forwarded it to the leader.

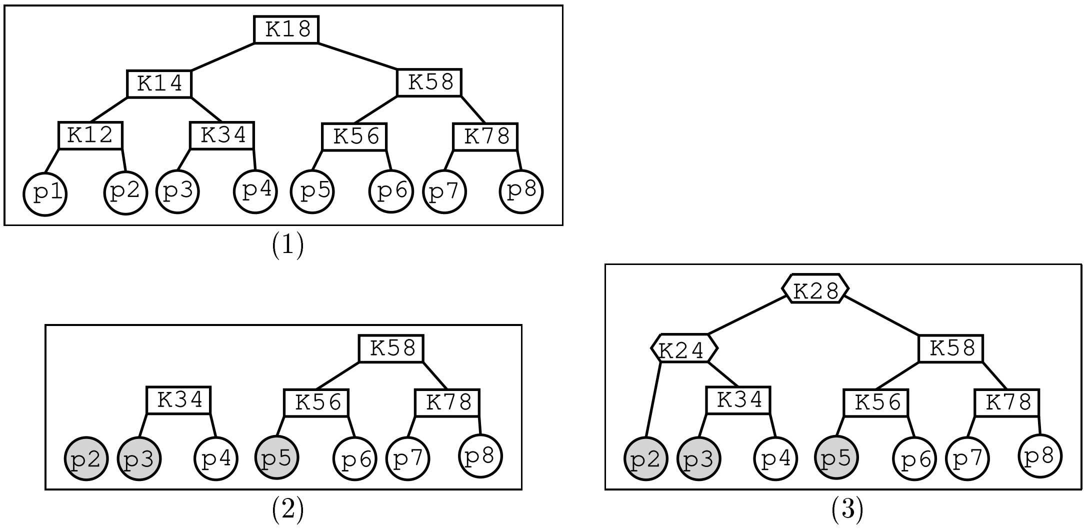

Figure 3: The effect of leave on a group key-graph of a group G of eight members. (1) The initial keygraph. (2) The tree after member p_1 leaves. (3) The merged tree.

III: Final stage: multicast. The leader bundles all pt-2-pt messages that it has received into a single message and multicasts it. The bundled message includes a set of encrypted keys. From this bundle, members compute their respective sets of keys. This set is guaranteed to be the set of keys from themselves to the root.

To clarify the workings of Dist, examine the execution from Figure 2 in detail. The starting point for the figure is a group G of eight members $\{p_1, \dots p_8\}$ with the key graph described in Figure 3(1). Member p_1 leaves G. All keys known to p_1 must be discarded, this includes keys K_{12}, K_{14}, and K_{18}. This operation splits G into three separate subtrees, (see Figure 3(2)) the first includes $\{p_2\}$, the second $\{p_3, p_4\}$, and the third includes $\{p_5, p_6, p_7, p_8\}$. In the first phase, subleaders p_2, p_3, and p_5 send their keygraphs to the leader. The leader computes the best tree mergable from these subtrees and multicasts it, see Figure 3(3). The missing subkeys are K_{24} and K_{28}. The three subleaders, together with the leader, engage in a protocol to choose and securely disseminate K_{24} and K_{28}. At the end of the protocol, all members receive their respective key sets, in particular, they all know the common group key that is at the root of the tree. This key can now be used to encrypt and sign all inner-group messages.

4 Analysis

Secure channel caching had a profound effect on the final performance of our protocols, though we did not realize this at first. Since this is the only from of caching considered in this paper, we simply refer to the *caching* optimization.

Initially, we thought that caching would improve protocol performance, for all protocols. This is however not true, some protocols enjoy performance improvements to a much greater extent than others. Nonetheless, in this section, we provide an analysis of the protocol performance, without considering caching effects.

There are two benchmarks against which we measure the protocols. These are named *JoinLeave* and *AllOne*. In the JoinLeave scenario, a member leaves, another member joins, and then a rekey operation is performed. In the AllOne scenario, G is initially in a state where all members are in singleton groups, $\{p_1\}, \{p_2\}, \ldots, \{p_n\}$. G merges into $\{p_1, \ldots, p_n\}$, and a rekey operation is performed. In both cases, we measure:

Load: Total number of integer exponentiations performed in the group.

Latency: Latency of integer exponentiation operations. This means, the longest causal chain of dependent Diffie-Hellman exchanges. For example, in the Binary algorithm, this chain will likely be of logarithmic length.

Total multicast size: Total number of bytes multicasted.

Communication Latency: Communication latency, in seconds.

We try to minimize the number of exponentiations, and this is the motivation for our statistics. In essence, the first statistic (*Load*) measures the total CPU load on the system. The second statistic (*Latency*) measures how long, in terms of accumulated CPU time, it take to perform a rekey. A measure of total load is the amount of bytes multicasted, only Dist uses multicast, and we are interested in quantifying the amount of bandwidth used. Finally, we measure communication latency that needs to be added to CPU latency for the total.

Due to the caching optimization for secure channels, a variable number of channels are produced for each rekey operation. This adds variance to statistics I,II, and IV. In this section, we do not take into account the channels that are already cached. This optimization is considered in the performance section where actual protocol performance is measured.

Table 1 below depicts costs for each of the protocols under the two benchmarks. There are 5 entries per box:

chans: The number of secure channels required. This should be viewed as an upper limit on the number of channels that are actually created at run time.

pt2pt: The number of point-to-point messages sent.

bytes: The number of bytes sent point-to-point.

mcast: The number of multicast messages sent.

bytes: The number of bytes sent through multicast.

There are several symbols we use:

A**:** An acknowledgment, 1 byte.

K**:** A key, 16 bytes.

$T(n)$**:** A structure describing the group key-tree. The key-tree contain inner-nodes, and leaf nodes. Inner-nodes contain a symbolic key name and up to two subtree pointers. Leaf nodes contain a group-member name. $\|T(n)\|$ is proportional to n.

$I(x)$**:** A set of instructions sent to the leaders describing what measures to take in order to merge the group key-tree. The instructions include a record of keys to create and distribute. The size of the instructions varies. In the JoinLeave case, there are $log_2 n$ records. In the AllOne case, the number of records is n.

Both $T(n)$ and $I(x)$ use symbolic member and key names. To simplify tree merge operations, all member and key names are required to be unique across all possible views. A simple approach is to use a machine's IP address combined with a counter. However, this would require 20bytes for the representation of key and member names. Instead, we rely on the fact that the group has less than 256 members, to encode the names in a single byte. For member names, we use a member's *rank* in the group. Since the group view is agreed, all members have agreed *ranks*, between 0 and $n-1$. Using ranks is tricky however, as member ranks change when view changes occur, requiring translation. A similar technique can be used for key-names, as the number of keys is smaller than the number of members. This technique pushes record size to a bare minimum, reducing message sizes.

		Basic/Binary	Dist
JoinLeave	chans	$n-1$	$log_2 n$
	pt2pt	$n-1$	$3log_2 n$
	bytes	$K(n-1)$	$2\|T(n)\|) + 2Klog_2 n$
	mcast		3
	bytes		$\|T(n)\| + \|I(log_2 n)\| + 2Klog_2 n$
AllOne	chans	$n-1$	$n-1$
	pt2pt	$n-1$	$3n-3$
	bytes	$K(n-1)$	$2\|T(n)\| + 2K(n-1)$
	mcast		3
	bytes		$\|T(n)\| + \|I(n)\| + Kn$

Table 1: Analytic comparison table. We have removed the compulsory acknowledgment messages from the tables, as it is shared among all the protocols. This cost comes to an additional $n-1$ messages of total size $A(n-1)$.

Figure 1 shows the analysis for the three algorithms under the AllOne and JoinLeave benchmarks. We have removed the ack collection costs from the table for simplicity. This phase costs $n-1$ pt-2-pt messages and $A(n-1)$ bytes.

In the JoinLeave case, we excpect Dist to have superior performance since it uses a logarithmic number of channels and messages. Furthermore, the total number of bytes sent is lower. Dist should be inferior in the AllOne case, since it uses more messages, and total message size is higher.

To give a better estimate for the expected latency, we analyze the AllOne case. In Basic we expect latency n, since the leader creates $n-1$ channels, costing $n-1$ consecutive exponentiations. The last member must complete the Diffie-Hellman exchange, adding another operation for a total of n. In Binary, the asymptotic latency is $3(log_2 n - 1)$. To see this, we examine a simple tree of size 3. Group G comprises of p_1, p_2, and p_3. Member p_1 is the root p_2 and p_3 are its children. Member p_1 (1) chooses a new key K (2) opens a channel to p_2, and sends K encrypted pt-2-pt (3) opens a channel to p_3 and sends K encrypted pt-2-pt. This costs two operations. Member p_3 completes the Diffie-Hellman exchange, adding one to the total latency. Notice that p_2 can perform its exponentiation while p_1 is performing its second exponentiation (for p_3). Hence, the total latency is three. For full binary trees, the latency increases by three for each level of tree. The asymptotic latency is $3(log_2 n - 1)$. In Dist all channels are created in the same round, and the most highly loaded member, m_h, creates $log_2 n$ channels. This incurs a series of $log_2 n$ back-to-back exponentiations at m_h. The receiving members can do the exponentiations in parallel, only the last member adds decryption to the latency for a total of $log_2 n + 1$.

5 Performance

We used several dozen machines, all Pentium boxes ranging from PentiumI 133Mhz to PentiumIII 500Mhz. Our machines run the Linux or BSDI operating system, with a 10Mbit/sec switched Ethernet to connect them. Only a subset supports multicast, so we could measure network latency only on that subset. The machines were all lightly loaded, and the network was fairly quite during measurements, this provided high quality measurements. Each measurement was repeated more than 100 times, and results were averaged. We had initially expected the measurements to confirm our analytic analysis, and show that Dist is better in all cases. However, this is not the case, as we shall see below.

Figure 4 shows the performance of the three algorithms under the AllOne benchmark. All algorithms have equal load $2(n-1)$. Since private information has to be passed to each member of the group, a connected graph of secure channels is a minimum. Such a graph includes at least $n-1$ edges, hence, $2(n-1)$ is optimal. Initially, the cache is empty for this scenario, and performance is precisely as we foresaw analitically. The best algorithm in terms of exponentiation latency is Dist. The Binary algorithm has latency $3(log_2 n - 1)$. In Basic, the latency is n, since the leader performs $n-1$ exponentiations, and the last member adds a single operation for total of n.

The total multicast size used by Dist is in the KBytes range, but should not be a problem for modern networks. Even a 10Mbit/sec Ethernet can easily route such traffic. Communication latency for all algorithms is less that 30ms, and rises slowly with group size, we believe that even with 100 members, this is insignificant compared with the accumulated cost of Diffie-Hellman exchanges.

Figures 6(1a-4a) show the performance of the three algorithms under the JoinLeave benchmark. The best algorithm, in terms of average number of exponentiations is Basic. It has constant latency 3, and constant load 4. It suffers from high variance in both measures because when the leader is replaced, all its secure channels are lost, and the new leader has to build them from scratch. In favorable runs, latency and load are two. In bad scenarios, it is n and $2(n-1)$ respectively.

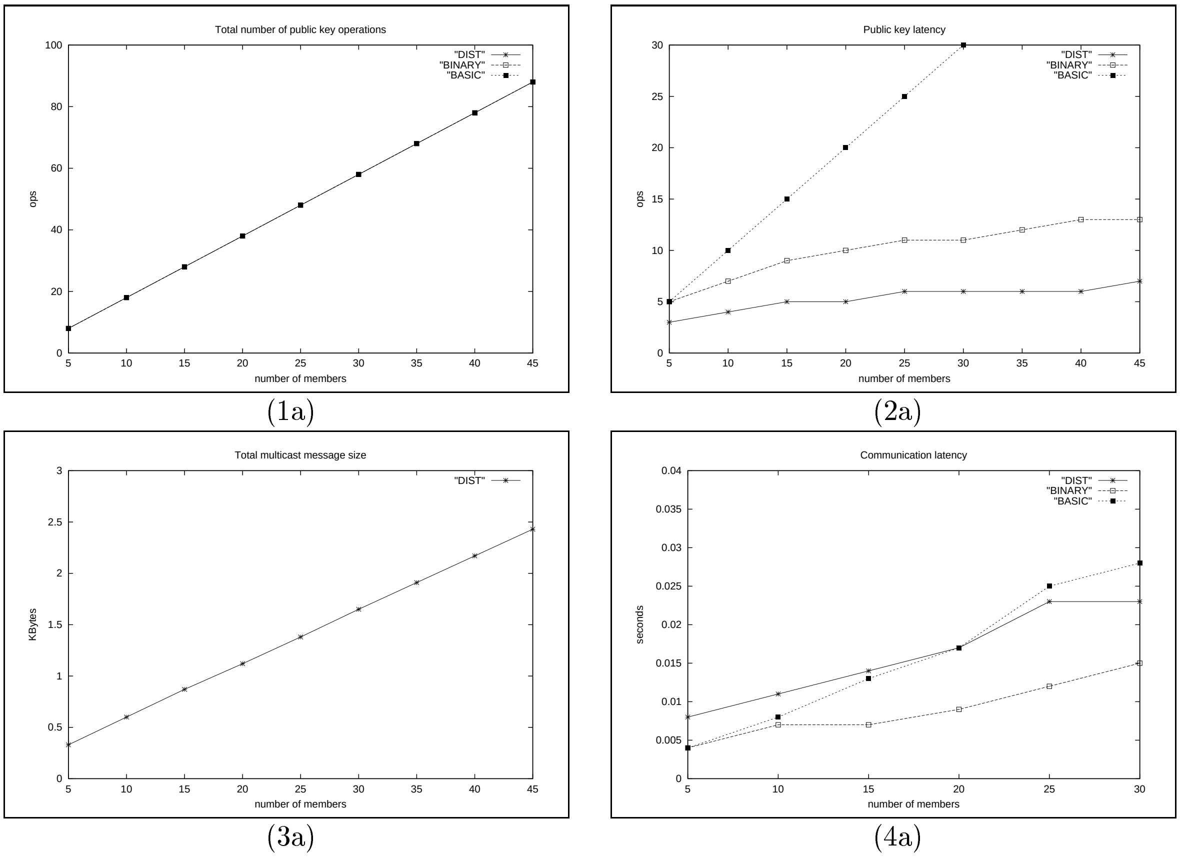

Figure 4: Performance comparison of the three algorithms for the AllOne case.

The Binary algorithm has variable latency and load. Both depend on the effectiveness of the caching optimization. When members join and leave, many edges must be "moved" in the binary graph. In the majority of cases, it is low level members that join and leave. This allows missing edges to be created simultaneously, incurring little latency. The total load depends on the amount of channels that are in the cache at the time of rekeying. Experimental evidence shows that caching is very effective in this case, latency is around 2.8, and total load is close to logarithmic.

For Dist, when a member leaves, the group key-tree is split into $log_2 n$ subtrees that need to be reconnected. A newly joined member adds another subtree of size one. In total, $log_2 n$ channels are needed to interconnect the subtrees. The majority of these channels are not cached, and the cost is average latency $log_2 n + 1$, and load $2log_2 n$. Dist has the highest latency (by a factor of two), and its load is similar to Binary.

Dist sends less than 1KBytes for a 45 member group, which is insignificant. Communication latencies are comparable, at 30 members they stand at around 20ms, with little difference between the algorithms.

To summarize, Dist does not overload the network and has comparable latency to the other protocols. It has comparable exponentiation load, though Basic and Binary are a little better. The surprising result is that Basic, the simplest algorithm, seems to come out best. It has good communication latency, constant exponentiation latency, and constant load. Furthermore, Basic and Binary both have better latency than Dist. This was a surprise, since they use a linear number of secure channels, while Dist was designed to use a logarithmic number of channels. It seems that the simple algorithms enjoy much better *cache locality* than the complex algorithm.

However, we were worried about variance exhibited by both Basic and Binary. We used another scenario to examine comparative performance, Specifically, we examined the case where the group leader is removed.

In Figures 6(1b-4b) the performance of the three algorithms for the case where the leader dies is shown side by side. In Basic the new leader has to create channels to $n-1$ members. Therefore the latency is n, and total load is $2(n-1)$. In Binary, when the leader dies, many tree edges must be "moved". In fact, almost all edges must be moved, and about *half* are not cached. Hence, the total load is $n/2$. Latency is almost the worst case $3(log_2 n - 1)$, since so few channels are cached. Dist has logarithmic latency $log_2 n + 1$ and logarithmic total load $2 log_2 n$. In the graph this is a little better due to caching effects. As in the previous graph, communication latencies are comparable, and multicast load by Dist is insignificant.

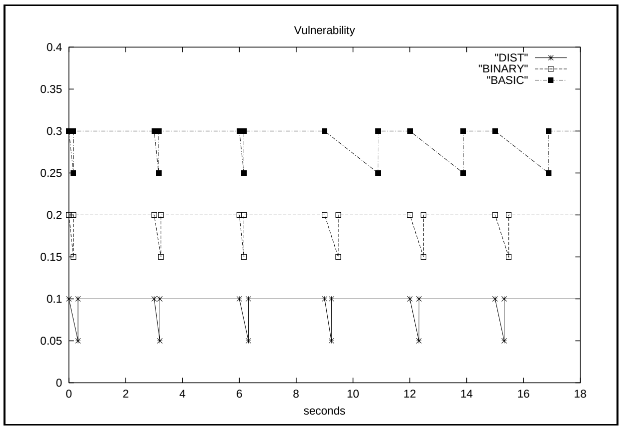

Figure 5: Vulnerability of the three protocols under failure conditions.

To cap this section, we provide a close look at an important measure of security. An application is vulnerable to attack while it is switching keys, as the old key is used while it is already known to be compromised. To depict this situation, we chose a 45 member group, and a scenario where three non-leader faults are injected, followed by three leader-failures. For each failure, we depict the time it takes to recover assuming that an exponentiation takes 40 milliseconds, and that all protocols take 80 milliseconds to complete (this includes the default group-communication view-change protocol). Figure 5 depicts secure system state as "1" and insecure as "0". The three protocols are on par, for the regular case, and Basic is the best. If the leader fails, then Dist is the best and Basic completely fails, Binary is somewhere in the middle.

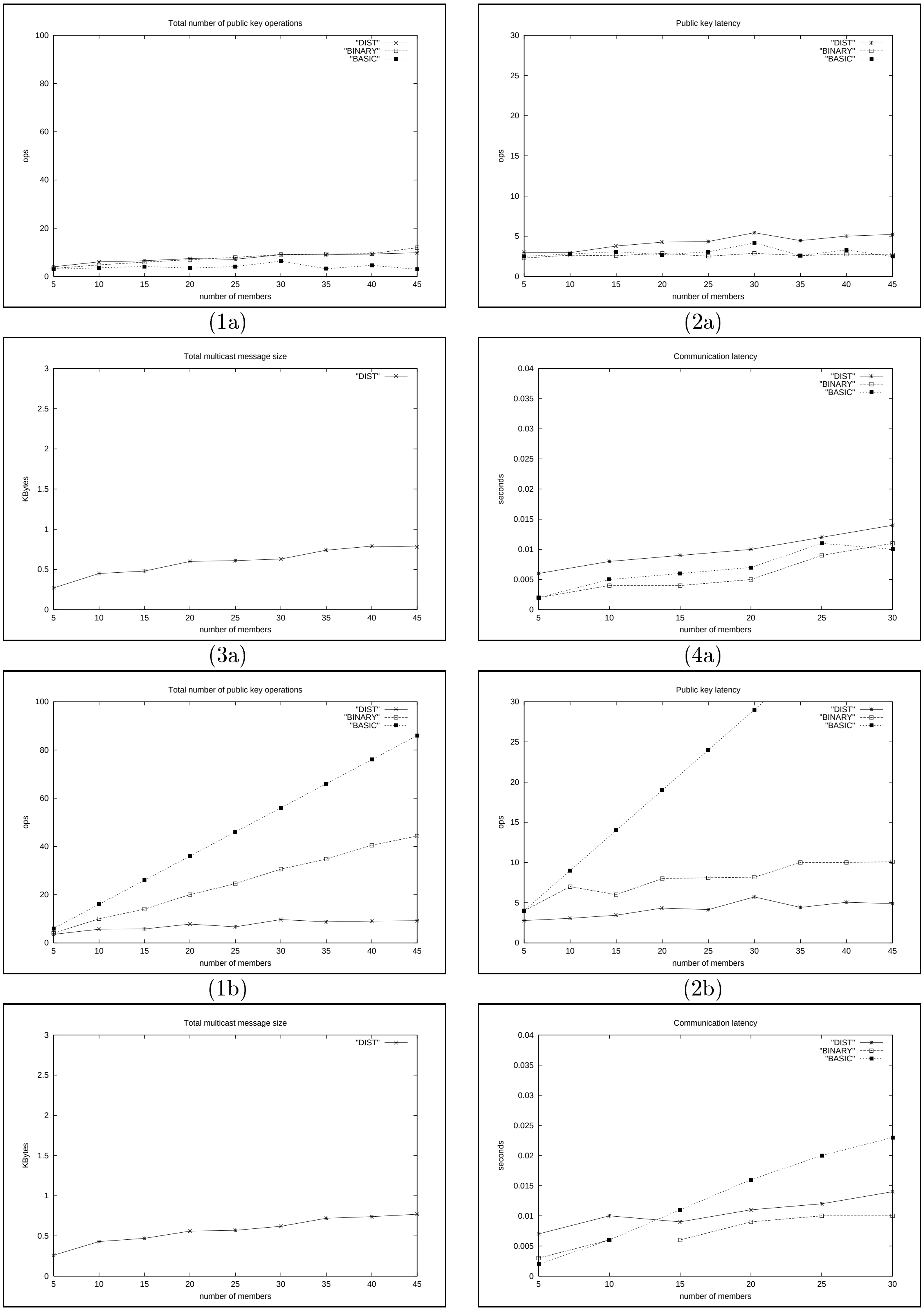

Figure 6: Performance comparison of the three algorithms for the JoinLeave case. Figures (1a-4a), the regular case. Figures (1b-4b), the leader dies.

6 Conclusions

We have compared three different protocols for group rekeying. Of the protocols described, Basic and Binary cost less but are severely disrupted by certain classes of failures, exposing the user to potentially extended periods when a compromised key might continue to be used. Dist, is just a bit more costly, but substantially reduces potential exposure.

Rekeying is not a very common operation, and our investigation shows that all three protocols are roughly comparable in most situations. Developers expect predictability, and Dist has the major advantage of giving the same performance no matter what process fails. In contrast, by choosing Basic or Binary, the developer would gain a small improvement in average latency at the cost of an extremely disruptive delay if the leader happens to fail. We believe that this consideration establishes Dist as the best algorithm within the group. At the cost of a relatively minor loss of performance, the protocol gives a form of fault-tolerance that involves greatly reduced disruption when a worst-case failure occurs.

7 Acknowledgements

We would like to thank Gene Tsudik, and Tal Anker for helpful discussions.

References

[1] K. Birman. *Building Secure and Reliable Network Applications*. Manning Publishing Company/Prentice Hall, 1997.

[2] K. P. Birman. A review of experiences with reliable multicast. *Software, Practice and Experience*, 29(9):741–774, Sept 1999.

[3] C.K. Wong, M. Gouda, and S.S. Lam. Secure group communication using key graphs. In *ACM SIGGCOM*. ACM, September 1998.

[4] Gera Goft and Esti Yeger Lotem. The as/400 cluster engine: A case study. In *International Workshop on Group Communication (IWGC'99)*, September 1999.

[5] H. Krawczyk, M. Bellare, and R. Canetti. Hmac: Keyed-hashing for message authentication. RFC 2104, IETF, Febuary 1997.

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

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

[8] Kenneth Birman and Thomas Joseph. Exploiting virtual synchrony in distributed systems. In *11th ACM Symposium on Operating Systems Principles*, February 1987.

[9] K.P. Kihlstrom, L.E. Moser, and P.M. Melliar-Smith. The securering protocols for securing group communication. In *Hawaii International Conference on System Sciences*, volume 3(31), pages 317–326, January 1998.

[10] M. Steiner, G. Tsudik, and M. Waidner. Cliques: A new approach to group key agreement. In *IEEE International Conference on Distributed Computing Systems (ICDCS'98)*, May 1998.

[11] Mark J. Cox, Ralf S. Engelschall, Dr. Stephen Henson, Ben Laurie, Eric A. Young, and Tim J. Hudson. Open ssl. http://www.openssl.org.

[12] M.K. Reiter, K.P. Birman, and L. Gong. Integrating security in a group oriented distributed system. TR 92-1269, Department of Conmputer Science, University of Cornell, February 1992.

[13] O. Rodeh, K. P. Birman, and D. Dolev. Optimized group rekey for group communication systems. In *Symposium Network and Distributed System Security*, Febuary 2000. To appear.

[14] O. Rodeh, K.P. Birman, M. Hayden, Z. Xiao, and D. Dolev. Ensemble security. TR 1703, Department of Conmputer Science, University of Cornell, 1998.

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

[16] M. Reiter. Secure agreement protocols: Reliable and atomic group multicast in rampart. In *ACM Conference on Computer and Communication Security*, pages 68–80, November 1994.

[17] M. Reiter, K.P., Birman, and R. Renesse. A security arcihtecture for fault-tolerant systems. *ACM Transactions on Computer Systems*, 4(12), November 1994.

[18] R.V. Renesse, K. P. Birman, M. Hayden, A. Vaysburd, and D. Karr. Building adaptive systems using ensemble. TR 97-1638, Cornell University, July 1997.

[19] R.V. Renesse, K.P. Birman, and S. Maffeis. Horus, a flexible group communication system. *Communications of the ACM*, April 1996.

[20] W. Vogels, D. Dumitriu, K. Birman, R. Gamache, R. Short, J. Vert, M. Massa, J. Barrera, and J. Gray. The design and architecture of the microsoft cluster service – a practical approach to high-availability and scalability. In *Symposium on Fault-Tolerant Computing*, number 28, June 1998.

[21] Y. Amir, D. Dolev, S. Kramer, and D. Malki. Transis: A Communication Sub-System for High Availability. In *FTCS conference*, July 1992.

[22] Y. Amir, G. Ateniese, D. Hasse, Y. Kim, C. Nita-Rotaru, T. Schlossnagle, J. Schultz, Jonathan Stanton, and Gene Tsudik. Secure group communication in asynchronous networks with failures: Integration and experiments. In *1999 International Conference on Distributed Computing Systems*, August 1999. In submission.