Scalable Process Group Membership for the Spread Toolkit

Ryan W. Caudy
caudy@cnds.jhu.edu

Advisor: Dr. Yair Amir
yairamir@cs.jhu.edu

A project report submitted to The Johns Hopkins University in comformity with the requirements for the degree of Master of Science in Engineering

Baltimore, Maryland

October 2004

Abstract

Group Communication Systems (GCSs) are a part of the core infrastructure of many distributed systems. By providing reliable, ordered, many-to-many messaging services, in addition to group membership services, they can enable applications such as database replication [1], distributed logging, Internet Protocol (IP) address fail-over, and distributed load balancing. However, without careful design, scalability is not an inherent component of many such systems.

This project report discusses the design choices made in one GCS, the Spread toolkit, including using a client-daemon architecture to leverage hierarchical systems, a two-layer membership protocol that translates a membership service at the daemon-level to one that pertains to process groups, and several new enhancements made as part of this work. These enhancements include the the removal of limits on the size of state exchanges, a reorganization of the way group state is stored in the Spread daemon, and a shift to the use of representatives to decrease the messaging costs of the group state exchange.

In addition, the semantics and client interface provided by Spread's process group membership protocol are discussed, including a strengthened set of semantics to provide more information about the sets of members that were virtually synchronous in past configurations.

Acknowledgements

I'd like to thank my parents for everything they have done for me. Every day I'm reminded of just how lucky I was to have loving, kind, truly supportive parents. I owe any and all success in my life to them for the values they instilled, and their continued support.

I'd like to thank my advisor, Dr. Yair Amir, for giving me incredible opportunities in the Center for Networking and Distributed Systems. I may not have always taken full advantage, but I feel that I gained an incredible amount of skill, knowledge, and experience, both from the work I did, and from my conversations with a man who has a truly excellent perspective on the value of building *real* computer systems.

I'd like to thank all of the members of the CNDS lab, past and present, who have been astounding colleagues. John Schultz, for being the best man to think and talk with about any data structures or semantics problem I came up with. Jonathan Stanton, for being the key to my productivity each of the all-too-few occasions when we were able to work together. Claudiu Danilov, for being an outstanding guy, with great advice about *anything*. Ciprian Tutu, for teaching me the first lessons about writing a real paper, and just being a great person to talk with. Cristina Nita-Rotaru, for helping me get through a few difficult times, and being a good influence. Each and every other person associated with the lab deserves my thanks, as well, for helping to make it such an excellent place to do research.

I'd like to thank some of the people who helped me make it through Hopkins financially, including Mike Bloomberg, the STRIVE Benefactor, and the Benefactor's angel-of-everything, Susan Barnwell.

Last, but not least, I'd like to thank every one of the good friends who helped me get through Hopkins, including Mike Hilsdale, Darren Davis, Raphael Schweber-Koren, Peter Keeler, Ashima Munjal, and most especially Lisa Vara-Gulmez, my fiancée.

Ryan W. Caudy
October 2004

Contents

List of Figures

1 Introduction

The past decade has witnessed extremely rapid growth of the Internet, as part of a massive increase in the availablity of networking technology, to the extent that even citizens of developing nations often have cheap access to global communication over vast networks. With this increase, distributed applications have become an increasingly common part of everyday life for a continually growing number of individuals. Taking this reality into account, scalability in distributed systems must be treated as a goal of paramount importance.

Group Communication Systems (GCSs) are a part of the core infrastructure of many distributed systems. By providing reliable, ordered, many-to-many messaging services, in addition to group membership services, they can enable applications such as database replication [1], distributed logging, Internet Protocol (IP) address fail-over, and distributed load balancing. However, without careful design, scalability is not an inherent component of many such systems.

The Spread toolkit is one such GCS, developed at the Center for Networking and Distributed Systems (CNDS) at The Johns Hopkins University. This report describes the architecture of the Spread toolkit as it pertains to providing a scalable process group membership service. It will reveal the basic design choices that enable the construction of a scalable system of this type, as well as a number of improvements that have been made as part of this work.

The remainder of this report is organized as follows. Section 2 presents an overview of the Spread toolkit. Section 3 further develops a discussion of Spread's process group membership algorithm, and the improvements that have been made. Section 4 discusses extensions to the semantics of Spread's process group membership algorithm that were enabled by modifications designed to improve scalability. Section 5 provides performance results, while Section 6 provides a limited survey of related work. Section 7 concludes the report.

2 The Spread Toolkit

The Spread toolkit, publicly available as open source software from www.spread.org, is a GCS that implements a version of the Extended Virtual Synchrony (EVS) specification [6, 8, 1]. It provides messaging and membership services with strong guarantees to client applications that link with the Spread client library. This library provides a simple, powerful Application Programming Interface (API), and performs the low-level networking operations to allow clients to communicate over a reliable, first-in-first-out channel (i.e. TCP/IP or Unix domain sockets) with a Spread daemon process. Spread daemons communicate amongst themselves using User Datagram Protocol (UDP) packets, sending multicast or broadcast messages when possible and advantageous, and unicast messages otherwise.

2.1 Layered Membership Service

A large part of the scalability of Spread's membership service stems as a direct consequence from the client-daemon architecture described above. Rather than have each client participate directly in a global membership algorithm, which would severely limit the performance of a system with many clients, the client membership service is mapped onto a lower level daemon membership service. This two-layer membership protocol is conducted entirely by the daemons.

The first layer, the daemon membership, is responsible for setting up the network level membership of Spread daemons. It copes properly with partitions and merges of the underlying network, as well as failures and recoveries of individual Spread daemons. This protocol implements exactly the EVS specification as presented in [6]. It is expected, and observed in real systems, that this protocol operates effectively with tens or even hundreds of daemons. However, scaling to thousands would be difficult in the extreme.

The second layer, the group membership protocol, has two interacting components. The first, referred to here as the lightweight membership event handler, deals with group membership events that are not caused by an underlying change in the daemon membership, but rather by the join, leave, or disconnect of an individual client. The second, the group state exchange algorithm, exchanges messages between daemons who have successfully installed a regular configuration (see Section 2.2) together, in order to establish a shared state concerning the membership of client processes in the various multicast groups used within the system.

After handling a lightweight membership event or completing a state exchange, the group membership protocol ensures that clients are notified via succinct messages of the correct membership of each of their process groups. The process group membership protocol and its scalability is the subject of Section 3 of this report.

2.2 Extended Virtual Synchrony Overview

A complete specification of EVS would be well beyond the scope of this report. However, a partial, informal specification is included here, to clarify some of the discussion that follows in the remaining sections of this report.

As it pertains to this work, the vital component of the EVS specification is the notion of *configurations* and configuration changes. A *configuration* is a set of nodes that represents the membership of a network component, with a unique identifier associated that isolates it temporally from other configurations with identical node-sets. In the context of Spread, note that these identifiers are referred to as membership IDs when discussing the daemon-level EVS protocol, and group IDs when discussing the client-level EVS protocol.

Algorithm participants (nodes) deliver membership configurations of two types, *regular* and *transitional*. A regular configuration c may be preceded by several transitional configurations consisting of nodes who came together to c, and succeeded by several transitional configurations consisting of nodes who either move forward together to their next regular configuration, or crash. Thus, a given node installs a regular configuration c, followed by a transitional configuration c' consisting only of members from c, followed by a new regular configuration c'', and so on as long as membership changes continue to occur. The reason for the "or crash" possibility mentioned above is a fundamental property of asynchronous systems [2].

These configurations relate to message semantics in complex ways, but only one observation is necessary in this work. Specifically, any set of nodes that starts in a given configuration c and then installs the same following configuration c' delivers the exact same set of messages during c. This property, referred to as *failure atomicity*, is the same as saying that the nodes that installed c' from c were *virtually synchronous* during configuration c.

3 Towards a Scalable Process Group Membership

This section attempts to briefly describe the behavior of the process group membership protocol in Spread, and explain the design decisions that have affected its scalability.

3.1 Algorithm Specification and State Machine

A complete specification of the group membership protocol is contained in [9]. For this reason, this report only includes a brief overview of the algorithm. However, it is important to note that the following assumption holds throughout the remainder of this document:

> *The daemon membership protocol correctly handles delivery of configuration changes to the group membership protocol, and correctly delivers messages in those configurations exactly as specified in [6].*

The purpose of the group membership algorithm is to properly maintain the state of all groups in the system, collectively referred to as the $GroupsList$, and to ensure that clients receive proper notifications of transitional and regular configurations in order to correctly preserve EVS semantics.

The group membership protocol exists as a state machine, as depicted in Figure 1. The states and their basic functions are as follows:

GOP*(Groups-Operational)***:** In GOP, the algorithm is functioning as normal, during a regular configuration of the client-level EVS protocol. Light-weight membership events are handled by sending simple notification messages, installing new regular configurations with an (implicit) intervening transitional configuration of infinitesimal duration. Receipt of a transitional configuration from the daemon membership layer triggers a shift to the next state, GTRANS.

GTRANS*(Groups-Transitional)***:** In GTRANS, some groups are in a transitional configuration (i.e. those groups that were changed by the underlying configuration change). These groups receive notification of a transitional configuration, and deliver lightweight membership events as heavyweight membership events, in order to correctly preserve the client-level EVS semantics. Groups that have not changed behave exactly as in GOP, delivering simple notifications without the full EVS semantics. Receipt of a regular configuration triggers a shift to GOP, if the regular configuration has the same set of nodes as the previous transitional configuration, or else to GGATHER.

GGATHER*(Groups-Gather)***:** In GGATHER, client messages are blocked, preventing the arrival of any new lightweight membership events. At the beginning of this state, dae-

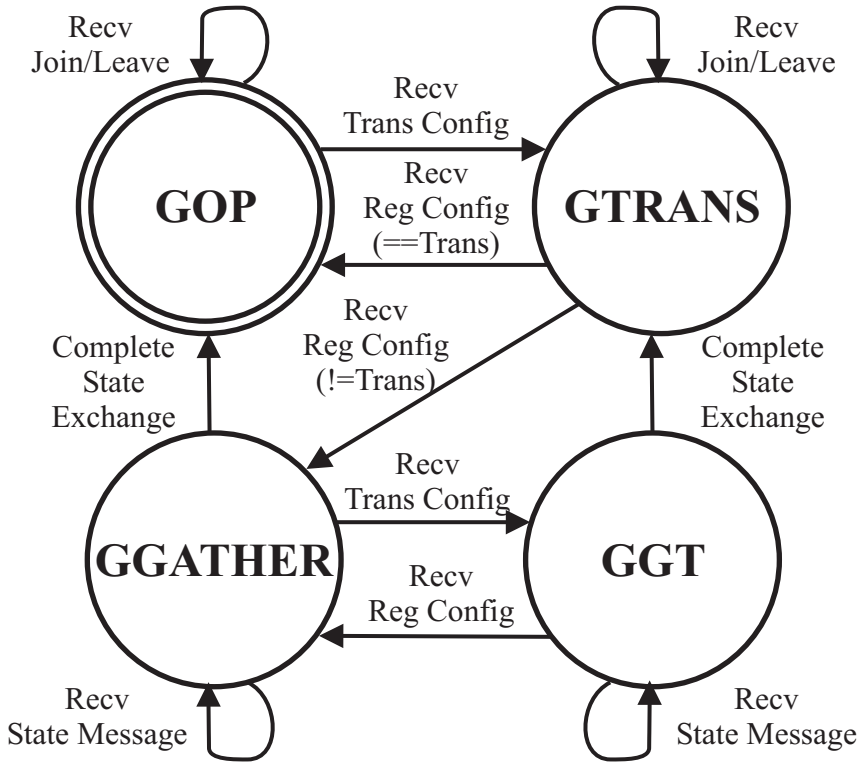

Figure 1: The Group Membership Algorithm – State Machine

mons send their $GroupsList$ state. If the state exchange is completed, the daemons compute the new $GroupsList$ deterministically, and provide notification messages to clients for any group whose membership changed either as a result of the transitional configuration or the new regular configuration. Once these notifications are completed, the algorithm returns to GOP. However, if another transitional configuration arrives before the completion of the state exchange, the algorithm shifts to state GGT.

GGT*(Groups-Gather-Transitional)***:** GGT is designed to handle cascading membership changes, that is, membership changes at the daemon level that occur before the group state is correctly exchanged after the last daemon membership change. During GGT, as in GGATHER, client events cannot occur, including lightweight membership events. A daemon in state GGT will wait for state exchange messages from the previous regular configuration. If all of the expected state messages arrive, the daemon will act as if GGATHER had in fact completed (performing the same computation, notification, and shift to GOP) and the cascading transitional configuration had in fact arrived after this point (causing a shift to GTRANS). If, instead, the next regular configuration arrives first, the algorithm will properly dispose of the old state messages it collected, shift to GGATHER, and begin the state exchange anew.

3.2 Initial Process Group Membership Implementation

Initially, the implementation of the group membership algorithm held to a very straightforward interpretation of the specification given above. The $GroupsList$ data structure was built as a *skip list* of **group** data structures, each of which maintained a *skip list* of **member** data structures. Each **group** maintained state for the group as a whole, such as the current group ID, while each **member** maintained state for the member it represented, as well as enough identifying information to locate the daemon to which the member belonged.

Using skip lists does have important scalability benefits, in terms of computational time – skip lists implement the ordered dictionary abstract data type, with expected $O(\log n)$ time for search, insert, and delete operations [7]. Since skip lists are ordered, as well, iteration over the $GroupsList$ in a desirable, deterministic order at each node can be conducted in $O(n)$ time.

However, there are drawbacks to this straightforward approach, including:

- Each daemon is responsible for sending the state for all of the members attached to it, independently from other daemons.
- A given daemon can only send exactly one state message, with an upper size limit determined by the maximum message size allowed by the GCS. This has the effect of limiting scalability artificially.
- All data in the $GroupsList$ is discarded when the new group membership was computed after a successful state exchange. This potentially wastes a large amount of computational time, because the data that is discarded is that for all of the daemons that have remained virtually synchronous with the daemon under consideration since it last completed a group state exchange, and hence is *up to date*.
- Members are managed individually, even though a large percentage of their state is common to all members in the same group attached to the same daemon.

3.3 Scalabity Improvements

The work detailed in this report includes a number of scalability improvements, as well as certain semantic enhancements (see Section 4), and extensive code changes for software engineering and clarity reasons. The scalability changes are explained in detail in this subsection.

The first improvement was removal of the limit on state-exchange messages per daemon. Daemons now send as many state messages as they require, with only one limitation: for a given group, the data about members from a given daemon must fit in one message. Removing this limit requires only a few modifications to the way such messages are built. A

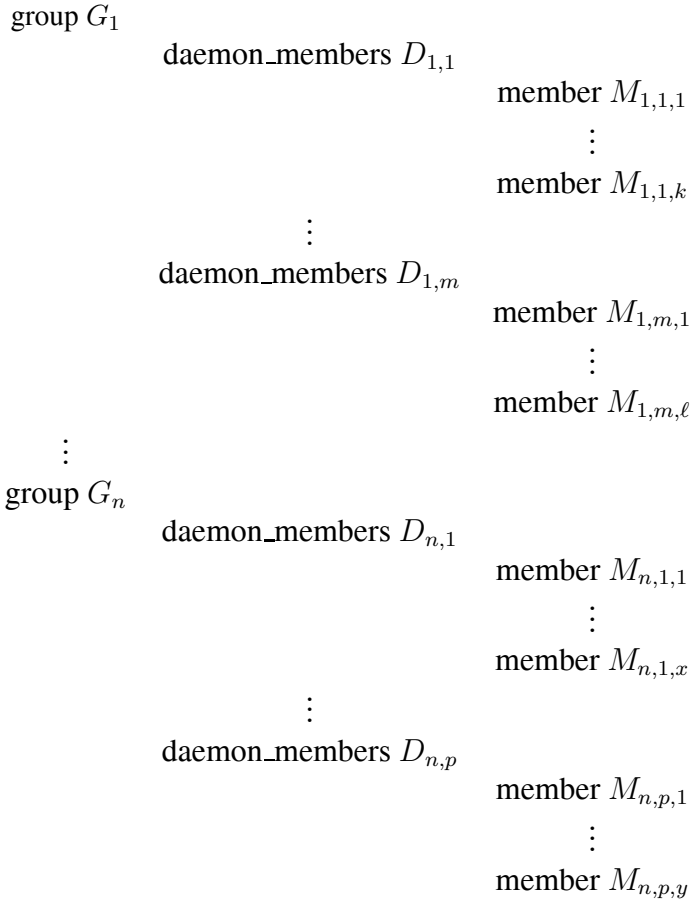

Figure 2: Three-level $GroupsList$ Hierarchy

small performance enhancement that went along with this change was to send only the last state-exchange message from a given daemon using totally-ordered delivery, and unordered for all earlier messages. Note that the requirement for the last message to be sent with an additional ordering guarantee ensures that the daemons in the system will complete the state exchange at the same place in the stream of messages being delivered during their configurations, ensuring consistent operation of the group membership algorithm.

Two key observations lead to the further changes that were made. First, when a network level change occurs, the group membership is changed not according to individual-member units, but rather in units consisting of all the members residing at a given daemon. Second, under certain circumstances, each daemon in a set is known to have exactly the same group state.

Leveraging the first observation allows for a reorganization of the $GroupsList$. Instead of using a two-level hierarchy of **group** structures and **member** structures, it's advantageous to instead use a three-level hierarchy. This new structure consists of **group** structures, which list **daemon_members** structures, which in turn list all of the **member** structures for a given daemon and group. See Figure 2 for an illustration of this reorganization.

This new level of hierarchy allows for most of the aforementioned state information to be kept *per daemon per group* instead of *per member per group*, resulting in a substantial savings in space for systems with large numbers of clients per group. In addition, various places in the group membership algorithm require iterating over all members in a group to examine this state. Considering that the number of daemons with members in a group is often substantially smaller than the total number of members, a sizable decrease in computational time used for certain parts of the algorithm is to be expected, most notably during the handling of transitional configurations that cause some daemons to become partitioned away from the daemon we consider. For a demonstration of this benefit, see Section 5.

This reorganization also makes it far simpler to take advantage of the second observation listed above. If daemons are known to have exactly the same group state information, one daemon can represent the others in the state exchange process. This condition holds when a set of daemons have completed an earlier group state exchange and have remained virtually synchronous with one another since. Thus, for one such set of synchronized daemons S, there are two operations that can change the set.

1. The delivery of a transitional configuration c, with member-set S'. In this case, $S = S \cap S'$.

2. The completion of a group state exchange with another set R, as defined by the receipt of all state messages from the representatives of each set. Note that this does not demand that the entire state exchange complete, allowing us to potentially consolidate two synchronized sets of daemons in GGT if the next regular configuration arrives before the full state exchange is completed. In this case, $S = S \cup R$.

In addition, the concept of a synchronized set of daemons, represented by only one daemon, makes it simple to avoid discarding and recomputing state information that is guaranteed to be correct. This results in significant performance and scalability improvements.

There may also be a significant savings in network time spent transferring state, since under most circumstances fewer overall messages need to be sent, saving overhead and protocol costs. However, the experiments conducted in Section 5 use very large numbers of members and groups, and so don't serve to evaluate this consideration.

4 Extending the Group Membership Semantics

One aspect of the group membership protocol that has been left out of our discussion up to this point are the semantics of the notification messages delivered to the clients. Note that clients receive notification messages on a per-group basis, and there there are two types of these *membership messages*.

4.1 Membership Message Semantics

A *transitional membership message* serves only to signal to the client that the messages it receives for a given group after the signal may not meet all of the ordinary guarantees that hold during a regular configuration (see [6]). However, this message does not provide any details about the membership of the transitional configuration it initiates. The reason for this is that that membership may not yet be known, due to the possibility that lightweight membership changes may occur during the transitional configuration. For instance, if a member m was notified that a member m' was in her transitional configuration, but then m' left the group before the next regular configuration was installed, this would violate the EVS specification as described in Section 2.2.

For this reason, the *regular membership message* that installs the following regular configuration in the client-level EVS protocol notifies the client not only of the membership of its new regular configuration, but additionally of the group members with whom that client was virtually synchronous during the last transitional configuration (provided that these members did not crash). This set of group members included with the regular membership message is referred to as a *virtual synchrony set* (or VS-set).

4.2 Extending the Virtual Synchrony Set

The changes to the structure of the $GroupsList$ enable an extension to the VS-set semantics to be made with little-or-no performance penalty or added complexity. Each daemon adds one additional field to the **daemon_members** structure, which contains the membership ID of the last daemon-level configuration in which the corresponding daemon was involved in a network-level change for the corresponding group. It is easy to see that **daemons_members** structures whose members are included in the next regular configuration and who have identical membership IDs have member sets consisting of members that were virtually synchronous with one another during the previous configuration, or crashed.

Since this information is passed to all daemons during the group state exchange algorithm, it becomes possible for each daemon to inform its members of not only their own VS-set, but also of additional disjoint VS-sets that cover every member of the new regular configuration. This extra level of information is referred to colloquially as, “who-came-

with-whom?" Some discussion has been made of attempting to leverage these extended semantics in group re-keying algorithms for secure group communication systems.

Note also that these additional semantics are well-complemented by the addition of representatives to the state exchange. Since client-level messages are not delivered during the execution of the state exchange algorithm, even a representative whose synchronized set grows as the result of a partially completed state exchange in GGT can accurately maintain the multiple VS-sets by simply not changing membership IDs, since doing so is unnecessary.

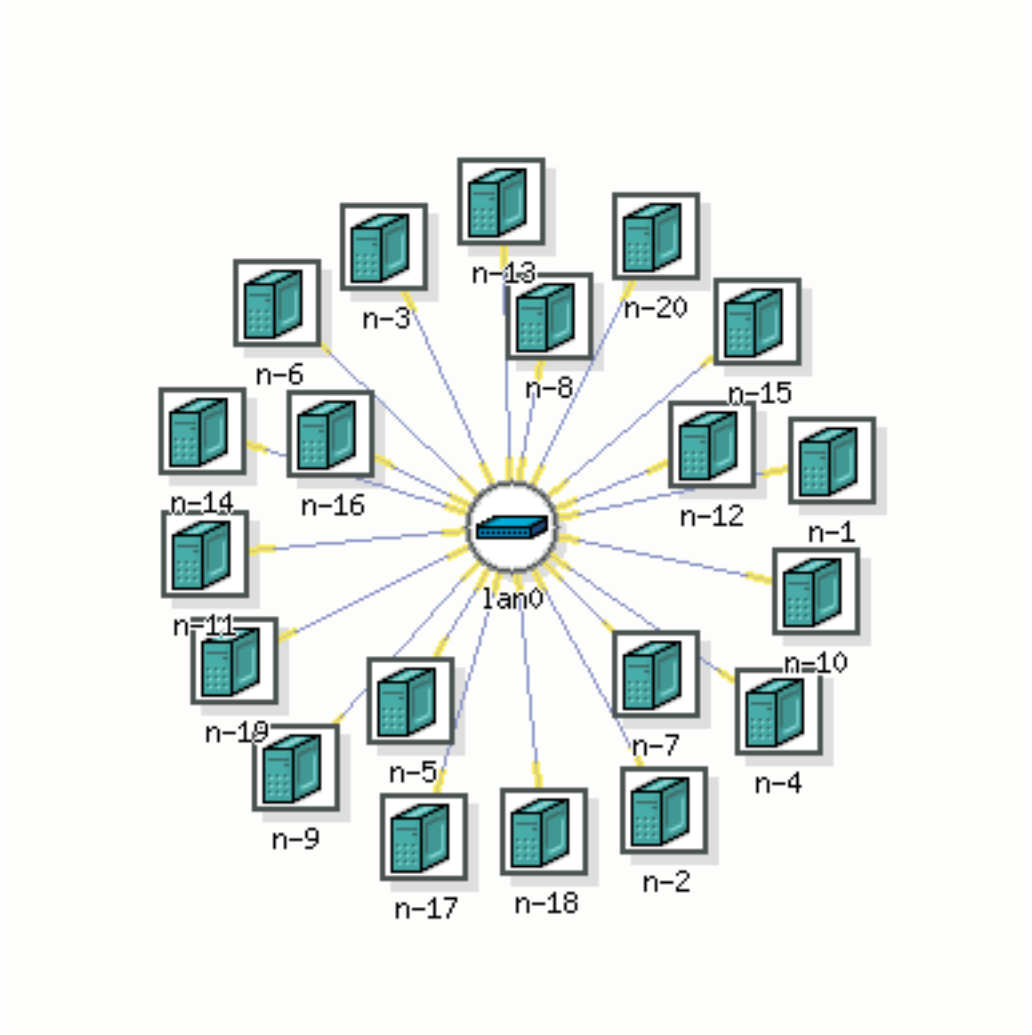

Figure 3: Emulab Visualization of Experimental Topology

5 Performance

5.1 Experimental Design

The experiments for this section were conducted using the Emulab Network Emulation Testbed[10] at the University of Utah. The network used was a single 100 Mbps LAN (Local Area Network), connecting twenty of the testbed's computers. Each node ran Redhat Linux 9.0, and had one 850MHz Intel Pentium III processor and 512MB of PC133 ECC SDRAM.

Each node had one Spread daemon, and 50 test clients, each of which joined a specificed number of groups, 100, 500, or 1000 in the tests conducted. This means that in each of these tests, each group has 1000 members, which leads to $100,000$, $500,000$, or $1,000,000$

group members in the system as a whole, respectively. One additional test director client was run on the last node, node 20, in order to trigger each measured action. Figure 3 shows the network topology as displayed via Emulab's NetBuild GUI visualization tool.

Since the goal of these experiments was a comparison between the older Spread daemon and the new one resulting from this project, each test was repeated (where possible) with each. Note that the older Spread daemon used has modifications to allow multiple state exchange messages to be sent, including the delivery guarantee enhancement discussed in Section 3, in order to allow comparisons with worthwhile numbers of members and groups. Unless otherwise discussed, each data point represents the average of twenty measurements.

5.2 Lightweight Membership Changes

Lightweight membership events were triggered using the test director client, which joined or left as the last member of the last group on the last daemon, according to the sorting used by the GroupsList data structure. The measurement results for the computation time to insert this new group member and create the correct notification message are shown in figure 4 and figure 5, comparing each daemon side-by-side.

The general trend observable from the graphs is that the time increases along a relatively flat line for each version of the Spread daemon, but that the newer daemon has somewhat better performance. It seems likely that this performance difference is due to the hierarchical organization of the GroupsList, which allows the new daemon to not rely entirely on the expected distribution of nodes in the skiplist for its time cost.

5.3 Heavyweight Membership Changes

Heavyweight membership events, i.e. partitions and merges of the set of Spread daemons that can communicate via the network, were triggered using administrative and testing tools created for use with Spread, specifically SpMonitor and SpCmd. They allow a different connectivity than that of the actual network to be imposed artificially on the Spread daemons.

The numbers reported for each of the following experiments are from the daemons that serve as representatives for each membership change in the new version of the algorithm. Total time costs for non-representative daemons are very similar, but only the representatives create state exchange messages, the time for which can be compared to the time the older daemons take to build their own purely-local state messages.

In general, it is clear from examination of the data that the new algorithm and implementation appears to scale linearly with the total number of members across all groups, and

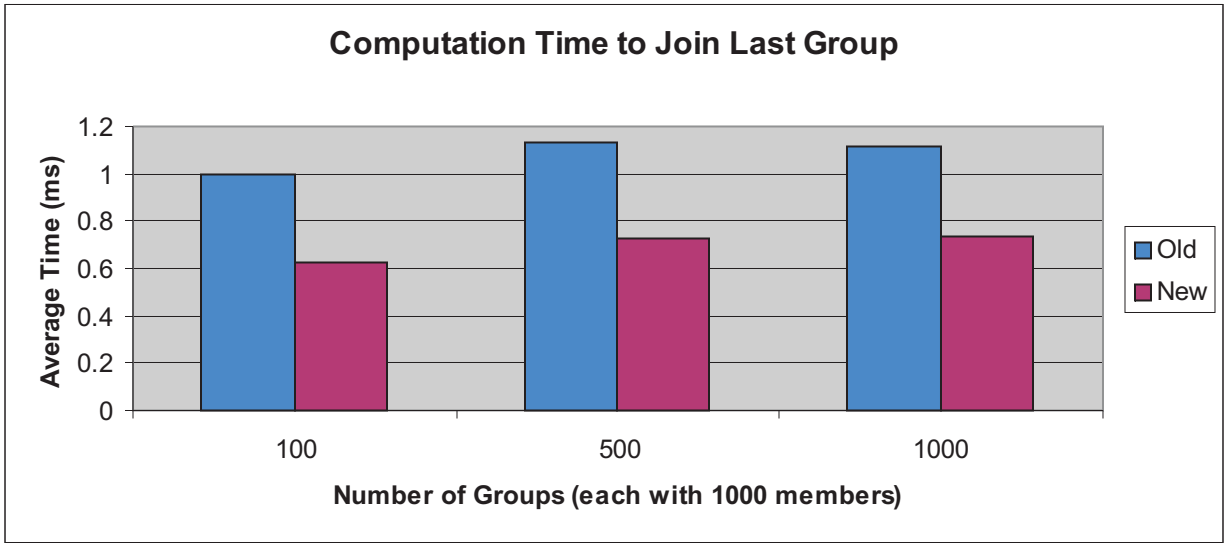

Figure 4: Computation Time to Join Last Group

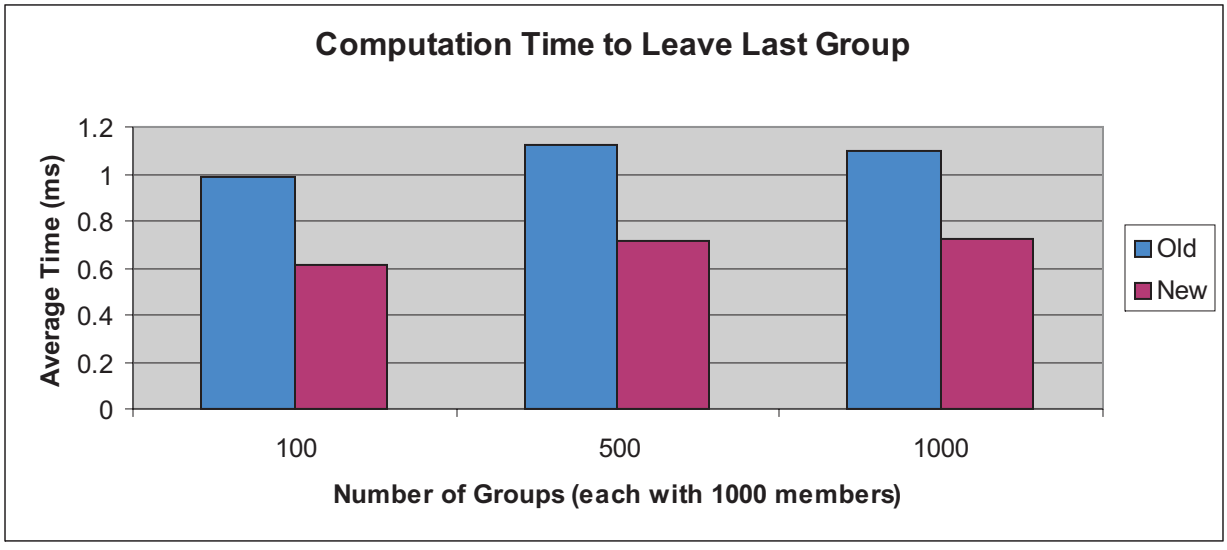

Figure 5: Computation Time to Leave Last Group

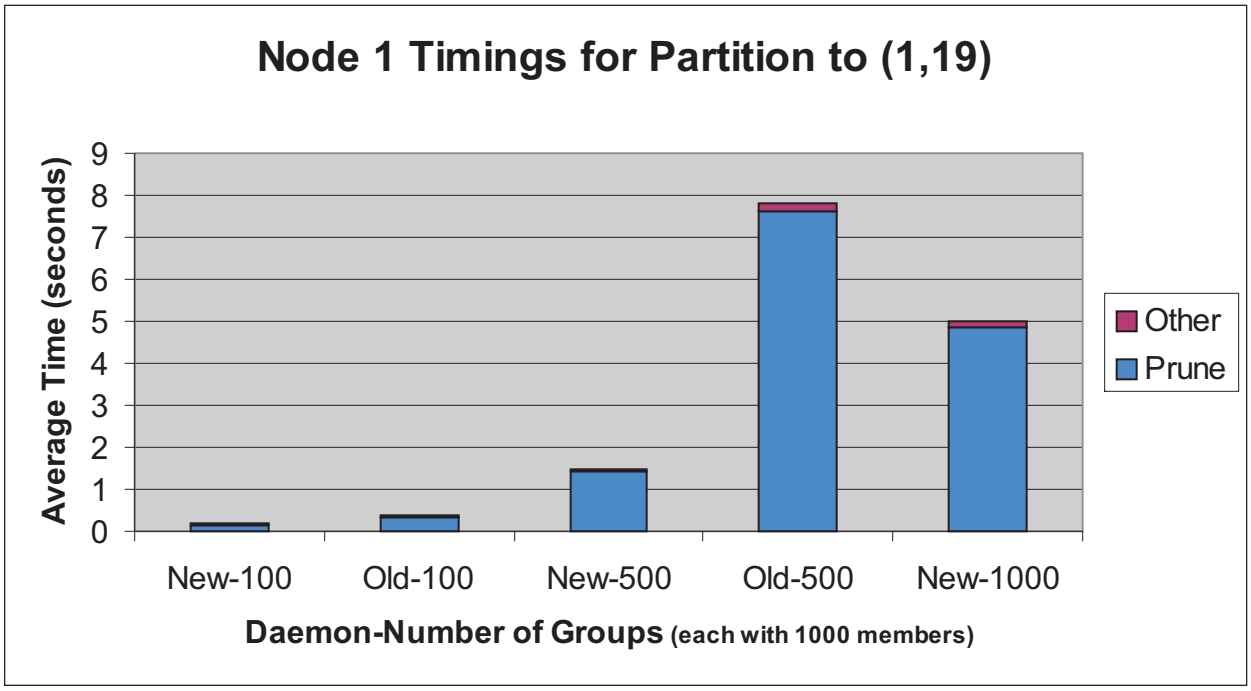

Figure 6: Partioning Costs at Representative of Singleton Partition

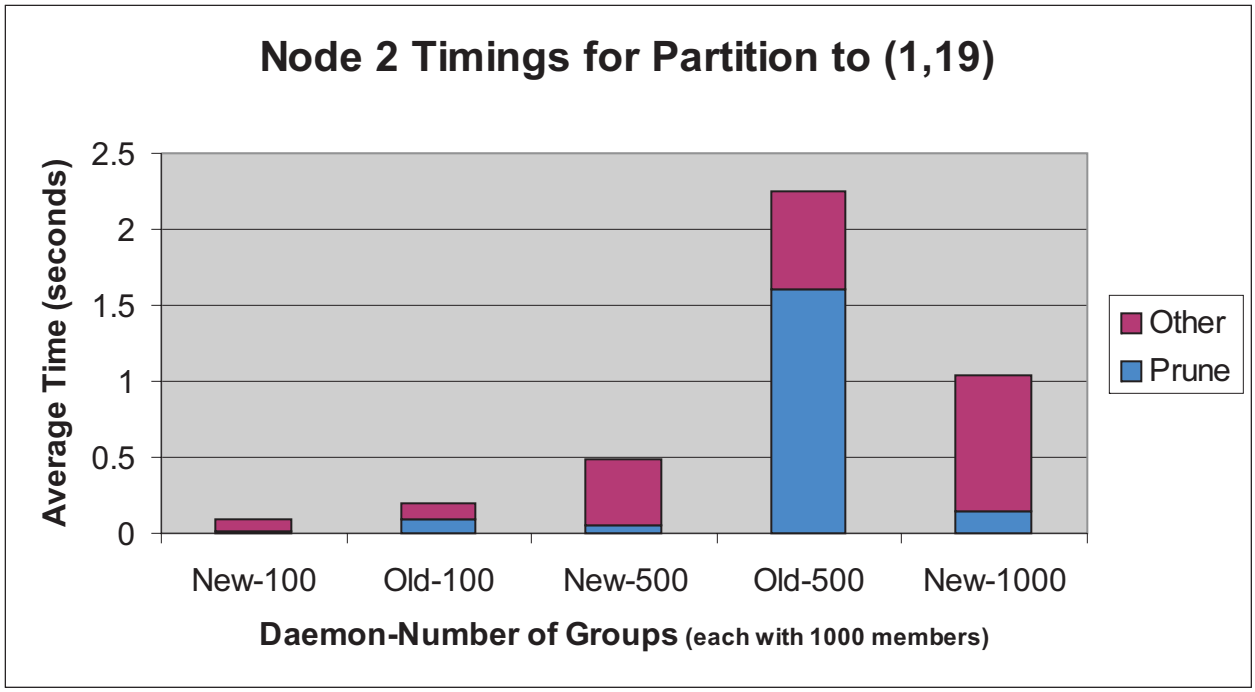

Figure 7: Partitoning Costs at Representative of Remaining Partition

impose very limited computational overhead. This does not appear to be the case with the older daemon.

Note that in order to conduct tests with the older daemon at 500 groups without triggering looping, cascading heavyweight membership changes, a modification was required to increase the token timeout, a key component of the daemon membership algorithm's failure detection, up to 30 seconds. This value is extremely high for most applications, because it allows a failed daemon to stall the entire Spread network for a long period of time. For similar reasons, tests were not conducted with 1,000 groups (i.e. 1,000,000 total group-memberships) for the older daemon. It was observed that the computation time for a merge grew to be approximately 80 seconds at node 2 and the others in its partition, which is unreasonably long to set the token timeout.

5.3.1 Partitions of One Daemon

Experimental results for partitioning the daemons into one set with only daemon $\{1\}$ and another with daemons $\{2, ..., 20\}$ are shown in figures 6 and 7. At node 1, the total times for the partition algorithm were 0.17s for 100 groups, 1.49s for 500 groups, and 5.00s for 1,000 groups with the new implementation; and 0.37s for 100 groups, and 7.82s for 500 groups with the old implementation. At node 2, the total times for the partition algorithm were 0.09s for 100 groups, 0.49s for 500 groups, and 1.04s for 1,000 groups with the new implementation; and 0.20s for 100 groups, and 2.25s for 500 groups with the old implementation. As mentioned above, each group has 1,000 members.

Node 1 must remove all but its own members from the GroupsList, and so spends a fair amount of time removing other daemons' members, as shown by the dominance of prune time in figure 6. However, it is notable that the older Spread daemon spends significantly more time in doing so at each number of groups for which a comparison is possible. This is likely due to the difference in the number of membership checks required – one per group for each of the 20 daemons for the new implementation, and one per group for each of the 1,000 members for the old implementation.

Node 2 has significantly lower costs to prune the GroupsList, since it must only remove $\frac{1}{20}$ of the total members. For this reason, the total cost is dominated more by the remaining time, which is spent mostly on building membership notification messages.

5.3.2 Merges of One Daemon

Experimental results for re-merging the partitioned sets described above are shown in figures 8 and 9. At node 1, the total times for the merge algorithm were 0.92s for 100 groups, 4.69s for 500 groups, and 9.30s for 1,000 groups with the new implementation; and 1.25s for 100 groups, and 16.05s for 500 groups with the old implementation. At node 2, the

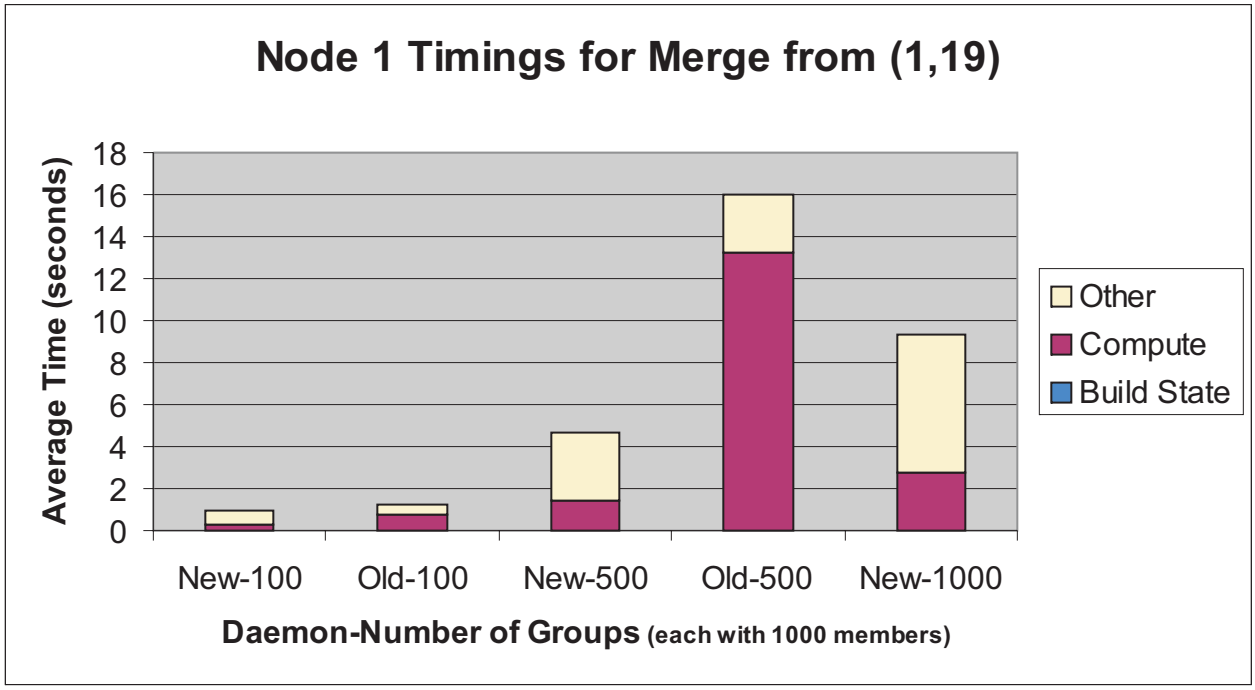

Figure 8: Merge Costs at Representative of Singleton Partition

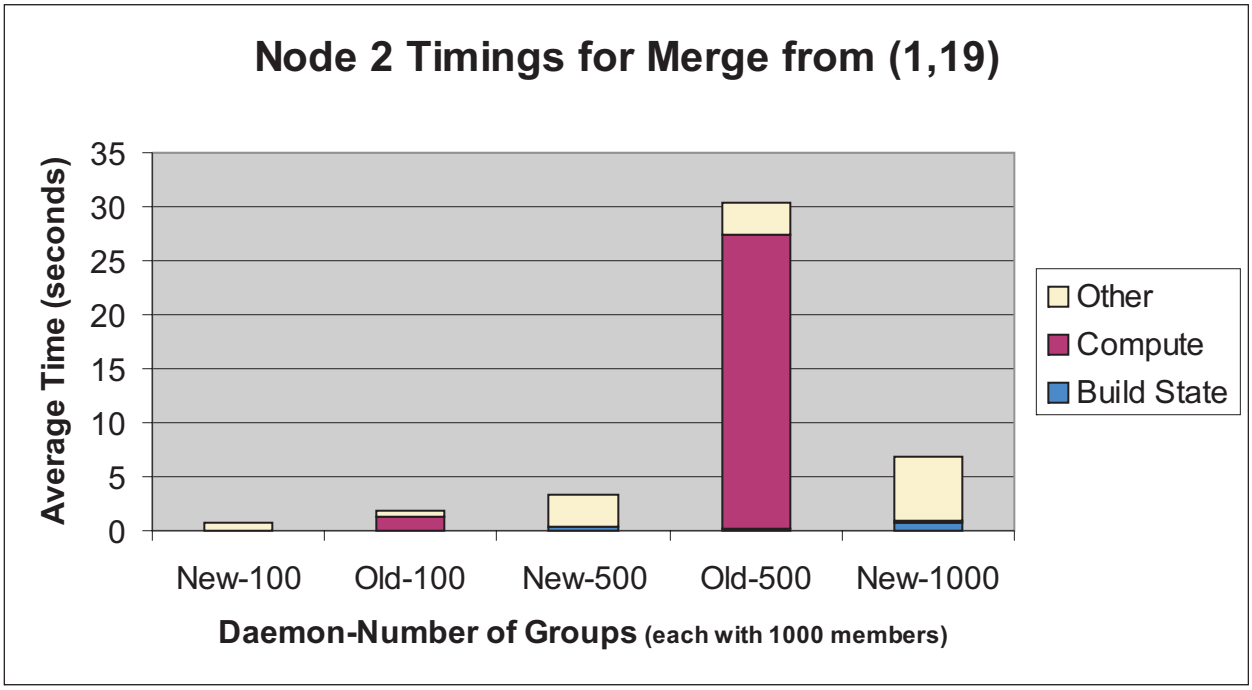

Figure 9: Merge Costs at Representative of Remaining Partition

total times for the merge algorithm were 0.67s for 100 groups, 3.39s for 500 groups, and 6.83s for 1,000 groups with the new implementation; and 1.86s for 100 groups, and 30.33s for 500 groups with the old implementation. As mentioned above, each group has 1,000 members.

Here, the easiest observation is the much higher computation cost paid by the older daemon when rebuilding the GroupsList. This is due to its naive policy of removing *all* members from each changed group, rather than only those that are not known to have correct state. This cost seems higher than it should be, but has been observed in previous performance tests when using the same skip list implementation.

The time spent building state messages is much greater for the new daemon than for the old daemon at these representative nodes. However, this difference is minimized wherever possible, and appears to have negligible impact when compared with the total time cost of the algorithm. Note that node 1 pays a much higher computation cost than node 2, for the new daemon, because it must do 19 times more work in processing messages and inserting members to its GroupsList. The "other" time category is spent mostly on building notification messages, and on the network transmission of the state messages, which have roughly the same total size for the two daemon implementations, but with lower metadata overhead for the new daemon.

5.3.3 Partitions into Two Equal Sets

Experimental results for partitioning the daemons into two equally-sized sets, $\{1, ..., 10\}$ and $\{11, ..., 20\}$, are shown in figures 10 and 11. At node 1, the total times for the partition algorithm were 0.11s for 100 groups, 1.18s for 500 groups, and 2.20s for 1,000 groups with the new implementation; and 0.45s for 100 groups, and 13.61s for 500 groups with the old implementation. At node 11, the total times for the partition algorithm were 0.11s for 100 groups, 0.82s for 500 groups, and 1.70s for 1,000 groups with the new implementation; and 0.53s for 100 groups, and 17.49s for 500 groups with the old implementation. As mentioned above, each group has 1,000 members.

Here, the most noticeable measurement is the extremely high cost for pruning the GroupsList for the old daemon with 500 groups. As in the measurements at node 1 for the singleton partitioning, the time to prune the GroupsList dominates, but less so in this case because of the equal division of the state information. It is not immediately clear why the pruning is slightly more expensive at node 1 than node 11 for the new implementation, but the reverse is true for the old implementation.

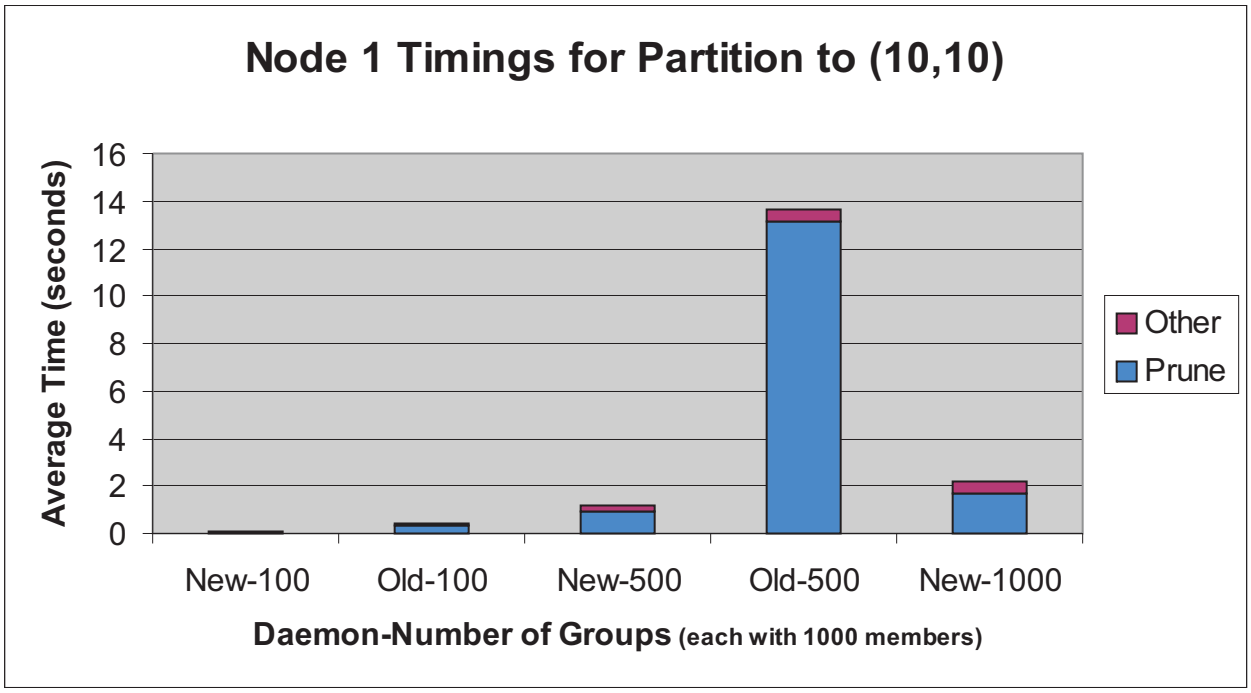

Figure 10: Partition Costs at the Representative of the First Equal Partition

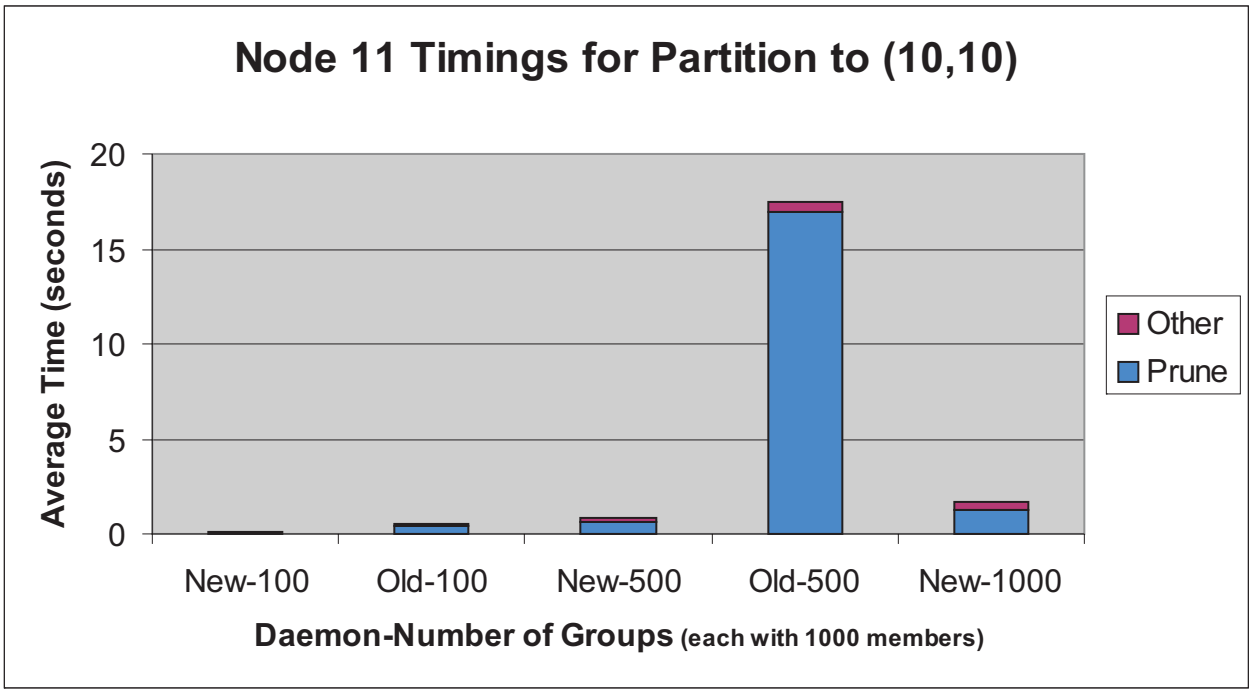

Figure 11: Partition Costs at the Representative of the Second Equal Partition

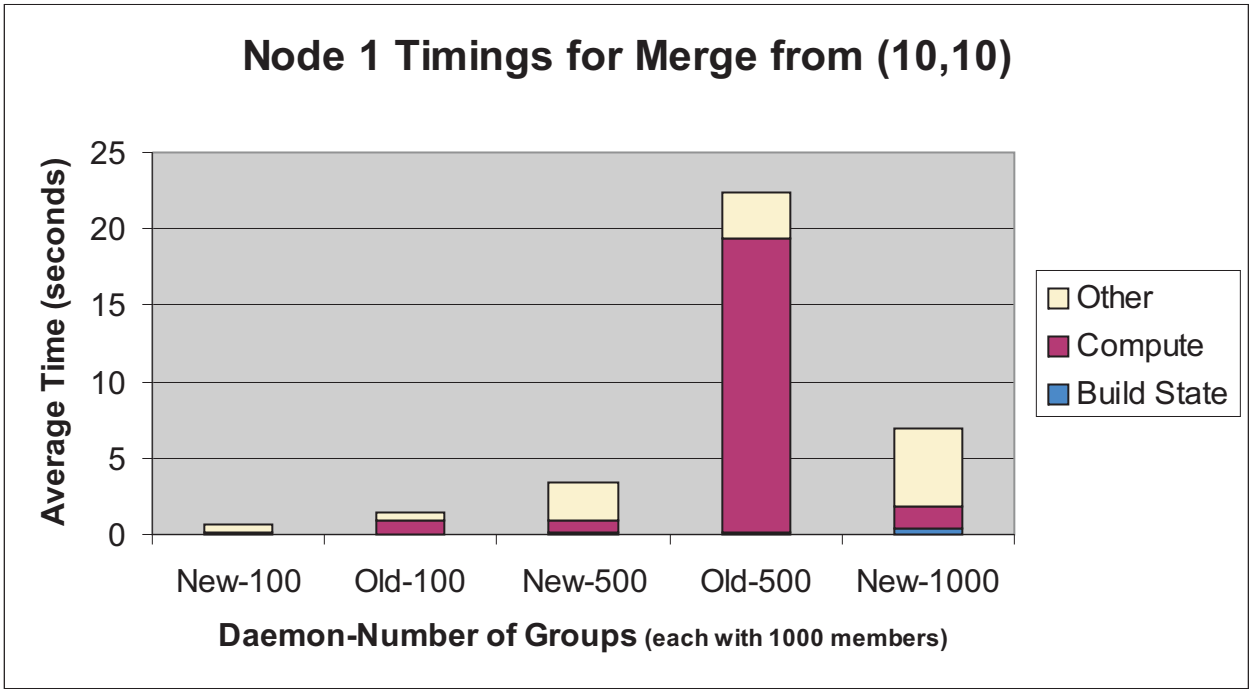

Figure 12: Merge Costs at the Representative of the First Equal Partition

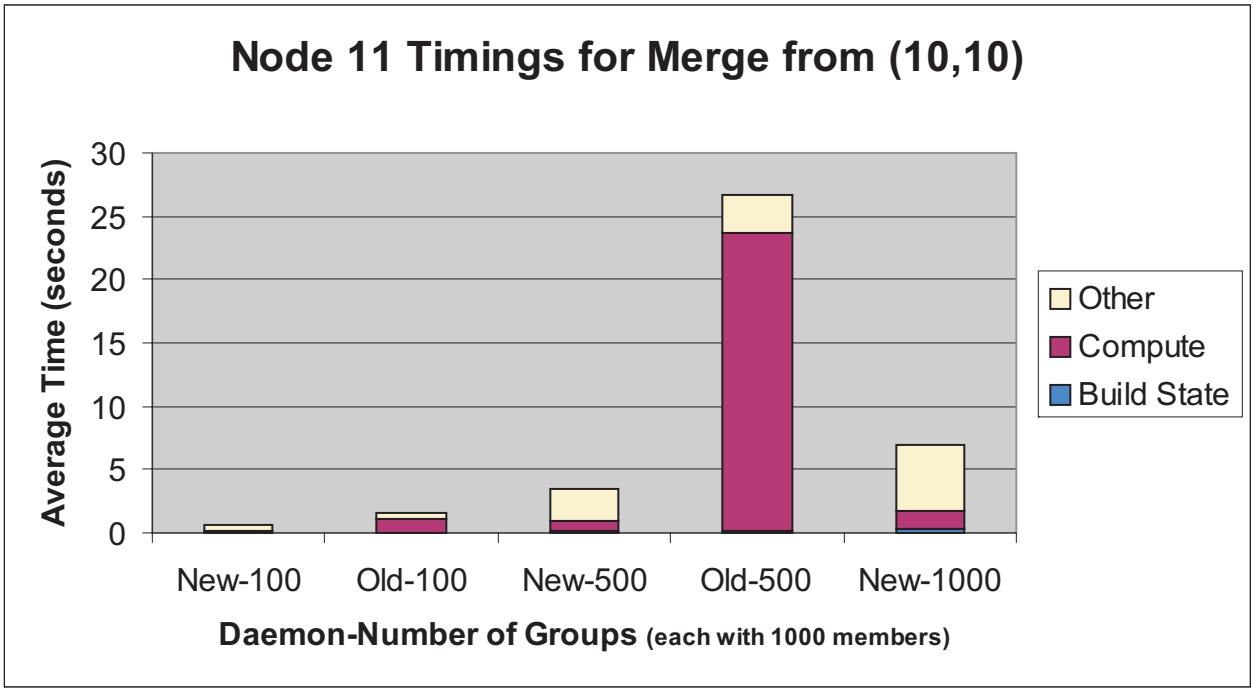

Figure 13: Merge Costs at the Representative of the Second Equal Partition

5.3.4 Merges from Two Equal Partitions

Experimental results for merging the above partitioning are shown in figures 12 and 13. At node 1, the total times for the merge algorithm were 0.69s for 100 groups, 3.45s for 500 groups, and 6.91s for 1,000 groups with the new implementation; and 1.46s for 100 groups, and 22.44s for 500 groups with the old implementation. At node 11, the total times for the merge algorithm were 0.68s for 100 groups, 3.44s for 500 groups, and 6.89s for 1,000 groups with the new implementation; and 1.63s for 100 groups, and 26.76s for 500 groups with the old implementation. As mentioned above, each group has 1,000 members. Again, the old implementation's computational cost is the dominating impression.

For the new implementation, the results are almost identical between node 1 and node 11, the leaders of their respective partitions. The performance for the new implementation shows the same general trends discussed earlier. It is worth noting that network transmission time dominates computation time, even on the hardware used, which is fairly slow by today's standards.

6 Related Work

One notable body of related work in process group membership algorithms for group communication is *Light-Weight Groups* [3, 4]. The work described in [3] attempts to provide a scalable membership service by dynamically mapping the membership protocol for several *light-weight groups* (LWGs) onto the membership of an underlying *heavy-weight group* (HWG). This allows a number of LWGs to conduct their membership protocols based on the underlying membership and messaging services, ammortizing the cost of the network-level membership. [4] introduces a version of this service that can operate in a partitionable environment.

The LWG and HWG services map roughly onto Spread's group membership and daemon membership protocols, respectively. However, there are two significant differences. Firstly, the LWG/HWG service is designed for the somewhat different semantics of Virtual Synchrony – for a complete explanation of the difference between EVS and VS, see [8]. Secondly, LWGs are dynamically mapped onto their underlying HWGs – this mapping can be changed for performance reasons based on various heuristics. Spread, on the other hand, has a fixed mapping. However, Spread's fixed mapping is designed based on the assumption that the administrators who design the architecture of a Spread network and the programmers who choose the mapping from clients to their associated Spread daemons take network and resource considerations into account. Thus, without any need for dynamically changing the association between the two levels of the membership protocol, it is possible to achieve a near-optimal mapping for a given environment. For this reason, however, it may require a great deal of intelligent management to properly configure and maintain a Spread network.

Moshe [5], a group membership service designed for wide area networks, seeks to specify a general algorithm that is portable across different underlying failure detection services and independent of the system using the membership data. Potentially, the view-oriented membership data that is delivered will be used by a GCS as a lower layer in providing its own services. The process-level membership provided by *Moshe* and its underlying failure detection service maps fairly closely onto Spread's daemon membership service. The processes provided with membership views by *Moshe* are directly engaged as nodes in a GCS, while Spread enables an additional level of hierarchy that enables it to potentially scale to many more client processes. In addition, Spread's tight coupling of failure detection, membership algorithm, and group communication semantics allow for better performance decisions to be made throughout the system.

Also relevant are a number of works that provide information about Spread's implementation, semantics, and/or performance, notably including [8] and [9].

7 Conclusions

This project report described the key elements of the scalability of process group membership in the Spread toolkit, a Group Communication System. It conducted a careful examination of the role of a client-daemon architecture in enabling this scalability, and also provided detailed explanations of several performance improvements to Spread's process group membership protocol. In addition, the semantics of the membership protocol and its notification messages were examined in detail, including several enhancements performed as part of this work.

Moderately comprehensive performance measurements of the scalability of both the original and the enhanced process group membership protocols were conducted using the Emulab[10] testbed. The discussion of these results emphasized that the enhanced protocol scales linearly with the number of group membership relationships, and minimizes computation through careful application of shared knowledge.

Strong performance increases with large numbers of groups and members were shown with respect to the previously-existing group membership algorithm and implementation. In tests with 20 daemons and 500 groups, with 500,000 total members across all groups, the group membership algorithm's average total times to partition the set of daemons in half and re-merge from this partitioning were decreased by at least 91% and 85%, respectively. For partitioning of a single daemon and the corresponding re-merge, the average total times were decreased 71% and 81%, respectively.

References

[1] Yair Amir. *Replication Using Group Communication Over a Partitioned Network*. PhD thesis, Hebrew University of Jerusalem, 1995.

[2] Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. Impossibility of distributed consensus with one faulty process. *J. ACM*, 32(2):374–382, 1985.

[3] Katherine Guo and Luis Rodrigues. Dynamic light-weight groups. In *The 17th IEEE International Conference on Distributed Computing Systems (ICDCS)*, pages 33–42, 1997.

[4] Katherine Guo and Luis Rodrigues. Partitionable light-weight groups. In *The 20th IEEE International Conference on Distributed Computing Systems (ICDCS)*, 2000.

[5] I. Keidar. Moshe: A group membership service for wans, 1999.

[6] Louise E. Moser, Yair Amir, P. Michael Melliar-Smith, and Deborah A. Agarwal. Extended virtual synchrony. In *The 14th IEEE International Conference on Distributed Computing Systems (ICDCS)*, pages 56–65, 1994.

[7] William Pugh. Skip lists: A probabilistic alternative to balanced trees. In *Workshop on Algorithms and Data Structures*, pages 437–449, 1989.

[8] John Schultz. Partionable virtual synchrony using extended virtual synchrony. Master's thesis, The Johns Hopkins University, 2000.

[9] Jonathan Stanton. *Practical Wide-Area Group Communication*. PhD thesis, The Johns Hopkins University, 2002.

[10] Brian White, Jay Lepreau, Leigh Stoller, Robert Ricci, Shashi Guruprasad, Mac Newbold, Mike Hibler, Chad Barb, and Abhijeet Joglekar. An integrated experimental environment for distributed systems and networks. In *Proc. of the Fifth Symposium on Operating Systems Design and Implementation*, pages 255–270, Boston, MA, Dec. 2002. USENIX Association.