Towards Implementing Group Membership in Dynamic Networks: A Performance Evaluation Study

by

Sophia Yuditskaya

S.B. Electrical Engineering and Computer Science
Massachusetts Institute of Technology, 2002

Submitted to the Department of
Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Engineering in Electrical Engineering and Computer Science
at the
MASSACHUSETTS INSTITUTE OF TECHNOLOGY

May 2005

Author .....
Sophia Yuditskaya
Department of Electrical Engineering and Computer Science
May 19, 2005

Certified by... .....
Dr. Roger Khazan
Research Scientist, Information Systems Technology, Lincoln Laboratory
Thesis Supervisor

Certified by.. .....
Dr. Clifford Weinstein
oln Laboratory
esis Supervisor

Accepted by.....(.....
Arthur C. Smith
Professor of Electrical Engineering
Chairman, Department Committee on Graduate Students

Towards Implementing Group Membership in Dynamic Networks: A Performance Evaluation Study

by

Sophia Yuditskaya

Submitted to the Department of Electrical Engineering and Computer Science
on May 19, 2005 in Partial Fulfillment of the
Requirements for the Degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

Support for dynamic groups is an integral part of the U.S. Department of Defense's vision of Network-Centric Operations. Group membership (GM) serves as the foundation of many group-oriented systems; its fundamental role in applications such as reliable group multicast, group key management, data replication, and distributed collaboration, makes optimization of its efficiency important. The impact of GM's performance is amplified in dynamic, failure-prone environments with intermittent connectivity and limited bandwidth, such as those that host military on the move operations.

A recent theoretical result has proposed a novel GM algorithm, called Sigma, which solves the Group Membership problem within a single round of message exchange. In contrast, all other GM algorithms require more rounds in the worst case. Sigma's breakthrough design both makes and handles tradeoffs between fast agreement and possible transient disagreement, raising the question: how efficiently and accurately does Sigma perform in practice?

We answer this question by implementing and studying Sigma in simulation, as well as two leading GM algorithms – Moshe and Ensemble – in a comparative performance analysis. Among the variants of Sigma that we study is Leader-Based Sigma, which we design as a more scalable alternative. We also discuss parameters enabling Sigma's optimal practical deployment in a variety of applications and environments.

Our simulations show that, consistently with theoretical results, Sigma always terminates within a single round of message exchange, faster than Moshe and Ensemble. Moreover, Sigma has less message overhead and produces virtually the same quality of views as Moshe and Ensemble, when used with a filter for limiting disagreement. These results strongly indicate that Sigma is not just a theoretical result, but indeed a result with important practical implications for Group Communication Systems: the efficiency of GM applications can be significantly improved, without compromising accuracy, by replacing current GM algorithms with Sigma.

Thesis Supervisor: Dr. Clifford Weinstein
Title: Group Leader, Information Systems Technology, Lincoln Laboratory

Thesis Supervisor: Dr. Roger Khazan
Title: Research Scientist, Information Systems Technology, Lincoln Laboratory

Acknowledgements

I would like to express my heartfelt thanks to my thesis supervisor Dr. Roger Khazan for the dedicated guidance and valuable advice that he has given me. He has taught me so much about the field of group communications, as well as the research process, over the course of my thesis work.

I am deeply grateful to my thesis supervisors Dr. Clifford Weinstein and Dr. Roger Khazan for welcoming me into MIT Lincoln Laboratory as a research assistant and for giving me the opportunity to work on this project for my thesis. This research was sponsored by the United States Air Force under Air Force Contract F19628-00-C-0002. (Opinions, interpretations, conclusions, and recommendations are not necessarily endorsed by the US Government.) I am deeply grateful to everyone who vouched for my character and abilities: special thanks to my academic advisor Professor Arthur C. Smith. I am deeply grateful to Steve Serra, Siva Ravada, Jeffrey Xie, Terry Xu, and Zhihai Zhang for their understanding and encouragement while I have pursued my degree.

I would like to thank my friends Debbie and Harvey Furey, and Rebecca S. Bloom, for their enthusiastic moral support. Thanks to my fellow Lincoln Lab research assistants Barry, Lisa, Michael, Daryush, and Nick, for their congeniality. Thanks also to my friends Nicole Alexander, Manjari Yalavarthy, and Andrew Lamb, for their gracious words of encouragement.

Finally, I would like to give special thanks to my family for their everlasting, steadfast love and support.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

Dynamic groups are an integral part of the DoD's vision of a Network-Centric Global Information Grid, enabling such important military capabilities as Collaborative Teams, Communities of Interests, and Command-and-Control on the move operations.

Group Membership (GM) is one of the fundamental services required for supporting dynamic groups, their applications, and services. Group-oriented applications are typically blocked while GM handles membership changes. As a result, the performance of such higher-level applications directly depends on the performance of GM – specifically, how fast GM is able to handle membership changes, and how frequently these changes occur. The impact of GM's performance on these applications is amplified in dynamic, failure-prone environments with intermittent connectivity and limited bandwidth, such as those that host military on the move operations.

This thesis is motivated by Khazan's recent *theoretical* result that solves the problem of GM simpler and more efficiently than previously believed [39]. This solution consists of two parts: a) a GM algorithm, called *Sigma*, which handles membership changes faster than previous algorithms, but possibly with lesser accuracy; and b) a mechanism for improving accuracy, which works in conjunction with Sigma.

The goal of this thesis is to evaluate how well Khazan's theoretical algorithm performs in practice, particularly in dynamic, failure-prone environments. Based on this evaluation, we also hope to provide insight into options for optimizing Sigma and formulate general guidelines for deploying GM in dynamic environments.

To achieve these goals, we implement and study several variants of Sigma, in comparison with two other GM algorithms. One of the variants of Sigma that we study is a new algorithm that we have designed as part of this thesis. This new algorithm improves Sigma's scalability by transforming its all-to-all type of communication into a leader-based version [37]. The two GM algorithms to which we compare Sigma are a) an optimistic all-to-all algorithm, called Moshe [34], and b) a leader-based algorithm deployed in the Horus and Ensemble group communication systems [19, 27]. We

simulate these algorithms in a dynamic environment driven by connectivity traces that have been collected from a wide-area network (WAN) [9, 29].

The results of our study indicate that Sigma is both practical and efficient [53]. Consistently with the theoretical results, Sigma uses only a single round of message exchange to handle membership changes, faster than Moshe and Ensemble. Moreover, Sigma has smaller message overhead and is virtually as accurate as the other two, particularly when used with a mechanism such as the one mentioned in [39] for improving Sigma's accuracy.

Our results also yield a general observation that, in dynamic environments, GM is practical only for those applications and services that do not require GM to respond to membership changes immediately as they are detected. These are the applications that are able, and in fact prefer, to delay GM responses for some time in order to ignore short transient disconnects and avoid useless, frequent view changes and the overhead associated with them. We reference some applications that fit this category in Chapter 2.

The remainder of this chapter is organized as follows. In Section 1.1 we describe the motivation behind our research. Section 1.2 provides an overview of the background relevant to this thesis, from the field of reliable distributed computing, to the fault-tolerance benefits of group communication and group membership, and finally a summary of Sigma. Section 1.3 summarizes the contributions and results of this thesis. A roadmap for the rest of this thesis is provided in Section 1.4.

1.1 Motivation

Collaborative, secure, timely, and reliable communication is essential to the success of national defense operations. Towards achieving these goals, the vision of fully integrating defense forces with Network-Centric Operations (NCO) has been articulated as a high priority by the U.S. Department of Defense. In the Joint C4 Campaign Plan of September, 2004, the Joint Chiefs of Staff emphasize a networked force as the key to increasing operational effectiveness, "enabling dispersed forces to more efficiently communicate, maneuver, share a common operating picture and achieve the desired end-state [30]". Maj. Gen. Paul Lebras, Commander of the AIA and Joint Information Operations Center, and Deputy Commander for Information Operations for the 8th Air

Force, describes the important role that NCO has played, to a greater degree than ever before, in Operation Iraqi Freedom, reflecting a fundamental "paradigm shift" in national defense techniques: "We had multiple platforms linked into distributed architectures, all of which understood the commander's intent, and all of which swiftly pushed data forward [40]".

The need for NCO has outpaced the technological resources available to implement it in such failure-prone mission-critical environments as are characteristic of military operations. Such operations require algorithms and services for performing group communication, collaboration, computation, security, and authentication. Solutions need to be scalable, secure, and efficient. They need to be designed to operate effectively in dynamic fault-prone environments with intermittent connectivity and limited bandwidth.

Dynamic groups are an integral part of the DoD's NCO vision. When considered at a level higher than network connectivity and bandwidth, a major feature that enables NCO is group-oriented operations and activities. Examples of such activities in the sphere of national defense include:

- Military operations carried out by dynamic Command, Control, Communications, Computer, Intelligence, Surveillance, and Reconnaissance (C4ISR) teams;
- Distributed data acquisition and processing;
- Satellite-based capabilities, enabling an edge in military maneuvers, surveillance, reconnaissance, and global communications [40];
- Command and control of large, joint military operations and multi-organization cooperative design activities, which involve very large, dynamic groups [15];
- Real-time policy management, group key establishment, and re-keying technology for groups in which membership can change rapidly for a variety of operational as well as security reasons [15].

Group-oriented operations are widely applicable, not only for military purposes, but also for many other distributed applications; some examples are data and service replication, resource allocation, load-balancing, real-time collaborative computing, and air traffic control [17, 22, 35, 36, 44, 46]. Many of these applications involve mission-critical operations in dynamic, failure prone environments.

Group membership (GM) serves as the foundation of many group-oriented applications. GM is an essential component of Group Communication Systems, which offer robust, fault-tolerant solutions for mission-critical applications [51]. Group membership (GM) services maintain and report group membership as it changes due to voluntary joins and leaves, security privilege revocations, as well as involuntary failures, recoveries, partitions, and merges – network events characteristic of dynamic, failure-prone network environments. Without GM, even reliable multicast could not be possible, because messages cannot be sent without knowledge of their destinations – the group members.

As we shall see in Chapter 2, while GM (or its encapsulating GCS) is handling a change in group membership, many higher-level applications are *blocked*, waiting to receive a new *view* of the membership (e.g., [26, 36]). A view is a pair consisting of the membership set and a view identifier, on which group members must agree (see Section 1.2.3). After the application processes receive the new view, they typically *synchronize* with the other members to make sure that everyone has also received it, and to bring their states to a consistent base from which all of them can resume their normal operation. These two observations imply that performance of GM applications directly depends on a) how long the underlying GM protocol takes to form new views and b) how frequently these new views are created. The fundamental role of GM in group-oriented applications makes optimization of its efficiency important.

1.2 Background

This section overviews the context, relevance, and controversies surrounding the issues that we address. The description proceeds systematically from the general to the specific, from the field of reliable distributed computing, to group communication and group membership, and concluding with an overview of Sigma.

1.2.1 Reliable Distributed Computing

The field of reliable distributed computing has evolved in recent years to develop and study solutions for mission-critical applications in dynamic failure-prone environments.

Distributed computing refers to systems and applications in which physically or logically separate entities such as servers or processes cooperate to coordinate actions at multiple locations in a network. Having recently emerged from a specialized niche to ubiquitous use, today it is a technology that literally everyone depends on, from the Internet to air-traffic control, management of financial data, and electronic medical records [17].

As distributed computing is increasingly used for mission- and life-critical applications, reliability becomes especially important. However, reliability has not kept pace with the lightning-fast spread of distributed computing. As an engineering discipline, reliable distributed computing is still in its infancy, and is an active area of research and innovation. Reliability involves three distinct goals: (1) tolerating failures automatically, (2) guaranteeing properties such as performance or response time, and (3) offering security against intentional threats [17].

The problem of maintaining distributed consistency inevitably arises in a distributed system for which reliability is important. Concurrent processing among the multiple distinct processes in a distributed system can lead to inconsistent local state at each of the entities. Also, processes can fail, and the network connecting them can experience congestion or link failures. Networks and the distributed systems built on top of them are failure-prone environments, full of asymmetries that lead to inconsistencies of state in the distributed system.

In order to maintain consistency in a distributed system, it is important for the participating processes to be able to synchronize with each other to obtain and agree on the most accurate state. Before synchronization can be done, however, the system needs to know which processes are participating. Not only can processes fail and recover, but also any robust design must allow for dynamism in which processes can voluntarily join and leave the system.

One useful abstraction towards achieving distributed consistency is to envision the participating processes as members of a group (Figure 1-1). Distributed applications can then use this group abstraction as a black box that provides them with an accurate list, or membership, of processes that are participating in the resynchronization procedure. In this research, we delve into the technical details, design issues, and practical analysis of what happens in that black box.

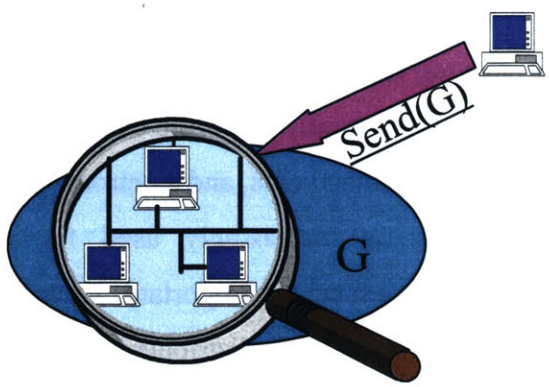

Figure 1-1. The Group Abstraction. Messages are sent to Group G, rather than to individual nodes. Image source: Idit Keidar.

1.2.2 Group Communication

Technologies that make use of the group abstraction implement a form of distributed computing called group communication. Group communication is a means for providing multi-point to multi-point communication, by organizing processes in groups [22]. Processes located at different nodes of a distributed system operate collectively as a group by using a group communication service (GCS) to multicast messages to all members of the group [26]. Isis [18], Transis [3], Totem [1, 7], Spread [2], Horus/Ensemble [19, 51], JGroups [16], and Xpand [23] are just a few of the GCS's that have been developed.

By providing such communication primitives as broadcasts to the group as a single entity, group membership changes, and migration of activity from one place to another, GCS's offer a modular solution -- a "black box" -- to reliable distributed computing. Outside the black box, these primitives appear instantaneous and atomic – enabling a distributed system to mimic the behavior of a centralized system [17]. Birman and Joseph, the designers of Isis and the first to propose and implement the idea of a GCS, write that "the major advantage of this abstraction is that many aspects of a distributed application can be treated as independent modules without compromising correctness [18]". In other words, higher level application code can be designed as if the system were centralized or synchronous, and application developers need not worry about the low-level concurrency issues of distributed systems.

True to their vision, GCSs have proven to be powerful middleware systems, facilitating development of distributed systems in dynamic, fault-prone environments by providing two distinct services: group membership and reliable multicast [39] [17] [26].

1.2.3 Group Membership

As middleware, GCSs are charged with handling, and thus buffering applications from, the effects of asynchronous failure-prone environments, in which processes can crash, reconnect, and partition. An essential component that enables the fault-tolerance capabilities of a GCS is its group membership service (GMS). The role of a GMS is to maintain the membership of a distributed system on behalf of the logically grouped processes that are currently active and connected [17]. As network events occur, such as process crashes and recoveries, and as processes voluntarily request to join or leave the group, GMS responds by delivering a view to the application that reflects the latest membership [34]. A view is a pair consisting of an identifier and a membership set. Given an accurate up-to-date view, the application then uses the GCS's reliable multicast service to deliver messages to the current view members.

The means by which a GMS forms a new view is a widely studied problem in the area of fault-tolerant distributed computing [45], and it is called the Group Membership Problem. Group Membership (GM) is the problem of maintaining a dynamic group of members and informing members about changes in the group [17, 22]. Changes in the group occur because of network events such as members joining and leaving the group, crashing, disconnecting, and reconnecting; also, the group can partition into disjoint components, and group components can later merge. The goal of GM is to provide each group member with the same, correct view of the current membership of the group. There are two parts to solving the Group Membership Problem: (1) determining the set of members that are currently connected and (2) ensuring that these members agree on the same view.

The group membership (GM) specification assumes an underlying failure detector, called a network event notification service (NS) that essentially achieves the first goal. With the membership set readily provided by the NS, the GMS is left with the problem of achieving the second goal – to ensure that all members agree on the same view of the

membership set. The purpose of the view identifier is to differentiate between two instances of the same membership set; for example, assuming no other network events, a membership set M is the same before process A leaves and after process A returns. To distinguish between the two, the view before process A leaves would be <1, M> while the view after process A returns would be <3, M>. Thus, agreeing on the same membership set also involves agreeing on the same view identifier. This goal is called achieving Agreement on Views.

Solutions to the Group Membership Problem are implemented by group membership algorithms. Most group membership algorithms have been unique to, and are closely associated with, the particular GCS in which they have been developed, such as Isis, Transis, Totem, and Spread [7, 18, 24]. However, a few GM algorithms have proven portable to some extent and therefore bear names of their own, such as Horus/Ensemble and Moshe [34, 51].

1.2.3.1 Efficiency of Group Membership Algorithms

It is important for GM algorithms to be as efficient as possible. The significance of optimizing GM algorithms comes from their critical role in group communications applications. Not only is GM the basis of all GCSs, but also GM serves as a foundation for powerful fault-tolerant services such as Totally Ordered Multicast [26], Virtually Synchronous Group Multicast [32], Intermittently Atomic Data Services [36], Group Key Agreement [5], and load-balancing replicated data [38]. Virtually all such applications have two states of operation: a normal mode, which proceeds while the network is stable, and recovery mode, triggered when a view change begins, when network instabilities cause changes in group membership. We discuss these applications in Chapter 2.

In all such applications, recovery mode represents an interruption of the application's normal operation, and it is therefore critical to minimize this reconfiguration time. Because GM plays such a central role in recovery mode, optimizing GM is an essential and potent step towards shortening reconfiguration time, improving the efficiency of these applications, and enabling them to run more smoothly in failure-prone network environments.

1.2.3.2 Membership vs. Consensus

The efficiency of GM algorithms has historically been constrained by the view that the Group Membership Problem can and should be solved as the Consensus problem, which is known to require a minimum of two rounds of message exchange[1] [33, 39, 41]. Although the two problems are known to be different, past solutions to GM have been influenced by Consensus, and therefore involve several rounds of message exchange. The most efficient among them, Moshe, is an optimistic algorithm that takes one round in the typical case, but two or more rounds in certain "out-of-sync" cases [34].

The Membership problem is weaker than the Consensus problem [22]. While Consensus requires each participant to make a single, irrevocable decision right away, Membership requires only that the correct, final decision be made eventually, once stability occurs. Stability means that there are no more network events affecting the group, or, if a partition has occurred, the group *component*. Also, the underlying network itself must remain stable, allowing group component members to communicate with each other, but with no one else. While formal definitions of the Membership problem require this stability to last forever, in practice it only has to last long enough for the "final" views to be formed.

1.2.4 Sigma

A recent theoretical result [39] is the first to propose a GM algorithm that fully harnesses the difference between Membership and Consensus. By virtue of this novel approach, the resulting algorithm, named Sigma, is guaranteed to achieve Agreement on Views within one round after the final network events affecting the group component become known to all the members. The single-round result in Sigma is achieved by decoupling and parallelizing two processes that run serially in other group membership algorithms: achieving Agreement on Views, and limiting disagreement.

Khazan notes that prior GM algorithms, implement "duplicate synchronization at the GMS and application levels [39]". Because only end-to-end confirmation of the agreement matters, the latter is necessary, but the former is not [39]. These prior GM

[1] More precisely, Consensus requires a minimum of two rounds only if at least one failure has occurred in the system. If no failures occur, Consensus is trivial, and can be achieved in less than two rounds.

algorithms generate new views only when members know that they are in agreement with each other. The extra synchronization at the GM level creates an unnecessarily high performance overhead. Because generating a view is conditional on knowing that agreement has been reached, these solutions often require two or more rounds of message exchange [39].

In contrast, Sigma allows members to generate transient, inconsistent views in the process of converging onto the same final view. Members do not know when they have agreed. This design enables Sigma to achieve Agreement on Views within one round of message exchange, once the group component becomes stable. There is, however, a tradeoff between providing lightning-fast Agreement on Views in some situations, and producing disagreement in other situations [39].

Certain asymmetric network events cause Sigma to produce inconsistent, transient views, which lead to disagreements. Although such disagreements are short-lived, it is desirable to avoid them, because they create unnecessary and useless overhead for applications. In addition, many applications, especially mission-critical ones, have a low tolerance for even momentary disagreement. Different applications have different definitions of what "short-lived" means – for some, it is on the order of seconds; for others, milliseconds. Also, some applications run in environments where asymmetric network events that cause disagreements are rare, while others may experience such asymmetries more often.

To eliminate disagreements, [39] suggests using Sigma with a filter for Limiting Disagreement (LD). Sigma's modular solution for limiting disagreement delays view deliveries until the latest proposals from all servers agree on the correct membership, and in the process filters out inconsistent views by preventing them from being delivered to the application. This modularity preserves Sigma's single-round performance when it is used with the LD filter. Since Sigma implements an established and widely used specification of group membership [22], it can be "plugged in" to virtually any group communication system, such as for example [4], [23], and [19].

1.3 Thesis Overview

In this section, we summarize the contributions of this thesis and provide a preview of the results.

1.3.1 Contributions

We present a performance analysis of Sigma to evaluate the practical implications of the theoretical results introduced in [39]. We implement and study Sigma in simulation using the ns-2 network simulation together with WAN connectivity traces collected by [9, 10] (also used in [29]). We compare Sigma's performance to that of Moshe, a recent practical, optimistic algorithm [34], which we implement in the same simulation environment. We also design and implement a leader-based version of Sigma (see Section 1.3.1.1), and compare its performance to that of a standard leader-based GM protocol that is used in Ensemble [19] and JGroups [16]. In our performance analysis of Sigma, we ask the following three basic questions:

- Is Sigma in practice more efficient than its predecessors? We evaluate efficiency by measuring a GM algorithm's average and maximum latency of execution (GM Latency), as well as the average message overhead that the algorithm produces.
- Is Sigma accurate enough to be useful in practice? We introduce and define two new metrics for quantifying GM performance: *agreement* – whether all members of a view have installed that same view; and *disagreement* – whether at least two members install a view with the same identifier, but different, nondisjoint membership sets. We measure how much agreement and disagreement a GM algorithm produces during execution of the trace, in terms of both percentages of views and actual raw numbers of views falling into each of these two categories.
- Is the LD filter necessary? If so, is it effective? In our measurements, we compare Sigma, where disagreements are unlimited, not only with Moshe and Ensemble, but also with Sigma_LD, the version of Sigma that has been equipped with the simple limiting disagreement filter proposed in [39].

1.3.1.1 All-to-All vs. Leader-Based Communication

Sigma uses all-to-all type of communication among members. Several prior results [14, 49] have suggested that leader-based communication schemes are more efficient in certain distributed environments, such as typical WANs and other high-latency fault-prone dynamic networks. Motivated by these results, we have also designed and implemented Leader-Based Sigma, which makes use of a more scalable and efficient centralized communication scheme.

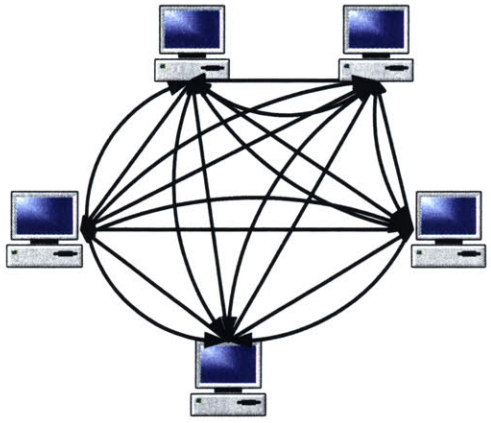

Figure 1-2. All-to-All Communication.

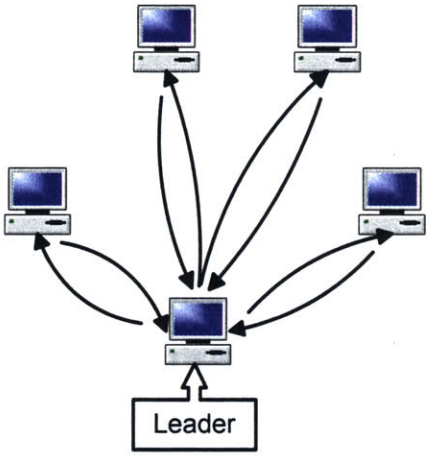

Figure 1-3. Leader-Based Communication.

In high-latency limited-bandwidth fault-prone dynamic networks, the number of message-exchange rounds and the number of messages within each round matters for GM protocols. In addition, the practical efficiency of a GM protocol can be optimized by sending messages over links with lower loss rates and greater bandwidth whenever possible. Leader-Based Sigma uses a minimal number of message exchange rounds like the original Sigma algorithm, and in addition reduces the number of messages sent during each round.

Also, like the original A2A protocol, Leader-Based Sigma is meant to be run by a relatively small number of membership servers maintaining membership information on behalf of a large set of clients (Figure 1-4). We believe that a combination of such a two-

tier architecture with GM servers running the fast leader-based GM protocol is important to enable large-scale GM services in high-latency, limited-bandwidth networks.

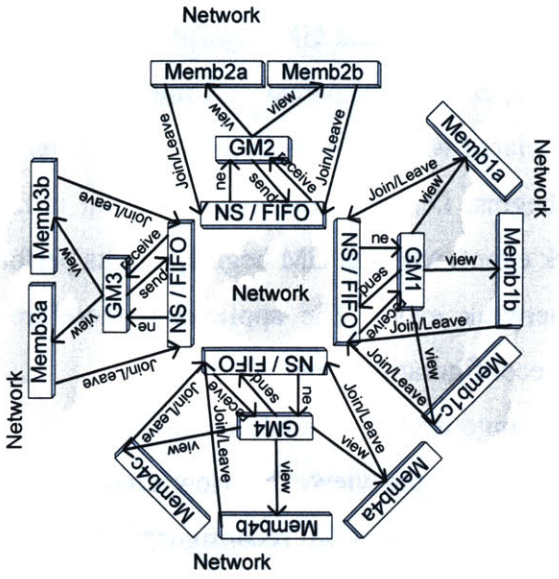

Figure 1-4. Two-tier Architecture. Clients (members) send join/leave requests to, and receive views from, the server (GM and NS/FIFO).

1.3.1.2 Sensitivity to Disconnects

As part of our performance analysis, we introduce a parameter called Sensitivity to Disconnects (SD), for the purpose of adjusting the sensitivity of the system to short-term network instabilities, similarly as discussed in [43]. Transient events, which occur frequently in wide-area networks, are difficult to distinguish from permanent events, because this requires knowledge of the future; applications have no way to distinguish a temporary departure from a permanent leave at the time of a node's disconnection [43]. Without such a distinction, GM delivers a view for each such transient event, the same as it would for a permanent event.

Many applications can tolerate transient disconnections without reacting to them; for such applications, it is important to minimize view changes due to transient events, because at the application level, each view change is associated with costly reconfigurations. In addition, applications have different definitions of what "permanent" means; SD enables them to adjust the granularity of events to be perceived as permanent.

By appropriately configuring SD according to their needs, applications can avoid unnecessary reconfiguration overheads by filtering out transient events.

At first glance, it may seem that the delay introduced by SD would defeat Sigma's purpose as an optimized single-round GM algorithm. We argue to the contrary: as seen by the application layer, SD is transparent and mutually exclusive of GM's execution. Reconfigurations associated with changes in group membership begin from the moment that a view change begins, i.e. when the network event is raised. SD only delays the raising of the network event, whereas GM begins, and its performance is measured, after the raising of the network event. The application therefore does not see the delay reflected in the view reconfiguration time.

The decision to change a view is made by the GM after the expiration of the SD. Once GM decides to change the view, the view change must be done quickly. Most applications tend to block while GM reconfigures; the priority is thus for GM to reconfigure as quickly as possible, once reconfiguration actually starts.

1.3.2 Results Preview

Figure 1-5 offers a glimpse into Sigma's efficiency relative to (a) Moshe and (b) Ensemble. Consistently with the theoretical results, Sigma forms views faster and with smaller message overhead than Moshe and Ensemble, across all SD values. Moreover, because most disconnects are transient in the real WAN traces that we used in our simulations, and are therefore detected asymmetrically by some members but not others, Moshe abandons its optimistic path and switches to its multi-round path for about half of the views.

In terms of accuracy, we observe that Sigma_LD, the version with the filter, produces virtually the same quality of views as Moshe and Ensemble. More specifically, for very small values of SD, Sigma_LD produces a tiny number of disagreed views – less than half of one percent; note that for such small SD values the view-oriented GM approach does not seem practical anyway, because it results in extremely high frequency of view changes.

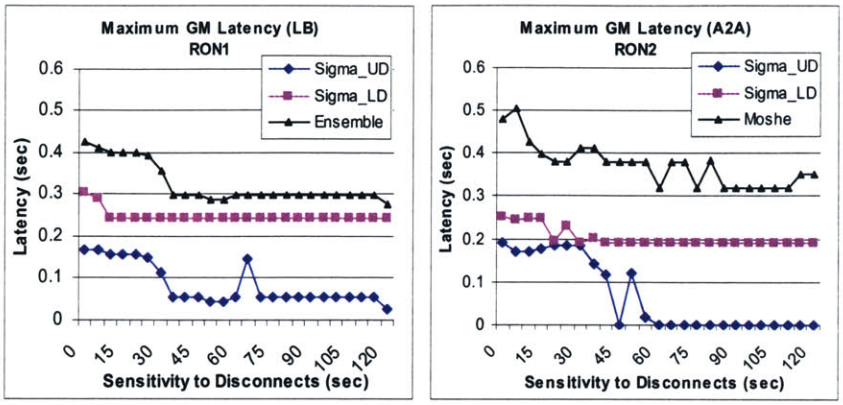

Figure 1-5. Sigma forms views faster than (a) Moshe and (b) Ensemble.

Importantly, for the rest of SDs (> 15sec), Sigma_LD, like Moshe and Ensemble produced no disagreement and 99-100% agreement on views. Together with the observation that Sigma_LD is faster than Moshe by as much as 260ms (by 30ms on average), these results confirm that Sigma_LD is widely applicable, and can potentially be used anywhere Moshe or Ensemble is used, with significant savings in latency and overhead.

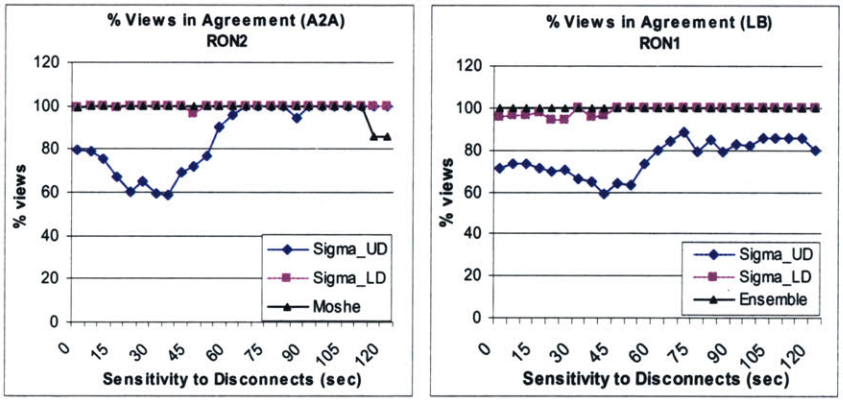

Figure 1-6. Agreement percentages for (a) Sigma vs. Moshe and (b) Sigma vs. Ensemble. Sigma_LD produces similar quality agreement as Moshe and Ensemble. Sigma_UD's performance improves with increasing SD.

Sigma without any filter (Sigma_UD) produces the same agreed views as both Sigma_LD and Moshe (and in the Leader-Based case, Ensemble); the differences in views between Sigma_UD and Sigma_LD consist almost entirely of short-lived views that Sigma_UD delivers during periods of asymmetric network instability. This observation is supported by the fact that Sigma_UD's agreement percentages increase to

match both Sigma_LD and Moshe (Figure 1-6(a)), and disagreement percentages decrease to zero, when transient disconnects of short duration are filtered out, through the application of a sufficient Sensitivity to Disconnects. The results are analogous when comparing Leader-Based Sigma_UD and Ensemble (Figure 1-6(b)). For applications that can tolerate these conditions, Sigma_UD is a powerful alternative that should be considered – not only does it have a smaller message overhead, slightly smaller even than Sigma_LD's, but also it is significantly faster than Moshe and Ensemble. For the specific WAN traces we used, which were collected from a real network, Sigma_UD is faster than Moshe and Ensemble by as much as 400ms.

1.4 Roadmap

The rest of this thesis is organized as follows. Chapter 2 provides context for our work of optimizing group membership by describing group communication systems and other applications of group membership that have been developed. Chapter 3 describes the group membership specification, including its properties, environment model, and architectural options. Chapter 4 specifies the algorithms that we study in this thesis – Sigma, Moshe, Leader-Based Sigma, and Ensemble – and offers a theoretical discussion of their performance tradeoffs. In Chapter 5, we describe how we implemented the simulation, from constructing the network topology and modeling of realistic network events, to our implementation of the group membership algorithms, failure detector, and analysis tools. Chapter 6 presents the results of the performance analysis, measuring agreement, disagreement, latency, view duration, and message overhead. In Chapter 7, we discuss Sigma's practical potential in light of these results, and conclude with a view to the future.

Chapter 2

Applications of Group Membership

To put into context the utility of Group Membership (GM) and group communication systems, as well as the importance of optimizing GM, we describe in this chapter some applications that have been designed and discussed in the literature of the field. First, we provide a brief history of group communication systems, along with an evolutionary timeline of some GCS that are widely known and referenced often in the field. We then proceed to a more in-depth description of five well-known applications that have been developed on top of GM: Totally Ordered Multicast [26], Virtually Synchronous Group Multicast [32], Load-Balancing Replicated Data [38], Intermittently Atomic Data Services [36], and Group Key Agreement [8]. These applications are a foundation for higher-level applications that often require the services (reliable ordered multicast, load-balancing, replication, and group security) and guarantees (total ordering, virtual synchrony, intermittent atomicity) that these applications provide.

The applications that we describe in this chapter directly depend on the views generated by GM. For most, knowledge of group membership is a pivotal factor that enables them to achieve their intended goals. In addition, many take advantage of the useful properties provided by the GM specification (see Chapter 3), such as the monotonically increasing property of view ids and the unique association between distinct membership instances and view ids.

No matter what their motivation for using GM, all these applications have one common feature: they have two modes of operation – a normal mode, and a reconfiguration mode (Figure 2-1). Normal mode proceeds while the network is stable, during which the application engages in activities that fulfill its characteristic purpose. Reconfiguration mode begins when network instabilities cause changes in group membership and in the process introduce inconsistencies of state into the distributed system. Because normal mode cannot proceed in the presence of such inconsistencies, each process participating in the application switches to reconfiguration mode in order to

synchronize its state with the others. Our discussion focuses on the reconfiguration mode of each application and the role of GM in its functionality.

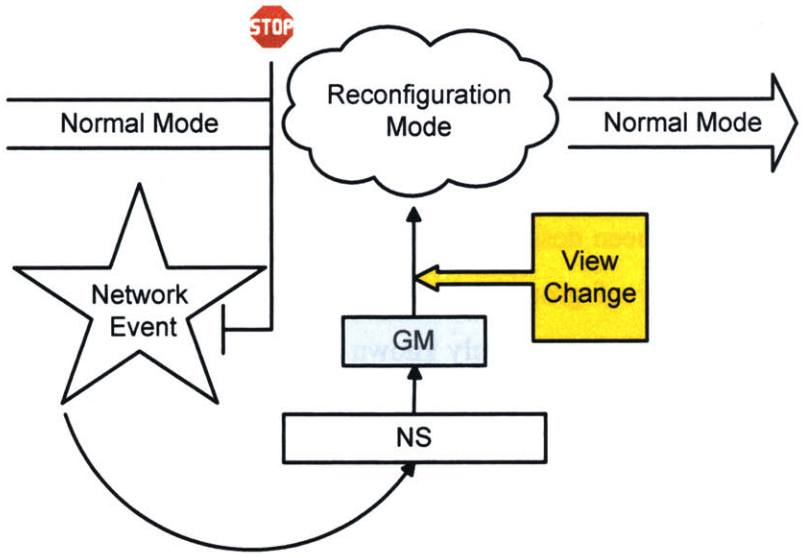

Figure 2-1. Normal vs. Reconfiguration Mode of GM Applications. Reconfiguration Mode represents an interruption of the application's normal operation.

2.1 Group Communication Systems

Group communication systems harness the power of the group abstraction to provide modular services for fault-tolerant group communication. Process groups were first proposed by Cheriton and Zwaenepoel in the design of the V system [21]. Birman and Joseph later applied this idea to the context of fault-tolerance, through the Isis group communication system [18, 52]. The V system (1985) was the first to make use of the group abstraction as a software base for constructing distributed systems. In the V kernel environment there are many cooperating processes on different machines; at a certain level of abstraction, these processes form logical groups. Cheriton and Zwaenepoel found that one particular operation enabled by this group-oriented organization is "group interprocess communication, an application-level abstraction of network multicast [21]".

The Isis group communication system (1987), which is considered to be the first GCS, demonstrated that the group-oriented approach to building fault-tolerant distributed software is simpler, more flexible, and more robust than alternative approaches [18]. Isis implements *virtual synchrony* for replicated services – messages are delivered to all

members in a consistent order, simulating a synchronous system in an asynchronous environment.

A variety of GCSs and supporting subsystems have been developed since Isis, each offering new innovations. Some examples are as follows. Transis (1992), a transport layer subsystem for GCSs, improves the efficiency of Isis by implementing broadcast communication [3]. Totem (1995) provides greater consistency by placing a total order on broadcast messages through the use of a single logical ring to organize group members [7]. Totem is extended to support multiple rings in [1]. Spread (1998) is a hybrid GCS that adapts a similar multiple ring protocol for use in Wide-Area Networks by using rings on a local-area level and bridging them with a Hop protocol [4] [2].

Horus/Ensemble (1996) introduces a novel stackable architecture for GCSs, in which each service is implemented as a different layer that can be modularly added or removed as needed by higher-level applications [19, 51]. JGroups [16], an open-source GCS implemented in Java, has recently emerged, which applies this same stackable architecture and a similar group membership algorithm as used in Horus/Ensemble, and offers a range of protocols and services that can be "mixed and matched" as needed by the application.

True to its name, Xpand (2000) offers expanded utility to as wide a spectrum of collaborative WAN applications as possible, by providing two types of services: weak and strong [13, 23]. The strong services closely resemble traditional GCS semantics, while in the weak services, requirements are relaxed to approximations, allowing room for QoS negotiation [13]. In Section 2.2, we also describe the VS group communication system that was developed to provide the view-synchrony property in addition to the usual GCS primitives.

2.2 Totally Ordered Multicast

As mentioned in Chapter 1, a group communication system (GCS) offers reliable multicast as one of its services, besides group membership. However, this multicast primitive offers only weak ordering guarantees, such as FIFO; a multicast service that can provide stronger guarantees about ordering can be far more useful to higher-level applications. There are three main types of ordering multicast message delivery: First-In-

First-Out (FIFO), Causal order, and Total order. FIFO (Figure 2-2) guarantees that if a message m0 is sent before a message m1 by process p, then m0 is delivered before m1 at all destinations that they both have in common [17]. While FIFO ordering is focused on events that happen at a single place in the system, Causal ordering (Figure 2-3) pertains to events that can span multiple processes [17]. Causally ordered delivery ensures that *any* two messages m0 and m1, sent by possibly different processes, where m0 was sent before m1, will be processed in the same order at all destinations that they both have in common [17]. Total ordering (Figure 2-4) is the strongest ordering option, requiring any processes that *receive* the same two messages m0 and m1 to *receive* them in the same order [17].

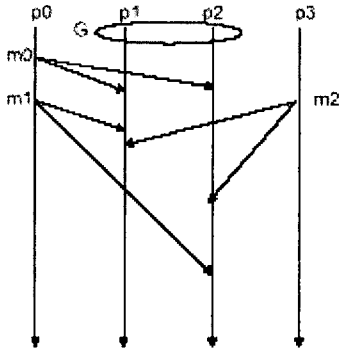

Figure 2-2. FIFO Ordering.
Image Source: [17].

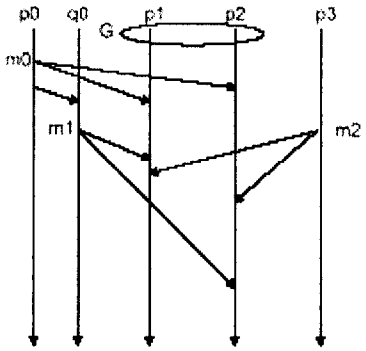

Figure 2-3. Causal Ordering.
Image Source: [17]

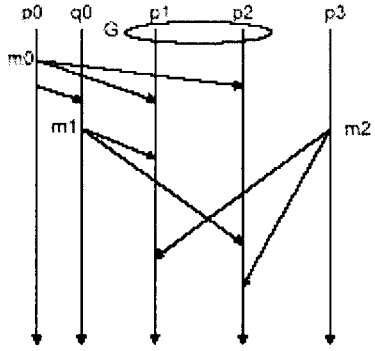

Figure 2-4. Total Ordering.
Image Source: [17]

Totally Ordered Multicast, designed by [26] and abbreviated VStoTO, is an application built on top of a View-Synchronous Group Communication System (VS), also designed by [26] based on COReL [31]. VS, like any other GCS, is itself an application of group membership, but in addition, it provides a within-view totally ordered broadcast service that guarantees message delivery within each view to follow a total ordering. VS and VStoTO serve as foundations for powerful distributed applications such as replicated data services and sequentially consistent memory, both of which we discuss in further detail in Sections 2.4 and 2.5.

Normal mode in VStoTO proceeds while the network is stable. Each process keeps, among other things, two state variables: a *content* relation and an *order* sequence. To each message M received from the client, VStoTO at processor p assigns a label L consisting of:

- The view id at p when the message arrives
- A sequence number
- And the Processor ID "p"

Processor p stores the <label, message> pair LM in its "content" relation, and sends LM to the other members of the current view, using VS. The other processors then add LM to their own "content" relations. A processor "p" is in a *primary view* when its view includes a certain quorum of processors in the membership set. When p receives LM while in a primary view, it also places the label L at the end of its "order" sequence. Otherwise, a processor p that receives LM while in a non-primary view simply records LM in "content". Each *content* relation is a partial function from labels to messages. When considered in combination with *content*, *order* describes a total ordering of sent messages. [26]

While the task of within-view total order is made simple by the use of VS, the challenge of VStoTO is to achieve total ordering across view changes. VStoTO achieves this during reconfiguration mode. Reconfiguration mode of VStoTO is initiated by the Group Membership service, when network events cause changes in group membership. GM agrees on a view and delivers it to VS, which in turn passes the new view to VStoTO. Upon receiving the new view, VStoTO proceeds to a state-exchange protocol, which involves exchanging and combining information to integrate the knowledge of different members of the new view. Members of the group execute a series of operations to determine which member among them has the most up-to-date information, such as whose *order* sequence to adopt for use in the new view by everyone else [26].

2.3 Virtually Synchronous Group Multicast

Virtually Synchronous Group Multicast [32] is a reliable multicast service that implements virtual synchrony semantics. As discussed previously, reliable multicast is one of the services that a GCS provides, in addition to the group membership service. In fact, reliable multicast can be seen as a client of the GMS. Virtual synchrony is a property guaranteeing that processes moving together from a view v to another view v' deliver the same messages in v. By thus synchronizing membership notifications (which ultimately result in a new view v') with regular messages, and in doing so, associating message send

and delivery events with views, virtual synchrony simulates a "benign" world in which message delivery is reliable within the group [32].

Like VStoTO, Virtually Synchronous Group Multicast operates with a normal mode and reconfiguration mode. The components of its implementation are as follows, as summarized from [32]. A connection-oriented communication service CO_RFIFO implements a FIFO queue channel for every pair of end-points. CO_RFIFO provides reliable, gap-free FIFO message delivery. Within-view reliable FIFO (WV_RFIFO) is built on top of CO_RFIFO and a membership algorithm (MBRSHIP), which additionally guarantees that a prefix of messages (not necessarily all) is delivered in the same view in which these messages were sent. This is done by tagging messages with the views in which they were sent and allowing delivery of a message only when its view tag matches the end-point's current view. During normal mode, Virtually Synchronous Group Multicast runs WV_RFIFO. [32]

Virtual Synchrony RFIFO (VS_RFIFO) is implemented on top of WV_RFIFO, extending WV_RFIFO to guarantee that endpoints which transition directly from view v to the same view v' deliver not just *some* prefixes of messages, but *the same* prefixes of messages. Reconfiguration mode consists of running VS_RFIFO, because it handles the transition from an old view to a new view. Reconfiguration mode begins when GM notifies WV_RFIFO of an impending view change by delivering a startChange identifier. Endpoints must then learn which other endpoints may transition from v to v', and achieve agreement with them on the exact same prefix of messages to be delivered [32]. WV_RFIFO runs in parallel with MBRSHIP.

2.4 Load Balancing Replicated Data

A replicated data service is designed in [38] that load balances queries and guarantees sequential consistency. Abbreviated VStoD, it is designed to operate on top of VS, the view synchronous GCS specified in [26]. VStoD maintains, in a consistent and transparent manner, a data object that has been replicated at a fixed set of servers. Clients can update and query this object; the replicated data is kept coherent by applying all update and query operations in the same sequence at all replicas.

VStoD is implemented by a layer of replicated servers on top of a communication layer that consists of a group communication service satisfying VS. The replicated servers form the group that is serviced by the GCS. The GCS is used for communicating update and query requests to the group members. VStoD relies on the VS property to enforce identical sequences of update requests at all servers and to schedule query requests correctly [38].

The replicated servers layer runs the VStoD application, and operates with a normal mode and reconfiguration mode. In normal mode, a server participates in an already established view, processing update and query requests from clients. Servers maintain, among other things, two prefixes of the sequence of update requests: *safe*, the update requests that are safe to execute, and *done*, the update requests that have already been executed. Normal mode guarantees that the safe and done prefixes are always consistent among all servers [38].

During recovery mode, servers synchronize their query and update request sequences. To synchronize queries, when a server learns of a new view, it moves its own pending queries for reprocessing and erases any information pertaining to the queries of other servers [38]. To synchronize updates, a server collaborates with others to ensure that the past execution histories of all servers of the new view are consistent. To do so, each server must be able to tell how advanced its state is relative to the others. Criteria for judging a server's "expertise" are (1) the latest primary view of which the server knows, (2) a server's *updates* sequence, and (3) a server's *safe* prefix. Because normal mode can only begin when servers have identical updates sequences and safe prefixes, this is the focus of resynchronization. Each server engages in "advancing the expertise" of other servers to the highest expertise of which it is aware, its *cumulative expertise* [38]. Servers adopt another server's cumulative expertise if it is more advanced than their own. Once this process of advancing expertise finishes, the server of a primary view extends its *safe* prefix to cover the entire *updates* sequence and moves all pending *update* requests not in the safe prefix back for reprocessing. After this step, reconfiguration ends and the server returns to normal mode.

2.5 Intermittently Atomic Data Service

Active data replication benefits not only from load balancing such as that provided by VStoD described in Section 2.3, but also from an enforcement of atomicity. In large-scale, wide-area network environments, providing atomicity is costly in terms of overhead and latency. To provide an effective alternative, Khazan and Lynch propose in [36] a weaker atomicity property, called intermittent atomicity, which guarantees that clients perceive the data object as atomic, but only while the underlying network component is stable. Atomic semantics are restored within some finite amount of time after stability returns [36]. During the restoration process, clients are informed about the current group membership and the new state of the data object. Khazan and Lynch describe in [36] a design for an Intermittently Atomic Data Service (IADS) that provides the functionality that satisfies this specification.

IADS allows a dynamic group of clients to access and modify a replicated data object in satisfaction of the intermittent atomicity property. It operates on top of a group communication service, and makes use of the virtual synchrony semantics that it provides. Specifically, the Virtually Synchronous Delivery property, as described in Section 2.2, guarantees that, if the object replicas were mutually consistent upon entering normal mode in view v, they remain mutually consistent when view v' is delivered. Normal mode is when IADS processes client requests to query and modify the object replicas. Reconfiguration mode begins when a view change occurs, and consists of state-transfer: "a new state of the object is computed from the merge of object replicas [36]". In the state-transfer protocol, members of the new view collect the states of each other's object replicas [36]. Then, each computes a new state for its replica by merging the collected states [36].

Some optimizations of reconfiguration mode are enabled by the virtual synchrony guarantees [36]. First, it is enough for only one of the members to communicate the state of its replica during the state-transfer protocol. This member must have been a member of the previous view, and must remain as a member in the new view. Second, state-transfer is unnecessary in situations where the membership of the new view is the same as the membership of the previous view. In this case, normal mode can continue, uninterrupted.

2.6 Group Key Management

To achieve maximum security for group communication, every view change must be "accompanied by a corresponding adjustment to group security parameters [5]". One such parameter is the secret group shared key, also called a *group secret*. Because most routine group security services depend on the sharing of a common secret, the group secret key is one of the most fundamental group security mechanisms [5]. Alternatives to a secret group key are public key encryption, and pairwise secret keys between each pair of members; however, both involve impractically high overhead [5]. We summarize protocols for group key management presented by [5],[6], and [42].

Group key management enables authenticated and private communication within a group. Normal mode consists of members using the group key for secure communication. Reconfiguration mode generates a new secret group key following every group membership change. There are two types of group key management protocols – centralized, where the group key is generated by a single member, which then distributes it to the other group members; and distributed, where all group members participate in key generation. In distributed group key management, all group members collectively generate or agree on a group key. In most distributed protocols, the group secret key is a function of all group members' individual contributions. Because each individual member's contribution is known only to that member, such a contributory mechanism facilitates authentication. [6]

2.6.1 Secure Spread

Amir et al. implement group security services on top of the Spread wide-area GCS [2]. They make use of the Group Diffie-Hellman (GDH) protocol provided by the CLIQUES system [47, 48], for authenticated contributory group key management. CLIQUES depends on an underlying GCS to provide the group membership. GDH chooses a *group controller* that is charged with initiating key adjustments following group membership changes. The most recently joined group member is chosen to be the group controller. Each group member contributes equally to the group secret key.

CLIQUES implements other group key management protocols besides GDH [6, 48]. Centralized Group Key Distribution (CKD) chooses as group controller the oldest member of the group. Whenever the group membership changes, the group controller generates a new secret key by itself and distributes it to the group. Before doing so, it establishes a new secure channel with each joining member using authenticated two-party Diffie-Hellman.

Other options provided by CLIQUES are the Burmester-Desmedt (BD) protocol, which distributes computation of the group key among all members of the group, so that each member performs only three exponentiations; and Tree-Based Group Diffie-Hellman, which computes a group key derived from the contributions of all group members by using a binary tree.

2.6.2 Secure Ensemble

Rodeh implements group key management on top of the Ensemble GCS [42]. We briefly summarize his Diamond group key agreement protocol. Diamond takes advantage of the fact that Ensemble uses a leader-based GM algorithm, by following a centralized protocol design. The name comes from its use of a "diamond" graph structure to characterize the group membership and its secure channel infrastructure for efficient key exchange. When a membership change occurs, Diamond must wait for the new view to be delivered by GM, so that the diamond graph can be modified accordingly.

When GM delivers a new view, a representative from each merging group component sends its diamond structure to the leader, which then merges together the different diamonds into a new one D. The leader also computes a schedule Q that determines the order in which members will participate in the key exchange. The leader then multicasts both D and Q to the group. Upon receiving this information, the first member listed in Q chooses a new key K and multicasts it to everyone else. The last member in Q multicasts a ProtoDone message when it receives K. When members receive the ProtoDone message, they rebuild the new diamond structure D and resume normal mode. If a failure occurs during the running of the protocol, all members will abort the protocol. In this case, it is expected that the application will request a rekey in the forthcoming new view [42].

Chapter 3

Group Membership Specification

Our work is based on the standard group membership specification established in [22]. This chapter describes the environment, architecture, and properties that form the GM specification assumed by the design of Sigma, Moshe, and Ensemble. We carry over these assumptions into our implementation and analysis of these algorithms. The GM environment, described in Section 3.1, assumes a two-tier client-server architecture with an underlying failure detection and reliable FIFO communication service. As part of the discussion of the GM environment, we explain why GM cannot be fully replaced by failure detection.

Section 3.2 summarizes the liveness, local monotonicity, and self-inclusion properties that GM must satisfy in order to conform to the specification. In addition, we provide two new definitions that articulate what agreement and disagreement mean under the standard GM specification. We also discuss, in Section 3.3, the architectural options for achieving a scalable GM solution. Specifically, we focus on the tradeoffs between an all-to-all communication protocol and a leader-based one.

3.1 Environment

We assume the same widely accepted environment model as described in [34, 39]. The environment is asynchronous message-passing [41]. Processes may fail by stopping, and links may fail and later recover, possibly causing network partitions and merges. Network events may partition the group into components, and the partitions can be unclean: different members may have contradicting and asymmetric perceptions of the memberships of their group components.

Figure 3-1 illustrates the layers that comprise the GM environment. Most GM algorithms utilize external failure detection services and reliable FIFO (RFIFO) communication services [22]. Both Sigma and Moshe interface their failure detection service with a network event notification service (NS) [34, 39]. The task of NS is to inform GM of the events that affect the membership of the group. NS does this by

generating `ne_r(joining, leaving)` events at a member `r`. The members listed in the `joining` set are either joining or re-connecting to the group; those listed in the `leaving` set are either leaving or are suspected of having disconnected or crashed.

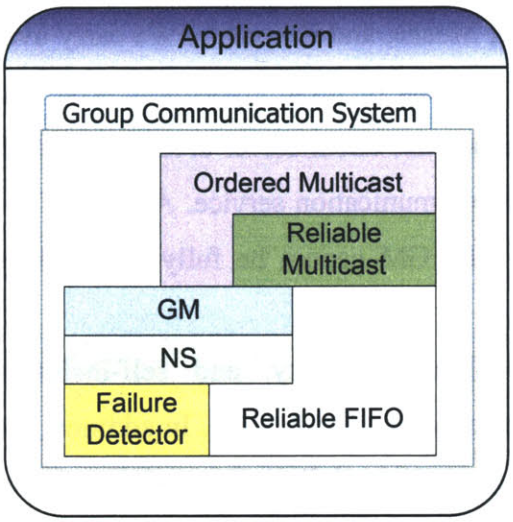

Figure 3-1. Layers of the GM Environment.

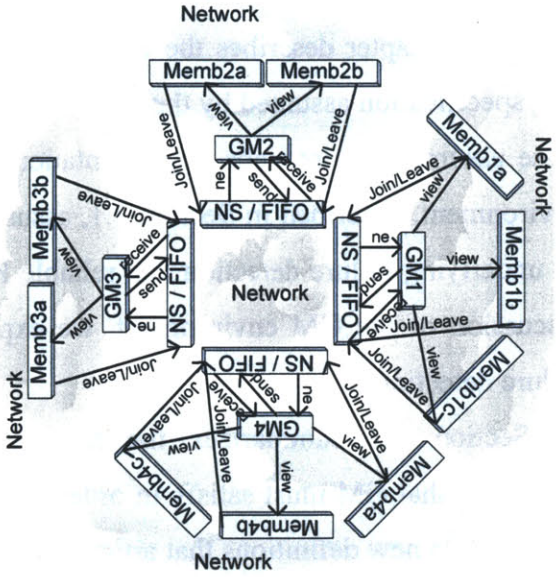

Figure 3-2. Two-tier Architecture. Clients (members) send join/leave requests to, and receive views from, the server (GM, NS/FIFO).

Sigma and Moshe require NS at least to be complete – to correctly identify all permanently disconnected or crashed members. We assume NS and RFIFO to be such that, if member `r` sends message `m` to member `u`, then either `m` is delivered to `u`, or NS notifies `r` that `u` is unreachable. NS need not satisfy other properties, e.g. accuracy, symmetry, or transitivity.

While the absence of such helpful properties does not violate correctness of Sigma and Moshe, it affects their behavior and performance. It is advantageous to use NS services that attempt to provide such properties. CONGRESS is an example of such a NS service [12, 34]. As discussed in [22], Ch. 8.1, NS and RFIFO can be implemented separately, but "are often implemented jointly by the same service, over an unreliable network". A number of such services are referenced by [22], which suggests that TCP itself "implements a similar [joint] service over the unreliable IP protocol". GM can therefore be implemented directly on top of TCP.

3.1.1 Single-tier vs. Two-tier Architecture

In [34, 39], Sigma and Moshe are described as part of a two-tier client-server architecture, in which a small set of membership servers maintain membership information on behalf of a large set of clients (Figure 3-2). As explained in [34], such architecture is more appropriate for supporting large groups in WANs than a single-tier architecture, in which all members participate as membership servers.

Henceforth, we simplify our discussion and implementation of the GM environment to a single-tier architecture. This is sufficient for studying the agreement properties of these algorithms for small groups of nodes. Moreover, the results of such a single-tier analysis extend to a two-tier setup in which the nodes that we study act as membership servers, each supporting a large number of clients.

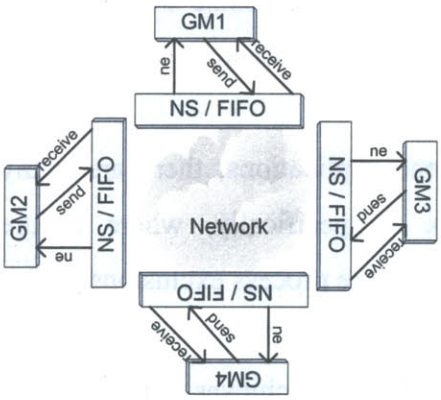

Figure 3-3. Single-tier architecture.

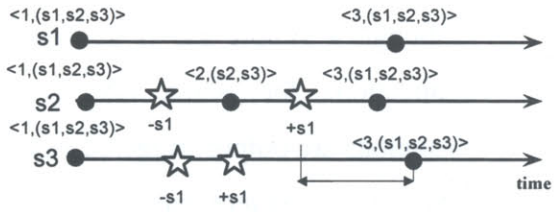

Figure 3-4. Illustration of Example 3-1. Filled circles are `view` events and stars are `ne` events.

Figure 3-3 depicts the resulting single-tier architecture. Each member runs a GM algorithm, which exchanges messages with other members by using reliable FIFO links and receives notifications from NS about changes in group's membership.

3.1.2 Failure Detectors vs. Group Membership

As we have seen in Section 3.1, the GM environment assumes the existence of an underlying mechanism for failure detection (FD) that informs GM of network events. Sigma and Moshe assume NS, the network event notification service, to implement FD.

To be able to detect whether a leave or a join has occurred, the failure detector must itself keep track of the connected members, in parallel to GM. Sigma's design, and the standard GM specification upon which Sigma is built, decouples FD and GM to such an extent that the separate application of FD without GM can be readily envisioned. Why, then, do failure detectors not obviate the need for group membership?

There is an important difference between the two – FD provides inconsistent information about failures, while GM provides consistent information by virtue of agreement on views [44]. [44, 50] demonstrate FD to be a possible alternative to GM for applications where failure suspicions do not lead to process exclusions, and, in the absence of crashes and suspicions, to perform as well as GM. However, not only do crashes occur frequently in real networks, but also many applications exist in which failure suspicions do require process exclusions. In such applications, Schiper et al. prove GM to be advantageous, and even mandatory [44, 50]. We emphasize the following reasons to explain why GM remains a necessary construct, despite the separate applicability of FD, and is therefore important to optimize:

1. Although a simple FD is sufficient for some applications, there are many applications for which GM is preferable – specifically, where failure suspicions are output-triggered and therefore require process exclusions [44].
2. GM is more resilient than FD in the presence of crashes [50].
3. FD achieves agreement on membership, while GM achieves agreement on view identifiers [39]. In other words, the group membership algorithm is responsible for associating a given view identifier with a given membership set such that all members compute the same association. This agreement on view identifiers is a powerful abstraction that serves as the foundation of view-oriented GM applications such as those described in Chapter 2.

3.2 Properties and Definitions

The established GM specification [22] that we follow requires GM to satisfy liveness, local monotonicity, and self-inclusion properties. The liveness property applies to NS – once NS perceives a group component to have stabilized, eventually every member must get the same views, reflecting correct membership. Eventual agreement is

required only after the group stabilizes, so the specification allows disagreement before stabilization. In fact, we cannot avoid disagreement while guaranteeing delivery of views with correct membership in an asymmetric, failure-prone environment [20, 22].

Property 3-1. (Liveness) In a group component perceived to be stable by NS, GM eventually outputs the same view, with membership G, to all the members of G, and does not output to them any subsequent views [39].

The specification also includes two safety properties: (1) local monotonicity – view ids must monotonically increase; and (2) self-inclusion – views must be delivered only to members listed in the views' membership set. The specification is partitionable, meaning there may be concurrent views with disjoint membership.

Property 3-2. (Local Monotonicity) If a member p installs view V2 after installing view V1, then the identifier of V2 is greater than that of V1 [22, 39].

Property 3-3. (Self-Inclusion) If a member p installs view V, then p is a member of V [22, 39].

3.2.1 Group Membership Specification in Action

Example 3-1 illustrates how group membership works according to the GM specification. This example demonstrates the three basic GM properties -- liveness, local monotonicity, and self-inclusion.

Example 3-1. Figure 3-4 shows three members, s1, s2, and s3. Initially, all have the same view v1=<1,{s1,s2,s3}>. Then, s2 and s3 receive network event *ne(-s1)* from NS informing them that s1 disconnected (depicted by stars labeled "–s1"). GM starts forming new view v2 with s2 and s3 as members, according to the self-inclusion property. After s2 finishes view formation, it delivers v2=<2,{s2,s3}> (depicted by the accordingly labeled filled circle). But before s3 has a chance to deliver v2, s1 reconnects, and s2 and s3 receive ne(+s1) from NS. Instead of delivering v2, s3 starts working with s1 and s2 on forming the next view. After the network stabilizes, all have the same view, v3=<3,{s1,s2,s3}>, satisfying the liveness property. Because the view identifiers increase with each new view, local monotonicity is also satisfied.

Example 3-1 illustrates an asymmetric situation in which s1 did not even detect that it disconnected from s2 and s3. Also, s2 delivered v2, but s3 did not, because the

membership changed before s3 was ready to deliver it. This behavior is allowed by the specification--GM allows members to disagree and deliver different views. What it requires is that after the group stabilizes, everyone must eventually get the same views.

A member may deliver more views than the number of network events (NEs) it receives, because not all NEs are reported to everyone. On the other hand, not every NE results in a view delivery, because view formation may be interrupted by subsequent NEs. In Example 3-1, member s1 does not receive any NEs but delivers view v3; conversely, member s3 receives two NEs but delivers only one view v3.

3.2.2 Agreed, Transient, and Disagreed Views

The goal of a GM algorithm is to deliver the same views, corresponding to correct group membership, to all the view's members. Sometimes GM algorithms do not realize this goal because membership changes while view formation is already in progress. In such situations, it is better to abandon the ongoing view formation attempt and move on to forming a view with correct membership, than to continue forming and deliver a view that has obsolete, outdated membership [17]. As a result, only a subset of members might deliver such interrupted views; these views end up being transient because they are soon succeeded by different views. Such transient views can lead to disagreements. We define agreed, transient, and disagreed views as follows.

> **Definition 3-1. (Agreed, Transient, and Disagreed Views)** A view v is an **agreed** view or is **in agreement** if and only if every member in v.set delivers v. A view v that is not an agreed view is transient. Views v and w are **disagreed** views or are **in disagreement** if and only if they have the same view ids but different, overlapping membership sets; i.e., if and only if $((v.id=w.id) \wedge (v.set \neq w.set) \wedge (v.set \cap w.set \neq \{\}))$.

In Example 3-1, v1 and v3 are in agreement because they are delivered to all their members. View v2 is transient because it is delivered to s2 but not s3. View v2 is not in disagreement with any other view because no other view in Example 3-1 has the same view identifier. Disagreed views are transient views that can occur in some GM algorithms (such as the theoretical Sigma algorithm) because members observe concurrent changes in group membership differently.

3.3 Scalability of Group Membership

All-to-all (A2A) message exchange is essential to reaching agreement within a single-round, but it results in high message overhead, specifically $O(n^2)$ unicast messages (or O(n) multicast messages) in the worst case for n participating nodes. Moreover, the worst case in A2A protocols is, in fact, the common case. As group membership protocols are scaled to larger and larger numbers of participating processes, they experience a steadily growing overhead as greater numbers of messages are exchanged, acknowledged, lost, and retransmitted. Greater message overheads lead to increased contention in the network, which results in network congestion, overflowing message queues, and ultimately message delay and loss. Lost messages need to be retransmitted, compromising the efficiency of the membership algorithm while further increasing the message overhead [49].

There are two orthogonal approaches to achieving scalability in a GM algorithm, and both aim to reduce message overhead. The first approach, which we have discussed in Section 3.1.1, is a two-tier architecture, in which a set of membership servers maintains group membership on behalf of a set of clients, the group members [34]. The scalability of a two-tier architecture comes from the idea of applying it to form a hierarchy of membership services, where membership servers at one level can at the same time also be clients of other membership servers that operate at a higher level in the hierarchy. Using this hierarchy, the number of nodes exchanging messages amongst each other in an all-to-all manner can be kept constant.

The second approach directly reduces the number of messages being exchanged, rather than the number of nodes exchanging them: the use of a leader-based protocol instead of A2A. Instead of having all members send messages to all other members, a leader-based protocol chooses one of the members to act as an intermediary for communication among members; this chosen member is called the "leader". Members only send messages to, and receive messages from, the leader. The leader can be picked deterministically without any additional communication, in a way that results in every member choosing the same leader when the underlying group component stabilizes (like in [19, 51]).

Bakr and Keidar [14] studied the effect of message overhead on algorithm performance in the context of the characteristically lossy nature of a WAN. They observed that the running time of A2A protocols in the presence of lossy links and message losses is actually longer than for leader-based protocols, despite the fact that leader-based protocols involve an extra communication step, because even relatively low loss rates get amplified by the greater volume of messages sent in an A2A algorithm. They observed this effect with just nine participating nodes; GM applications can conceivably involve much larger, and continuously evolving, numbers of participating members.

Urban and Schiper [49] found a leader-based consensus algorithm to be more efficient in practice than an equivalent A2A algorithm. They observed that the greater message overhead found in the A2A algorithm increases network contention, which in turn causes message loss and communication delays.

Both [49] and [14] agree that centralized, leader-based communication schemes can be more efficient in practice in certain networks than A2A protocols, due to dramatically reduced message overhead. They also imply that a leader-based protocol offers more room for performance optimization than A2A, by enabling the selection of a leader that can offer the best quality of service, for example one with greater resources or more reliable connections. Thus, in dynamic networks where message loss is common and unavoidable and the number of participating members can be large and evolving, we believe that Sigma can be made more scalable and efficient by transforming its all-to-all communication scheme into a leader-based one. In this thesis, we pursue this idea further by designing Leader-Based Sigma as a more scalable alternative to Sigma. We describe both Sigma and Leader-Based Sigma in Chapter 4.

Chapter 4

Algorithms

The algorithms studied in this work are Sigma, Moshe, Leader-Based Sigma, and Ensemble. This chapter describes these algorithms in greater detail and afterwards discusses example scenarios that demonstrate the tradeoffs between Sigma and Leader-Based Sigma, between Sigma and Moshe, and between Leader-Based Sigma and Ensemble.

Sigma [39] is the first group membership algorithm to achieve a single-round worst-case upper bound. A filter LD for limiting disagreement is suggested by [39] as a possibly desirable option for use with Sigma. We therefore study Sigma with and without the LD filter – Sigma_LD and Sigma_UD, respectively. Sigma follows an all-to-all (A2A) communication protocol; we therefore also call it A2A Sigma.

Moshe [34] is an efficient optimistic all-to-all algorithm that has been established to be practical. It achieves single-round agreement in the best case, but takes two or more rounds in the worst case. Its architectural similarities to Sigma, and its practical repute, make it an ideal candidate for a comparative analysis with Sigma.

We have designed Leader-Based Sigma [37] as a more scalable alternative to A2A Sigma, by transforming the all-to-all protocol into a leader-based one. As with A2A Sigma, we study Leader-Based Sigma with and without the LD filter. The GM algorithm that we refer to as Ensemble is a standard leader-based group membership algorithm that was first introduced in the Horus system [27] and has since evolved into the Ensemble [19] and JGroups [16] systems. As a leader-based algorithm with wide practical application, Ensemble is an ideal candidate for comparison with Leader-Based Sigma in our analysis.

4.1 Sigma

Sigma is a single-round GM algorithm that reaches Agreement on Views within one message latency after the final network events affecting the group component become known to all the members [39]. The algorithm has different members converging on the

same final view quickly after the group component becomes stable. Members receive network events from their local NS, as well as view proposals from other members. The basic idea is to send view proposals in response to a) network events and b) proposals with higher ids. After a network component stabilizes, the largest proposed view id reaches all members within one message latency.

```
Sigma[Member r]

On receive ne_r(joins, leaves):
  prop[r].set ← prop[r].set ∪ joins - leaves
  let max_id = max{prop[i].id | i ∈ prop[r].set
                  ∧ prop[i].set = prop[r].set}
  prop[r].id ← max(prop[r].id + 1, max_id)
  send_view ← true
  send proposal (r, prop[r].id, prop[r].set)
   to prop[r].set (deliver immediately to self)

On receive proposal_r(s, id, set):
  prop[s] ← ⟨id, set⟩
  if(set = prop[r].set)
    if(id > prop[r].id)
      prop[r].id ← id
      send_view ← true
    endif
    DeliverViewIfReady()
  endif

DeliverViewIfReady() =
  if((send_view = true) ∧ LD())
    deliver view ⟨prop[r].id, prop[r].set⟩
    send_view ← false
  endif
```

where ***LD()*** is a filter used in the Sigma_LD algorithm:
LD()≡((∀i∈prop[r].set)prop[i].set=prop[r].set))
For Sigma_UD, ***LD()***≡true.
* Each event handler is executed atomically.

Figure 4-1. Pseudocode for A2A Sigma.

Figure 4-1 shows the pseudocode for the Sigma algorithm run by a member r; all members run the same algorithm. There are two event handlers: A member can either receive a network event (NE) from NS or a proposal from another member. A proposal carries an id and a set. Each event handler is executed atomically. Sigma's relationship between GM and NS is shown in Figure 4-2.

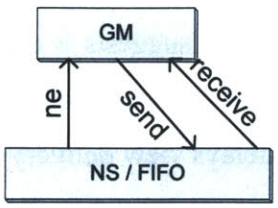

Figure 4-2. Sigma's Relationship between GM and NS.

When member `r` receives an NE, it does the following:

- updates its membership set `prop[r].set`;
- increments its identifier `prop[r].id` if already the largest among the proposals that `r` has for the current membership set; otherwise `r` sets its `id` to the largest one without incrementing;
- sends a proposal `<id, set>` to all members that are relevant to the membership `set`; it immediately processes its own proposal.

When member `r` receives a proposal `<id, set>` from `s`, it does the following:

- saves the proposal in `prop[s].set`;
- updates its `prop[r].id` if the proposal has a higher id for `r`'s current membership set;
- attempts to deliver view `<id, set>` if it is `r`'s own proposal, or if the proposal has a higher id for `r`'s current membership set.

Because Sigma operates by having all members send proposals to all other members in the membership set, Sigma is an all-to-all algorithm, and we also call it All-to-All (A2A) Sigma.

By itself, Sigma is a theoretical algorithm; one practical deficiency of Sigma is that it may generate inconsistent, transient views prior to converging on the correct final views. When a member receives an NE, it increments its identifier and outputs a view; if it receives a proposal for the membership set that it currently believes in and the proposal has a higher identifier, then the member adopts that identifier and outputs another view. This results in fast view agreements, but may sometimes produce superfluous (disagreed) views prior to this agreement. We denote the theoretical Sigma algorithm as Sigma_UD, where UD stands for "unlimited disagreement".

To remedy this deficiency, [39] suggests a design in which Sigma is used with a "filter" to prevent such problematic views from being delivered to GM clients. One possibility is to use a filter that delays view delivery until it is known that other members have come up with the same views. This is what all other existing GM algorithms do (see Section 4.2 and 4.4). Using such a filter would result in the same performance properties as those of other existing algorithms, requiring two or more rounds of message exchanges among group members in order to reach Agreement on Views.

Another possibility is to design filters that, when deployed with Sigma, would do better than existing algorithms. In particular, [39] suggests a simple filter, which we call LD (see Figure 4-1), that preserves the single-round worst-case performance of Sigma, and at the same time is claimed to be effective at weeding out inconsistent, transient views. Specifically, LD delays view deliveries until the latest proposals from all members agree on the correct membership set. This works, even though the ids of these latest proposals are not required to be the same, because after a group stabilizes, the membership sets are already the same, so the remaining task is to produce the same identifiers. Everyone sends their identifier and the member with the largest one wins. This maximal identifier in the worst case reaches everyone within one message latency. We denote the Sigma algorithm with LD filtering as Sigma_LD.

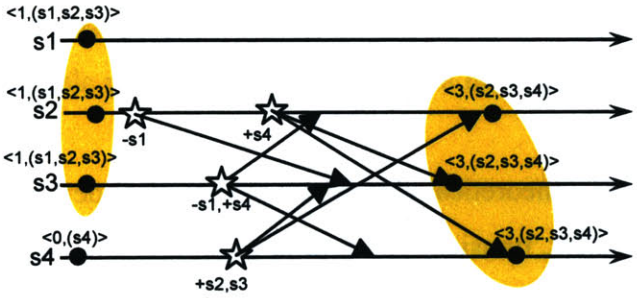

Figure 4-3. All-to-All Sigma_LD Example.

Example 4-1. Figure 4-3 illustrates Sigma_LD's operation in a scenario in which Sigma_UD would have produced a disagreement. As before, stars represent network events and filled circles represent view deliveries. There are two concurrent network events: s1 disconnects and s4 joins. Equipped with the LD filter, Sigma produces no

disagreements, and delivers view <3,(s2,s3,s4)> within one message latency after the final NE.

4.2 Moshe

As part of our comparative analysis, we compare Sigma with Moshe, an existing all-to-all group membership algorithm that is known to be practical. Moshe takes one round in the typical case, but may take one or more additional rounds for certain "out-of-sync" cases. Moshe is a group membership algorithm developed by Keidar, Sussman, Marzullo, and Dolev specifically for use in WANs [34]. It can be seen in some ways as a precursor to Sigma. Moshe is similar to Sigma in that each member runs the same algorithm and there are two event handlers: a member can either receive a network event from NS or a proposal from another member. Sigma preserves Moshe's client-server design, in which membership is maintained by a small group of dedicated membership servers on behalf of a larger group of processes. Sigma and Moshe are both optimistic algorithms.

In spite of their similarities, Moshe is different from Sigma in several ways. Unlike Sigma, Moshe does not reuse old proposals if they share the same membership set S with the newly forming view. In addition, Moshe implements extra synchronization to prevent disagreement and as a fallback measure when the optimistic case does not apply.

Moshe runs in two modes – Fast Agreement (FA) and Slow Agreement (SA). The FA mode is a single-round algorithm, which terminates successfully in the optimistic case – if every member s' $\in$ S receives new proposals from all members s $\in$ S for the same view. FA blocks if there is some pair of members s,s' $\in$ S, for which s does not receive such a proposal from s'. To prevent this livelock scenario, Moshe implements a blocking detection mechanism and switches to SA, the slow-path protocol, when these blocking cases are detected. Moshe's proposals include more information than Sigma proposals; this additional information is used to detect the "out-of-sync" cases that require the algorithm to switch to SA. The SA protocol involves one or more additional rounds of message exchange before the view identifier is determined and the view can be delivered. The reader is encouraged to consult [34] for an in-depth description of Moshe, as well as the detailed specifications that were used to implement Moshe for our research.

4.3 Leader-Based Sigma

The pivotal distinction between A2A Sigma and Leader-Based Sigma is that while A2A Sigma has all members both sending and receiving proposals, Leader-Based Sigma involves the selection of a leader that acts as an intermediary for communication among members. It is the duty of the leader to collect proposals from other members, construct a ready-to-install view, and share this view with the rest of the members, which they subsequently install.

```
Leader Based Sigma [Member r]

On receive ne_r(joins, leaves):
 prop[r].set ← prop[r].set ∪ joins - leaves
 let max_id = max{prop[i].id | i∈prop[r].set
                ∧ prop[i].set = prop[r].set}
  prop[r].id ← max(prop[r].id + 1, max_id)
  let L = leaderOf(prop[r].set)
  if (r = L) share_view ← true
  else DeliverViewIfReady(L)
  endif
  send proposal (r, prop[r].id, prop[r].set)
   to L (deliver immediately to self if r=L)

On receive proposal_r(s, id, set):
  prop[s] ← ⟨id, set⟩
  if (set = prop[r].set)
    if(id > prop[r].id)
      prop[r].id ← id
      share_view ← true
    endif
    ShareViewIfReady()
  endif

On receive view_r(s, id, set):
  prop[s] ← ⟨id, set⟩
  DeliverViewIfReady(s)

ShareViewIfReady() =
  if((share_view = true) ∧ LD())
    send view (r, prop[r].id, prop[r].set)
      to prop[r].set (deliver immediately to self)
    share_view ← false
  endif

DeliverViewIfReady(L) =
  if((prop[L].set = prop[r].set)
      ∧ (prop[L].id ≥ prop[r].id))
    prop[r].id ← prop[L].id
    deliver view ⟨prop[r].id, prop[r].set⟩
  endif
```

LD() is a filter used in the Sigma_LD algorithm:
LD()≡((∀i∈prop[r].set)prop[i].set=prop[r].set))
For Sigma_UD, **LD()**≡true.
* Each event handler is executed atomically.

Figure 4-4. Pseudocode for Leader Based Sigma.

Leader-Based Sigma (Figure 4-4) proceeds as follows. Upon receiving an NE, a member `r` updates its membership set `prop[r].set` and increments its `prop[r].id` as in the original algorithm. But instead of sending a proposal to all members that are relevant to the membership set, member `r` now sends a proposal only to the leader.

When the leader `L` receives a proposal from member `r`, it saves the proposal in `props[r]` and adjusts `prop[L].id` to be maximal, as before. Then, in addition to

installing the view itself, the leader also shares the view, by sending it to the rest of the members. Upon receiving a view from the leader `L`, member `r` saves it in its own `prop[L]` and conditionally delivers the view to its clients; the condition is the following: `prop[r].set` has to match `prop[L].set`, and `prop[r].id` has to be no greater than `prop[L].id`. This condition is also checked when member `r` receives an NE, because the NE may happen late after the leader already shared the latest view with `r`.

The leader is picked deterministically without any additional communication. For example, the leader can be chosen as the member with the largest process identifier in the current membership; this is the strategy we assume for the Leader-Based Sigma examples illustrated in Section 4.5. Other GM algorithms also use a strategy of picking the oldest member as the leader. In addition to such static strategies, some settings may benefit from a dynamic leader-selection strategy, which for example may account for connection qualities of different members.

Regardless of the leader-selection strategy, the leader may need to be changed when the current leader crashes or disconnects. Note that there may be transient periods during which different members have different perceptions of the membership, in which case members may select inconsistent leaders. Also, since the network environment is partitionable, it is possible for different leaders to serve a mutually exclusive subset of members.

Because the leader is a group member, in addition to its leadership duties, it also receives NE events. In leader-based Sigma_UD, when the leader receives an NE, it immediately sends itself a proposal, processes it, and installs the new view, as well as shares this view with the other member. This optimistic best-case scenario reduces the time required for the leader-based algorithm by one round.

Limiting disagreement in leader-based Sigma works similarly as in the A2A Sigma_LD, in that views are installed only when proposals from all the members agree on the membership set. In Leader-Based Sigma, however, the LD filter has been moved out of individual members' view delivery process, and into the domain of the leader. Specifically, the leader shares its view only when the LD condition is satisfied. Since a

view is installed only after the leader has shared it, all installed views thus meet the LD criteria, including views installed by the leader itself.

4.4 Horus/Ensemble

As part of our comparative analysis, we compare Leader-Based Sigma with an existing practical leader-based group membership algorithm – the one used by the Horus group communication system [51]. Because Horus has subsequently evolved into the Ensemble group communication system [28], we denote this group membership algorithm as Ensemble. The JGroups GCS [16] also uses Ensemble as its membership algorithm.

Ensemble was developed by Friedman and van Renesse as an implementation of weak virtual synchrony that does not block messages during view changes [27]. The specifications that we used for our implementation of Ensemble are detailed in [27]. We summarize the algorithm in this section.

The Ensemble algorithm assumes that

- The underlying environment provides reliable FIFO communication.
- The failure detector of process p_j will eventually generate a suspect event for any process p_i that stops receiving its messages due either to a lossy link or p_i crashing.
- If a message becomes stable, then every live process in the view will eventually learn about it. A message is stable if it is received by every live member of the view. A view is stable if all live processes in that view have received it.
- Each message is broadcast to all live processes in the suggested view of its invoking process.

The Ensemble algorithm proceeds as follows. Each view is associated with a contact, or leader. If this contact learns, during an already stable view, about another reachable contact with a smaller address, it sends the smaller contact a join request. When an available contact receives a join request or suspects that a member of its view has failed, it starts a view change. A contact is available if it is not already busy with a view change.

A view change is done by adding to the current view the newly joining processes and deleting faulty ones. Joining processes that are already members of the view are deleted

as well. Because such processes thought they were separated from the rest of the view, they may have refused some messages that were received by the rest of the view. They must therefore be eliminated before being allowed to rejoin.

The contact then sends this modified view as the "suggested view" to all the members of this view. Upon receiving a suggested view, a process adopts it, and sends an acknowledgement back to the contact. In the context of the Ensemble group communication system, processes would send this acknowledgement after sending all unstable messages from faulty processes to the contact that initiated the view change. This step is called "flushing", and the acknowledgement, which notifies the contact that this process has finished flushing and that all messages sent in the previous view have become stable, is called the "flushed" message. In our research, however, we study the membership algorithm in isolation, rather than in the context of a group communication system. Because the only messages in a membership algorithm have to do with view changes, the concept of flushing messages is irrelevant to our implementation. We therefore treat the "flushed" message as a process's acknowledgement of having received the suggested view.

Once the contact has received flushed messages from all the processes in the suggested view, it adopts the suggested view and sends the new view to the processes in the view, which adopt this view. On the other hand, if at least one member of the suggested view has failed and therefore was unable to send a flushed message to the contact, then the contact initiates a new suggested view that does not include this faulty member.

4.5 Example Scenarios and Discussion

In this section we consider the contrasting effects of using Leader-Based Sigma and A2A Sigma and discuss the tradeoffs involved. In Section 4.5.2, we compare Leader-Based Sigma to Ensemble. Finally, we give an example in Section 4.5.3 of Moshe's Slow Agreement algorithm, which is the main difference between Sigma and Moshe.

The scenarios in Figure 4-5 through Figure 4-9 show a system of four membership members: s1, s2, s3, and s4. Horizontal lines represent the passage of time. Filled circles represent installation of views. The views are shown in angle brackets; initial views are

on the left, and final views are on the right. Hollow stars represent network events (NEs). Hollow arrows represent ignored messages, and filled arrows represent messages that change state at the recipient. In Figure 4-5 and Figure 4-6, all arrows correspond to proposals. In Figure 4-7, Figure 4-8, and Figure 4-9, dashed arrows are proposals, and solid arrows are shared views.

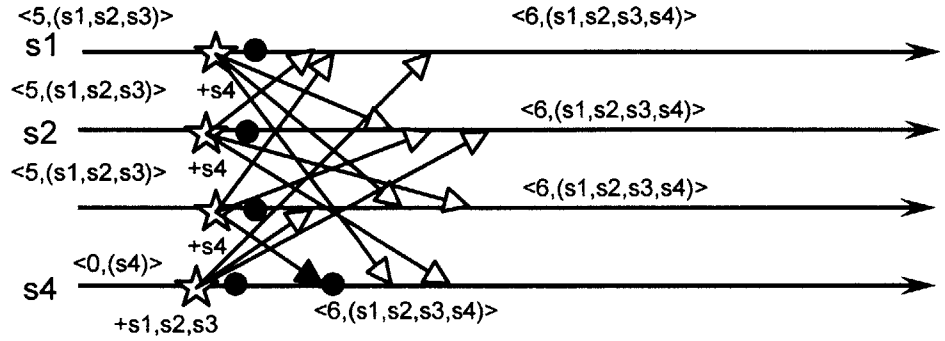

Figure 4-5. All-to-All Sigma, unlimited disagreement (Sigma_UD).

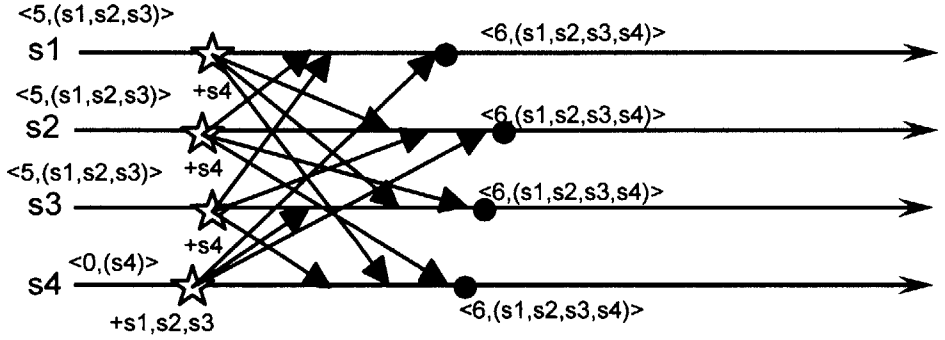

Figure 4-6. All-to-All Sigma with a filter for limiting disagreement (Sigma_LD).

4.5.1 LB Sigma vs. A2A Sigma

We first summarize the A2A versions in Figure 4-5 and Figure 4-6. According to Sigma_UD, members can install a view immediately when they receive an NE (Figure 4-5), or within one round of message exchange, when receiving a proposal with the same membership set, but higher-valued view id. In the example, member s4 first raises an NE where s1, s2, and s3 join its membership, resulting in view <1,(s1,s2,s3,s4)>, which it installs. Then, it receives a proposal from s3 with view <6,(s1,s2,s3,s4)>. Since the proposal offers a higher view id than its current one, member s4 installs the proposed view. In Sigma_LD (Figure 4-6), however, members must wait until proposals from all

the members relevant to the current membership set have matching membership sets. Once all these proposals have been received, the Sigma_LD members install the view.

Figure 4-5 and Figure 4-6 demonstrate the A2A nature of the message exchange, with each member sending proposals to the three other members. For just four members, there are already 12 unicast messages being sent at the same time (or 4 multicast messages to 3 members). With each additional member, the number of unicast messages increases by 2(n-1), where n is the new number of members. For n members, the number of messages being sent at the same time is n(n-1), or $O(n^2)$. A large-scale GM deployment can conceivably consist of tens of membership members; sending hundreds of messages in the network for just a single NE could lead to the various problems of contention, including network congestion, overflowing message queues at each member, and ultimately message loss. Message loss leads to retransmissions, which both slows down the membership algorithm and further increases the message overhead.

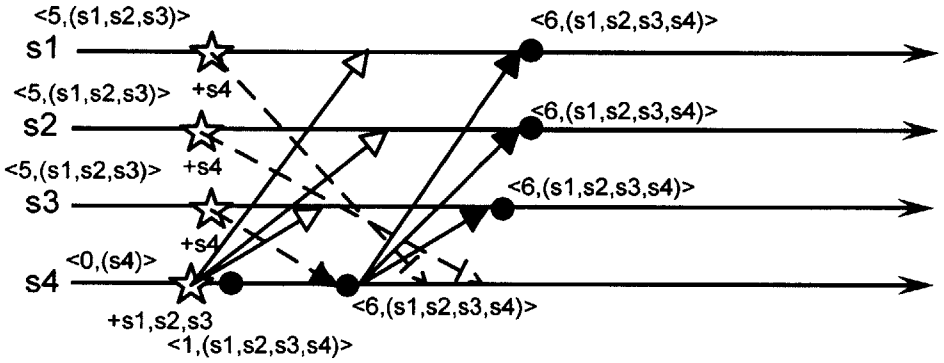

Figure 4-7. Leader-Based Sigma_UD.

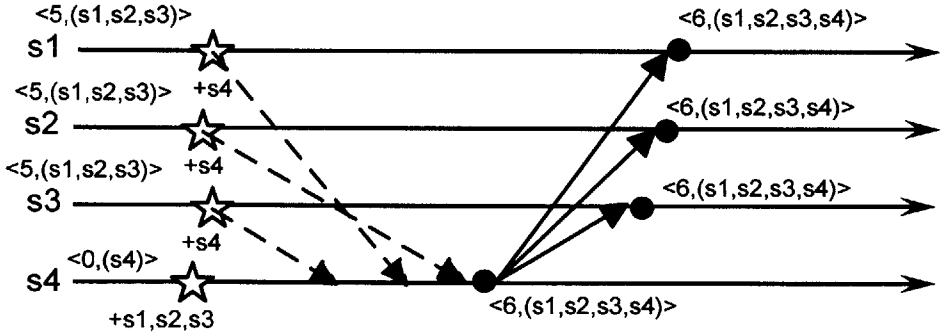

Figure 4-8. Leader-Based Sigma_LD.

Scenarios describing Leader-Based Sigma_UD and Leader-Based Sigma_LD are shown in Figure 4-7 and Figure 4-8, respectively. The leader selection criterion in this

scenario is the connected member with the largest id; when member s4 joins, it is chosen as the leader. When an NE is raised, a member forwards a proposal to the leader. The leader shares new views with the other members. Upon receiving a shared view from the leader, members install the view if its identifier is greater than or equal to their current perception of the correct view id. In Figure 4-7, after s4 receives an NE with s1, s2, and s3 joining, s4 constructs the view <1,(s1,s2,s3,s4)> and shares it with the other members. However, the other members have 6 as their proposal identifier; since this is greater than the view id 1 shared by the leader, the new view is ignored by the other members. Once leader s4 receives proposal <6,(s1,s2,s3,s4)> from s3, it corrects its perceived view id to 6, and shares the modified view, which is installed by the rest of the members. Note that in Leader-Based Sigma_UD, the leader shares a view without checking if all proposed membership sets match, and thus it immediately shares a view when an NE is raised. However, in Leader-Based Sigma_LD, similarly to A2A Sigma_LD, the leader waits until all proposals relevant to its current perception of the group membership have matching membership sets before sharing the view. Thus, in the latter case, the leader does not send any messages when an NE is raised.

Figure 4-7 and Figure 4-8 demonstrate how leader-based Sigma delays view installation by one communication step. In A2A Sigma, views are installed just one round after the last NE occurs in the worst case, and immediately in the best case (for Sigma_UD). In Leader-Based Sigma, views are installed only after the leader has shared the view. The leader shares views one round after the last NE event occurs in the worst case, and immediately in the best case. The members receive the shared views one message latency later, and thus Leader-Based Sigma's execution is always one message latency longer than A2A Sigma.

On the other hand, these scenarios demonstrate Leader-Based Sigma's reduction in message overhead compared to A2A Sigma: for n members, the total message overhead is O(n) in the worst case for Leader-Based Sigma_LD and in the common case for Leader-Based Sigma_UD. As described in Section 3.3, the worst-case message complexity for Leader-Based Sigma_UD is still $O(n^2)$, in the case where all n proposals have different id's and arrive at the leader in order of increasing id. This worst case,

however, is different from, and much less common than, the worst case of A2A Sigma_UD, which is, in fact, its common case, for example, as depicted in Figure 4-5.

By reducing the message overhead from $O(n^2)$ to $O(n)$ in the common case (or in the worst case, when using the LD filter) Leader-Based Sigma thus offers a significant reduction in network contention and message loss. Avoiding message loss means fewer retransmissions, resulting in a more efficient execution of the membership algorithm, despite a theoretically established extra round of message exchange. In addition, Leader-Based Sigma, like other leader-based algorithms, offers a parameter for controlling the quality of service of links over which messages are sent: the choice of leader. Because all messages are sent either to or from the leader, choosing a leader with the best resources and most reliable connections can give a performance edge to Leader-Based Sigma even in the worst case. In contrast, A2A's performance is sensitive to slow and lossy paths [14]. In this light, in certain practical networks, Leader-Based Sigma can offer better overall performance and scalability than A2A Sigma.

4.5.2 LB Sigma vs. Ensemble

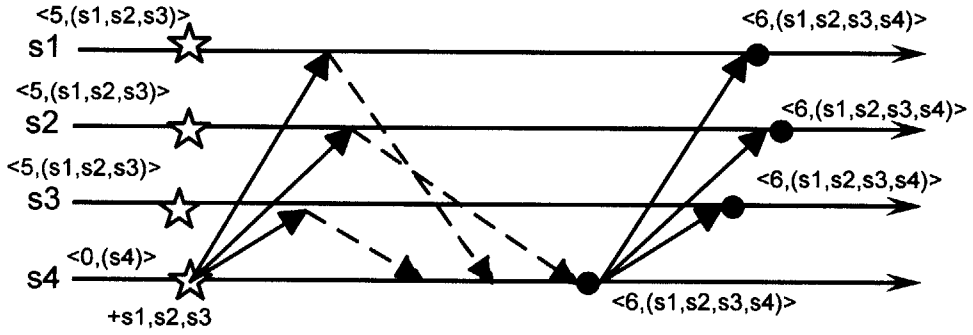

Figure 4-9. A standard leader-based GM algorithm like Ensemble.

Leader-based Sigma must also be considered in the context of other leader-based algorithms that have been studied and implemented in the past. Figure 4-9 demonstrates a communication pattern of Ensemble. As we have seen in Section 4.4, Ensemble is similar to Leader-Based Sigma_LD, but it is different in two ways. First, Ensemble's leader is the only member that reacts to network events. Upon learning of a network event, the leader sends a message to the other members asking them to send in their view proposals, delaying everyone's participation in the algorithm until this message from the leader

reaches them. This delay results in GM taking three communication steps, in contrast to the two steps taken by the Leader-Based Sigma_LD.

The second difference is that Ensemble's leader waits to receive new proposals with the matching view identifier and membership set from all the members before sharing the view with the members. In contrast, Leader-Based Sigma_LD can reuse an old proposal with a different identifier, as long as the membership set matches. Consequently, Leader-Based Sigma_LD's less rigorous filtering may avoid unnecessary delays that might occur in the traditional GM algorithms.

4.5.3 A2A Sigma vs. Moshe

In this section, we contrast A2A Sigma's operation with that of Moshe, by describing a scenario in which Moshe switches to its Slow Agreement protocol. Because Moshe's Fast Agreement causes the same message exchange patterns as described in Section 4.5.1 for A2A Sigma_LD, Moshe's main distinction is its Slow Agreement protocol. The asymmetric scenario in Example 3-1 is one of several cases that result in Moshe switching to SA, as depicted in Figure 4-10.

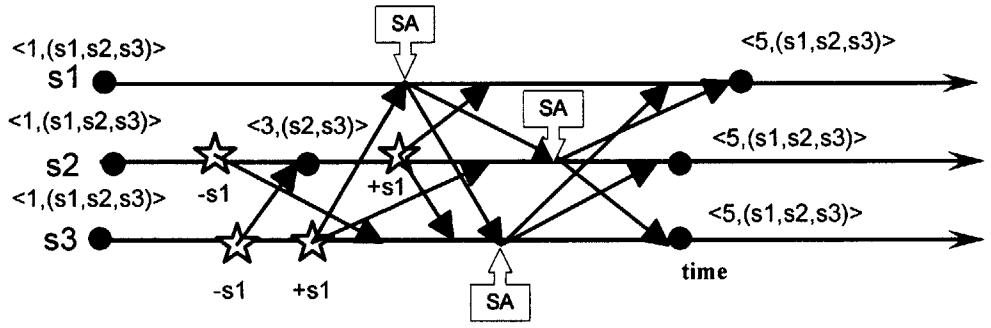

Figure 4-10. Moshe Slow Agreement Scenario.

A proposal sent by member X for a view with id=Y is denoted pX(vY). To recap, s2 and s3 receive ne(-s1) and send proposals p2(v2) and p3(v2), respectively, to each other, while s1 still thinks it is connected. Receiving p3(v2), s2 increments its viewid, and installs view <3,(s2,s3)>. However, s3 does not receive p2(v2) until later, when its membership set is old, and is therefore ignored. Meanwhile, s3 receives ne(+s1), adds s1 to its membership set, and sends proposal p3(v4) to s1 and s2. Receiving p3(v4) without an accompanying network event, s1 initiates SA, which sends proposals p1(v4) of type

SA to the other two members. Receiving p1(v4), s2 and s3 switch to SA, and each similarly sends SA proposals p2(v5) and p3(v5), respectively, to the other two members. Once received, agreement is reached, and all three install view <5,(s1,s2,s3)>. Thus, in this case, SA takes two extra rounds to reach agreement.

Chapter 5

Simulation

As one of the goals of this thesis, we present in simulation a performance analysis of Sigma's practical potential and the tradeoffs involved, before introducing Sigma into a real network. This chapter describes how we implemented the simulation. Chapter 6 describes the analysis results from the simulation, and in Chapter 7 we discuss the implications of these results. In all, we implemented and studied six algorithms in this simulation: the four variants of Sigma (A2A Sigma_UD, A2A Sigma_LD, LB Sigma_UD, and LB Sigma_LD), and the two practical algorithms, Moshe and Ensemble, for comparison. These algorithms have been described in Chapter 4.

5.1 Platform

The simulation was implemented using the ns-2 Network Simulator. Ns-2 is a library that provides functionality to simulate asynchronous communication channels, TCP, routing, and multicast protocols over wired and wireless networks [25]. The following ns-2 constructs were useful in implementing the simulation. All of these constructs are extensible, enabling customized implementations that accommodate the specific nature of the intended simulation. Figure 5-1 illustrates these constructs.

- **Nodes** represent the physical machines in the simulated network topology. Each Node gets assigned a unique address automatically by ns-2 at creation, and maintains a series of ports that serve as an interface to the network.
- **Links** correspond to the physical connections among Nodes. Ns-2 provides simulation tools for a variety of link types, ranging from point-to-point simplex- and duplex-link connections, to wireless and broadcast connection media. For our simulation, we used duplex, or bi-directional, links, each specified by two endpoint Nodes, link bandwidth, link delay, and queue type. Link delay represents the time required for a packet to traverse a link. The amount of time required for a packet to traverse a link is defined to be s/b+d where s is the packet size, b is the speed of the link in bits/sec, and d is the link delay in seconds. Queues represent buffers where

packets are held as they arrive. When the queue fills up, it must decide which packets to drop in order to prevent overflow. A queue is defined by the particular dropping strategy that it uses; for example, drop-tail (FIFO) or random early drop (RED).

- **Agents** and **Sinks** correspond to processes that execute specific transport-layer protocols. Ns-2 implements a variety of Agent and Sink types, including a comprehensive library of TCP variants. An Agent and Sink pair is associated with a pair of Nodes, or more precisely, a pair of ports that define a link. The Agent runs on a port on the source Node , and the Sink runs on a port on the destination Node. The Agent sends packets to the latter port in behalf of the source Node, and the Sink receives these packets in behalf of the destination Node.
- **Packets** are the ns-2 representation of this fundamental unit of exchange in network communication. Each new Packet is first created, or allocated in memory, and then its header information initialized with the appropriate values. All Packets used in this simulation have an IP header, among others, in which must be set the source and destination Nodes' address and port. Packet headers also include such information as a timestamp, TTL, and any user-defined application- or protocol-specific fields. Once the Packet header is defined, any relevant application data is attached, and the specified Agent sends the Packet to its destination.
- **Timers** enable control over the scheduling of specific events. Timers are set to expire after a specified delay. Upon expiration, a Timer will execute application- or protocol-specific procedures or events.
- The **Application** construct enables the definition of applications, overlays, or traffic generators that sit on top of the transport-layer Agents.
- **AppData** represents application-specific data that Packets may transport. If a Packet arriving at a Node contains any AppData, ns-2 extracts the AppData and automatically passes it to the Application layer for processing.

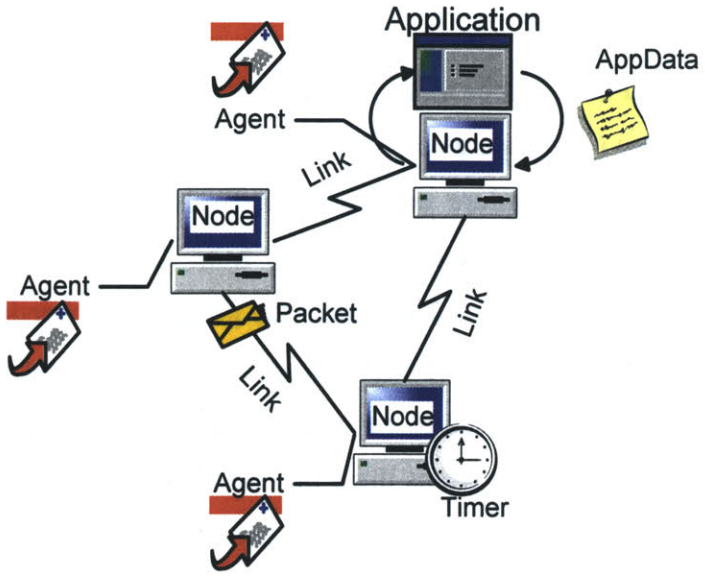

Figure 5-1. Elements of the Ns-2 Platform.

The implementation of an ns-2 simulation involves the use of two languages; ns-2 is written in C++ with an OTcl interpreter as a front end. The constructs described above, and in fact the majority of ns-2's constructs, are implemented in two parallel class hierarchies, one in C++, and the other in OTcl. This dual design, as described in [25], is intended to apply the characteristic properties of each language to the two different aspects of implementing a network simulation: protocol implementation and topology configuration.

- C++ offers optimal run-time speed for an object-oriented language, while requiring extra time for reconfiguration and debugging, because C++ code needs to be compiled before it is run. C++ is therefore ideal for defining new protocols and applications, which involve detailed processing of bytes, packet headers, and implement algorithms that run over large data sets.
- OTcl, on the other hand, optimizes reconfiguration time, but it is much slower than C++ at run-time. Thus, OTcl is ideal for setting up and configuring the network topology, which only needs to run once, at the start of the simulation.

Thus, network topology is more easily configured using OTcl, while new network protocols are more easily implemented using C++.

5.2 Implementation

Our simulation design follows the model described in Chapter 3, in which the group membership service (GM) operates in partnership with a notification service (NS), which interfaces GM with the failure detector and reliable FIFO (RFIFO) communication service. NS learns about network instabilities from the failure detector, and interprets them into network events (NEs) that change the group's membership. NS communicates these NEs to GM, which adjusts its view according to the resident membership algorithm and may require view proposals to be sent to other membership members. GM passes these proposals to NS, which wraps them into network packets and sends them, through the simulated network, to the destination members. Similarly, NS receives view proposal packets from other members, unwraps them, and passes the proposals to GM.

In a real world situation, a GM implementation would preferably follow the more scalable two-tier architecture described in Chapter 3. Each membership server would be a dedicated machine that serves a local group of distinct client machines. For simulation purposes, however, we consider a membership server as representing an abstract set of local clients, by treating each simulated server as if it were also a client. We therefore refer to them simply as "members".

The task of implementing the group membership simulation consisted of five major components: (1) a design for modeling network events and failure detection, setup and configuration of the network topology; (2) implementation of a FIFO communication service for message transport through the network; (3) a notification service (NS) that communicates network events to the group membership algorithm; (4) the membership service (GM) module that implements the group membership algorithms that we simulated; and collection and analysis of data from the simulations to enable the comparative study of Sigma that we describe in Chapters 6 and 7.

5.2.1 Modeling Network Events and Failure Detection

We model network events, simulate the failure detector, and derive the network configuration by using existing trace data collected over real wide area networks (WANs). Such traces provide a wealth of authentic network activity, and serve as a

valuable resource towards creating realistic network scenarios in simulation. The trace data set that we used comes from the Resilient Overlay Networks (RON) project [9, 11]; this trace was originally used in [29]. RON traces offer information about the connectivity between each pair of participating members; this property has motivated our use of the traces to simulate a NS failure detector.

RON is an application-layer overlay set on top of the existing Internet routing infrastructure that improves the quality of service in a participating network of nodes. This is achieved through a distributed process of network outage detection and fast recovery by re-routing packets to avoid faulty network paths. RON thus acts in large part as a failure detector. Nodes monitor the connectivity amongst each other by sending probes to other nodes. To probe, each RON node independently picks a random node j, sends a packet to j, records this fact in a log, records if there was a response, and then waits for a random time interval between one and two seconds before probing again. If there is no response, it is considered to be a loss, and the offending path earns a "point" towards qualifying as an outage. Interpreting these losses into path outages is an application-dependent procedure; for example, the original RON study [11] used 4 consecutive losses as the qualifying parameter.

An excerpt of a RON trace is shown in Appendix A. Each line is a distinct entry, and each entry consists of seven fields [10]:

- **source**, the originator of the probe.
- **dest**, the destination to which the probe was sent.
- **ron**, a flag to denote the RON link type. This was always 0 throughout the RON traces that we used, which means that probes were sent directly on the Internet. The other options are 1, in which the RON link is latency optimized, and 2, in which the RON link is a loss optimized path.
- **send1**, the time at which the source originally sends the probe to the destination.
- **rec1**, the time at which the probe was received on the interface at the destination.
- **send2**, the time at which the probe was sent back to the sender by the destination.
- **rec2**, the time at which the probe was received at source's interface.

The clocks on the RON servers were only roughly synchronized; thus, the time fields are accurate only in relation to other time entries on the same machine. For example,

send1 and rec2 are compatible for relative analysis, because both times were computed by the same physical clock. In contrast, send1 and send2 cannot be compared, because they were each computed on different machines, and thus different physical clocks.

A loss is represented in the trace by a zero value in any or all of the time entries. It is not clear why some disconnect probes list zeroes in all the time entries, while others list zeroes only in the rec1 and send2 fields. We assume that these differences are due to variations in logging configuration among different nodes.

The particular RON trace used in our simulation was also used and analyzed by [29]. In this trace, there are sixteen nodes that are spread out across the United States and Europe. Each pair of nodes probe each other once every 22.5 seconds on average. This trace contains continuous probing for the two-week period from August 2 through August 16, 2002. A detailed description of the properties of this trace, including loss rates, the number and duration of partitions, and the degree to which communication is symmetric and transitive can be found in [29].

5.2.2 Notification Service

The notification service (NS) was implemented as an ns-2 Application object called NESvcTrc. It is designed as an event handler driven by network events that are interpreted from the RON trace. The NS process at each member parses the same RON trace, line by line; each member ignores all traces except those with its own member ID in the source field. The simulation follows the timing of events as specified by the RON trace data; NS keeps a timer, NSTimer, for this purpose. When the simulation starts, NSTimer initializes by reading the first line of the RON trace. The time interval specified by rec2-send1 is set as the delay after which the first network event is scheduled to occur; NSTimer is set to expire at the end of this time interval.

Because of the existence of probe statistics in which send1 and/or rec2 could be marked as zero, deriving a valid time delay interval for which to schedule events from such probes in the manner described above was impossible, because it would yield a negative value for the time delay. To solve this problem, we implemented a feature in NS that computes a running average of computed time delays across all encountered probe

lines. For probe lines in which the time delay could not be computed from the trace statistics, this average delay value was used instead.

The expiration of NSTimer triggers the execution of the event handler, which reads and processes the next probe line in the RON trace. The event handler ignores the probe line if the source ID does not match its own, and resets the timer for the rec2-send1 time delay given by this line. If the source ID does match, then NS interprets the probe data to determine whether it should be treated as a leave, join, or heartbeat network event.

These events are forwarded to other currently connected members, and communicated to the membership service, after a Sensitivity to Disconnects (SD) delay (see Section 5.2.2.3). If a join or heartbeat event occurs within the SD after a leave event for the same member, the leave event is cancelled. Similarly, if a leave event occurs within the SD after a join event for the same member, the join event is cancelled. When an event is cancelled, all forwarding scheduled for this event is cancelled as well.

An event that is still in effect when the SD elapses is then forwarded to other currently connected members and passed to the membership service. NS sends and receives messages in behalf of itself as well as the membership service, doing so by passing messages to, and retrieving messages from, the FIFO communication service. As an interface to FIFO, NS maintains two sets of TCP Agents; one set for sending messages and the other for receiving messages. Agents in the former set are implemented by GCSAgent and in the latter by GCSSink. Each set contains one Agent for every other node, each Agent therefore representing a single link to (in the former set) and from (in the latter set) this node.

5.2.2.1 Events

NS learns that a member has left whenever a line is encountered in which any or all of the time entries in the RON trace are zero; in other words, a lost probe. When a leave event occurs, NS removes the leaving node from its list of connected members and notifies the membership service of the leave.

A probe line is treated as a join if the destination node is not listed among NS's list of connected members and the probe is not a loss (see Section 5.2.1). If the probe is a loss,

then NS ignores this line and moves on. When a join event occurs, NS adds the joining node to its list of connect members and notifies the membership service of the join.

Heartbeat events occur whenever a non-loss probe line is encountered in which the destination node is already listed in the NS's list of connected members. A heartbeat event about member M cancels any scheduled leave events about M, because a heartbeat removes all doubts that M is indeed connected. Otherwise, nothing further is done during a heartbeat event.

5.2.2.2 Network Event Forwarding

By itself, the RON trace is not equivalent to an NS trace. The NS failure detector must ensure that NS quickly propagates information to different GM nodes. In contrast, as described in Section 5.2.1, a RON node chooses a single target node randomly during each iteration of the RON probing process, and consequently, each node ends up probing every other node only every 22.5 seconds on average. Because a node requires such a long time-frame to learn about its connectivity with all other nodes, the RON trace by itself could not be used in place of NS.

To use the RON trace in the NS simulation, we had to implement a solution in which each NS forwards network events to the NS's of other currently connected members, before NS communicates these events to its GM. In this way, the NS's of other members learn of events and communicate them to their GMs at roughly the same time. This implementation corresponds to one possible implementation of an NS service.

5.2.2.3 Sensitivity to Disconnects

Another issue in the implementation of NS was the idea of being able to adjust its sensitivity to short-term network instabilities. For most applications, if a member disconnects and then reconnects shortly afterwards, it is better to just ignore these two network events, rather than produce two additional views. Because they are short-lived, such transient events cause unnecessary view changes, which lead to wasteful processing and interrupt the application's normal operation.

Transient events occur frequently in wide-area networks and are difficult to distinguish from permanent events, because this requires knowledge of the future;

applications have no way to distinguish a temporary departure from a permanent leave at the time of a node's disconnection [43]. Without such a distinction, GM delivers a view for each transient event, the same as it would for a permanent event. It is important to minimize view changes due to transient events, because at the application level, each view change is associated with costly reconfigurations.

To this end, we implement a Sensitivity to Disconnects (SD) in the NS. Each member's NS maintains a timer for every other member. When member r encounters a disconnect for member u in the RON trace, it sets timer u to expire in SD seconds. When timer u expires, member r forwards a "leave" network event to all other members (except member u), and communicates this event to its GM. If member r discovers that member u has reconnected according to the RON trace before timer u expires, then timer u is aborted, and the initial disconnect and subsequent reconnect are ignored. Likewise, when member r encounters a reconnection to a previously disconnected member, it forwards this information to the members to which it is currently connected.

Different applications would have different requirements for the time interval between the occurrence and resolution of such transient network instabilities that they can tolerate without having to create a new view. Applications have different definitions of what "permanent" means; SD enables them to adjust the granularity of events to be perceived as permanent. By appropriately configuring SD according to their needs, applications can avoid unnecessary reconfiguration overheads by filtering out transient events. We believe that including this sensitivity parameter in our study was important to obtain more comprehensive results that account for a variety of applications.

5.2.3 Group Membership Service

The Group Membership (GM) service is situated on top of NS. The interface and interaction between GM and NS is quite simple, as shown in Figure 5-2:

- NS communicates network events to GM,
- GM passes view proposal (and shared view, if leader-based) messages to NS for transport to other members, and
- NS passes view proposal messages received from other members to GM.

Our design for the GM module was flexible enough to enable the implementation of all six GM algorithms[2] within the same framework and minimal, if any, modifications outside the GM module. In so doing, we have reinforced the idea, first proposed by the designers of Horus [27], that a modular design can enable different group membership algorithms to be plugged into a single group communication infrastructure. Such modularity adds flexibility to the group communication paradigm, leaving the choice of group membership algorithm to the application, depending on its needs.

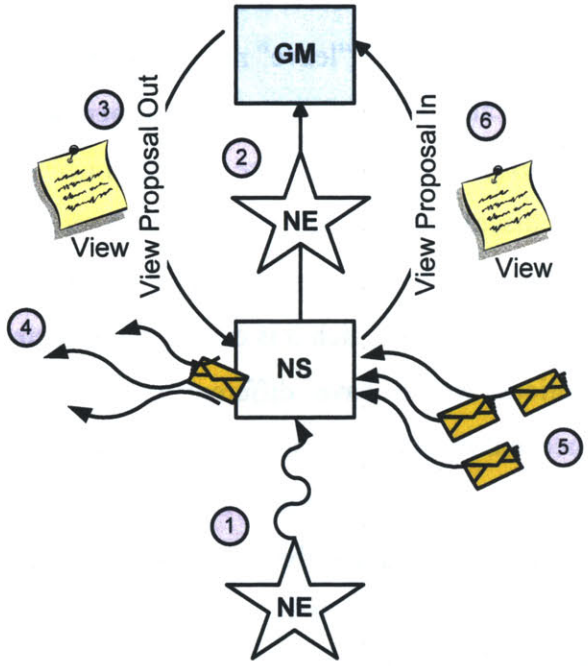

Figure 5-2. Interface between NS and GM. (1) NS receives network events (NE) and (2) passes them to GM. (3) GM forms view proposals to send to the GM's of other currently connected members, and passes them to NS. (4) NS packages the view proposals, and sends them out using reliable FIFO. (5) View proposal messages of other members are received by NS, which unwraps them and (6) passes the enclosed view proposals to GM.

We establish a multipurpose data structure **View** of the form <id, set> where id is an integer that specifies the view identifier and set is an array of values that uniquely identify member Nodes. The View data structure was used not only to represent the official view maintained by a member, and in the leader-based case, shared by the leader, but also as the view proposals that GM sends to the GM's of other members. This minimal specification of a view is sufficient for Sigma and Ensemble, as well as many

[2] A2A Sigma_UD, A2A Sigma_LD, Moshe, LB Sigma_UD, LB Sigma_LD, and Ensemble

other GM algorithms, and it was extensible in the case of Moshe, which maintains additional information in its views.

We implemented the six group membership algorithms described in Chapter 3 straightforwardly from their pseudocode specifications. In our implementation, we model just one group; this implementation can trivially be extended to multiple groups by associating different member processes on a single physical node with distinct group identifiers. Each server in our implementation is considered to represent its clients, and therefore each server is also treated as a client. In other words, the servers themselves are the members that connect to and disconnect from the group. The process of installing a view is represented by writing information about the view to a file.

We model only involuntary network events in our implementation: when members disconnect, they do not know that they have disconnected; therefore, when they reconnect, they do not know that they have reconnected. In doing so, we restrict our study to two different failure types – crash failures, and the transient, usually asymmetric, disconnects due to network congestion or localized link outages.

In contrast, we do not study voluntary joins and leaves; the reliable FIFO guarantees of the communication service (see Section 5.2.4), which we implement using TCP, mean that agreement on views always occurs for voluntary network events, since join and leave requests can be addressed and reliably delivered to all members.

5.2.4 Communication Service

The reliable FIFO communication service that GM assumes is straightforwardly implemented by the ns-2 TCP infrastructure. We implement two new classes, GCSAgent and GCSSink, as subclasses of TcpAgent and TcpSink, respectively, to interface the group membership simulation to this infrastructure. Specifically, GCSAgent implements an association between a TcpAgent and an instance of the NESvcTrc application. GCSAgent wraps View objects into proposal packets before sending them with its underlying TcpAgent. We implement a subclass of AppData, called GCSAppData, to hold the View data within the packet. On the receiving end, GCSSink implements a similar association, between TcpSink and NESvcTrc. GCSSink invokes ns-2's "process-

data" command to unwrap the GCSAppData from received packets, so that NESvcTrc can pass it on to the group membership service.

5.2.5 Network Topology Configuration

In our simulation, a member is represented as an ns-2 Node object, which communicates with other Nodes using TCP. Each node is physically connected to every other node by a duplex link with a bandwidth of 2Mbps and a propagation delay uniquely derived for each link from the RON trace. Each node acts as both a TCP agent for sending messages, and a TCP sink for receiving messages.

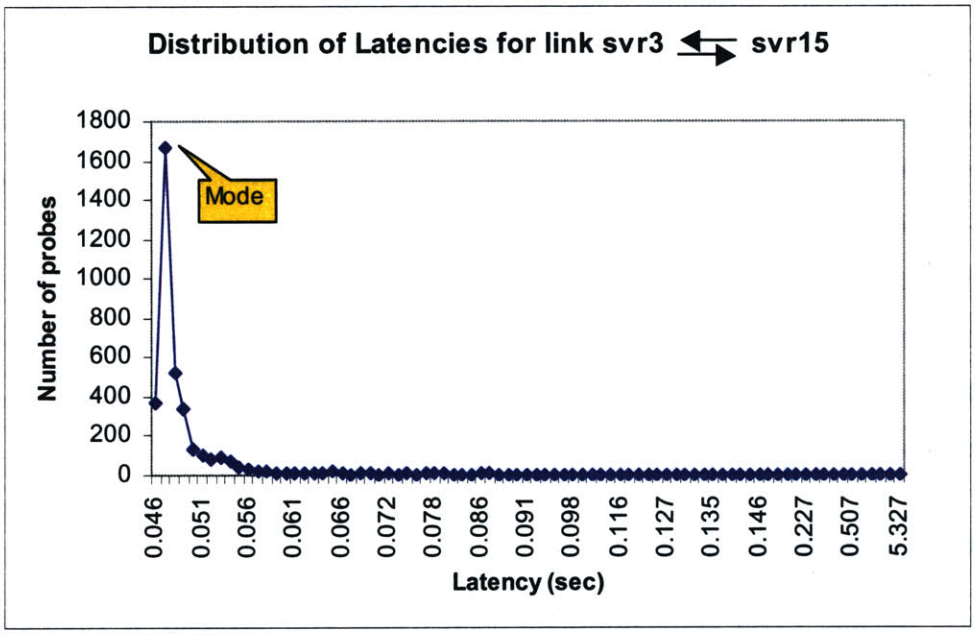

Figure 5-3. Example distribution of latencies. This particular distribution is for the link between node #3 and node #15.

Propagation delay for each duplex link was assigned on an individual basis based on our analytical investigation of the latency distributions that occur in the RON trace between each pair of nodes. For each source/destination pair (A,B), all entries in the RON trace for which A is the source and B is the destination were extracted. The round-trip latency was calculated for each entry as the difference between the time at which the packet was received from the destination at the source's interface (rec2) and the time immediately before the source originally sends the packet to the destination (send1). This round-trip latency was divided in half, and then added to the cumulative list of latencies

collected for this particular link. The latencies in this list were rounded to the nearest thousandth, and the mode, or the most frequently occurring latency, was chosen to be the propagation delay for this link. The reason why the mode was chosen instead of the mean is because the range of latencies could be quite large, as shown in Figure 5-3. Occasional periods of congestion could greatly increase the propagation delay for a given link, albeit temporarily, perhaps by one or more orders of magnitude. In the process, such extreme outliers could skew the average, and thus misrepresent the actual latency of the link.

On the other hand, the minimum occurring latency might have been most accurate measure in terms of its ability to capture the physical limitations of the link. However, if this minimum latency seldom occurs, then assigning such a propagation delay to the link is unrealistic, and perhaps too optimistic for a worst-case analysis. It is interesting to note that, as demonstrated by Figure 5-3, the mode for every single link that we analyzed was either the minimum occurring latency, or within one thousandth of the minimum.

The network was configured in ns-2 by an OTcl script, such as the one shown in Appendix B. This script was built in an automated fashion externally to the simulation, by a C++ standalone program that we wrote. First, a list of all the participating RON servers was extracted from the RON trace. This process parses the RON trace data to count the number of distinct servers participating in the trace, and assigns node ID's based on the corresponding distinct machine identifiers found in the trace. Even though the simulation was run numerous times with varying parameters and membership algorithms, the same network configuration was used at all times. Thus, the OTcl file was built once and then reused throughout the study.

First, a Node is created corresponding to each server. An instance of the NESvcTrc application that defines our implementation of NS is also created for each server. Next, the linkages and communication agents are configured. For each link, a GCSAgent and GCSSink are created. The GCSAgent is attached to the source Node for that link, while the GCSSink is attached to the destination Node for that link, using "attach-agent". Before the GCSSink is attached, the NESvcTrc object associated with the destination of the link is set as a parameter of the GCSSink. Afterwards, the GCSAgent is attached to the NESvcTrc object associated with the source of that link.

There are n*(n-1) such linkages built for n nodes. Once this process finishes, the script starts up each individual NESvcTrc Application instance, and begins the simulation. The simulation preserves a completely distributed network model: each member runs its own FIFO, NS, and GM components locally.

5.2.6 Analysis methods

We describe in this section the functionality that we implemented for analysis in the simulation environment. Data about the simulation is collected into data files. These data files are then interpreted by analytical functions that we have implemented to measure the amount of agreement and disagreement, the time it takes for the GM algorithm to deliver a view, and the message overhead.

5.2.6.1 Data file format

As mentioned in Section 5.2.3, we represent the act of installing a view by having the member write the view and related information to a file. Each member keeps its own data file for this purpose. We collect the following information in these data files, for each view installed:

- Time the NE that resulted with the view was raised;
- Time at view installation;
- View ID;
- View Members;
- Installation source flag – indicates whether the view was installed immediately after an NE is raised (N), or after receiving a view proposal (R).

For Moshe, we also collect data relating to whether the view was installed through fast agreement (FA) or slow agreement (SA).

In addition, members collect data on message overhead during the course of the simulation. At the end of the simulation, each member writes its message overhead total to its data file. Members running Moshe also collect message overheads separately for FA and SA runs, and write this additional data to their data files.

5.2.6.2 Agreement/disagreement

We calculate agreement and disagreement both in terms of raw numbers and percentages. Views in agreement are those for which the view id is the same for the same membership set; all members listed in the view must have installed this view. We define a disagreement as a view id for which there is more than one nondisjoint membership set with this same view id (see Definition 3-1). To obtain percentages we divide the raw number of agreed or disagreed views by the total number of views.

5.2.6.3 GM Latency

Towards studying exactly how much more efficient Sigma is than Moshe and Ensemble, it is necessary to measure how long it takes, in seconds, for the membership algorithm to execute – in other words, how much time elapses between the time an NE occurs, and the time that the member installs the view reflecting that NE. The timestamps that we obtain from ns-2 are universal time, because at each member, we get them by calling Scheduler::instance().clock() rather than using a local timer. Thus, to analyze GM latency, we calculate LVT-LNT for each view, where LVT is the latest time at which the view was installed, across all members, and LNT is the latest time at which the last NE was received before installing this view, across all members. For the analysis, we calculate the average LVT-LNT, its standard deviation, minimum, and maximum over all the views for a given SD. We plot these values as a function of SD.

5.2.6.4 Message overhead

We calculate average message overhead by summing up the number of messages sent by all the members and dividing by the number of members. For Moshe, average message overhead is also broken down into message overhead due to FA and SA. Message overhead is measured per total simulation run, not per round or per execution.

Chapter 6

Performance Analysis

Using the simulation that we implemented as described in Chapter 5, we have conducted a performance analysis of Sigma. Our performance analysis consists of two studies – the first study, denoted as A2A, presents an evaluation of All-to-All Sigma in comparison with Moshe; the second study, denoted LB, evaluates Leader-Based Sigma in comparison with Ensemble.

For each of these two studies, we ran two groups of simulations. Each simulation parses one million lines of the RON trace, equivalent to roughly 48 hours. Group RON1 simulated the first million lines, and group RON2 simulated the second million lines. The results that we present in this section are reinforced by the fact that they are consistent across both groups of simulations.

In each group, we simulated twenty-five sets of simulations for each of the three membership algorithms: Sigma_UD, Sigma_LD, and Moshe for our A2A study; and Leader-Based Sigma_UD, Leader-Based Sigma_LD, and Ensemble for our LB study. Each set varies the Sensitivity to Disconnects (SD) from 0 to 120s, at 5s intervals. Members start with viewid = 0 and all possible members in the membership set.

We distinguish the A2A and LB algorithms into separate studies, rather than directly comparing Sigma with Leader-Based Sigma, for several reasons. First, it is natural comparatively evaluate Sigma in the context of Moshe, because Moshe is also an A2A algorithm; similarly, it makes sense to evaluate LB Sigma in comparison with Ensemble, which is also a leader-based algorithm. Secondly, our simulation does not allow us to directly study the tradeoffs between A2A and LB, because a) we do not simulate lossy links, and b) our simulations are not run in the context of a larger encapsulating public network. Because lossy links and network congestion are major factors motivating the use of LB instead of A2A, our simulations would not adequately reflect the benefits of using LB. In addition, Leader-Based Sigma is not intended as an unequivocal replacement for Sigma, or vice versa. The choice between A2A and LB depends on the application, group size, and quality of the network environment.

For each simulation, we measure number and duration of views, agreement, disagreement, GM latency, and message overhead. These measurements consider all the members in relation to each other, rather than in isolation. We count a set of views as one distinct view if their <id, set> pairs are the same.

6.1 All-to-All Study: Sigma vs. Moshe

In this section, we present the results of the A2A study, in which we compare Sigma_UD, Sigma_LD, and Moshe. As a brief overview, we make the following observations.

- The total number of views is similar for all three algorithms, following a distinctively exponential decrease with increasing SD. Correspondingly, the duration of views increases in a near-exponential trend with increasing SD – with no SD (SD = 0 seconds), views change every 15 seconds; but with 60 seconds of SD, views remain unchanging for as long as four hours. This is consistent with intuition, because a longer SD hides the short, transient outages that dominate the trace.
- Sigma_LD matches Moshe's near-perfect agreement levels, with forty-five out of fifty data points achieving 99-100% agreement, and zero disagreement. Without the benefits of a limiting disagreement filter, Sigma_UD produces many non-agreed views, resulting in lower agreement percentages and higher disagreement percentages.
- Raw numbers of agreed and disagreed views, however, show the differences in percentage between Sigma_UD and Sigma_LD to be subtle – the raw trends for all three algorithms overlap and follow a similar exponential decreasing pattern. Because of the exponential nature of the raw trends, an analysis of percentages is skewed at longer SD's, due to the exponentially decreasing total number of views.
- Moshe produces zero disagreement, Sigma_LD produces negligible disagreement for 0-10s sensitivity delays and zero disagreement thereafter, and Sigma_UD produces zero disagreement for SD $\geq$ 60 seconds.
- Sigma_LD is faster than Moshe by an average of 30ms and by as much as 260ms. On average, and in the worst case, Sigma_LD is slower than Sigma_UD by the average link latency of the network topology.

- Both Sigma_UD and Sigma_LD operate with less message overhead than Moshe. To be precise, the message overhead of Sigma is roughly half that of Moshe.

6.1.1 Number and Duration of Views

Figure 6-1 shows the total number of views for the A2A algorithms -- Sigma_UD, Sigma_LD, and Moshe – to be similar in RON1. The number of views is an exponentially decreasing function of SD (Figure 6-1(a)). Figure 6-1(b,c,d) groups together data points sharing the same order of magnitude, in a piecewise analysis. The overlapping trends in these graphs suggest that the total number of views is similar for all three algorithms, with the following exceptions: Sigma_UD delivers about two thousand more views than either Sigma_LD or Moshe at SD = 0s. For SD > 0s, Sigma_UD performs similarly to Sigma_LD and Moshe. The results are reproduced for RON2 and are shown in Appendix C in Figure C-1. The only difference in RON2 is that Sigma_UD delivers more views than Sigma_LD or Moshe for SD ≤ 40s, instead of just at SD = 0.

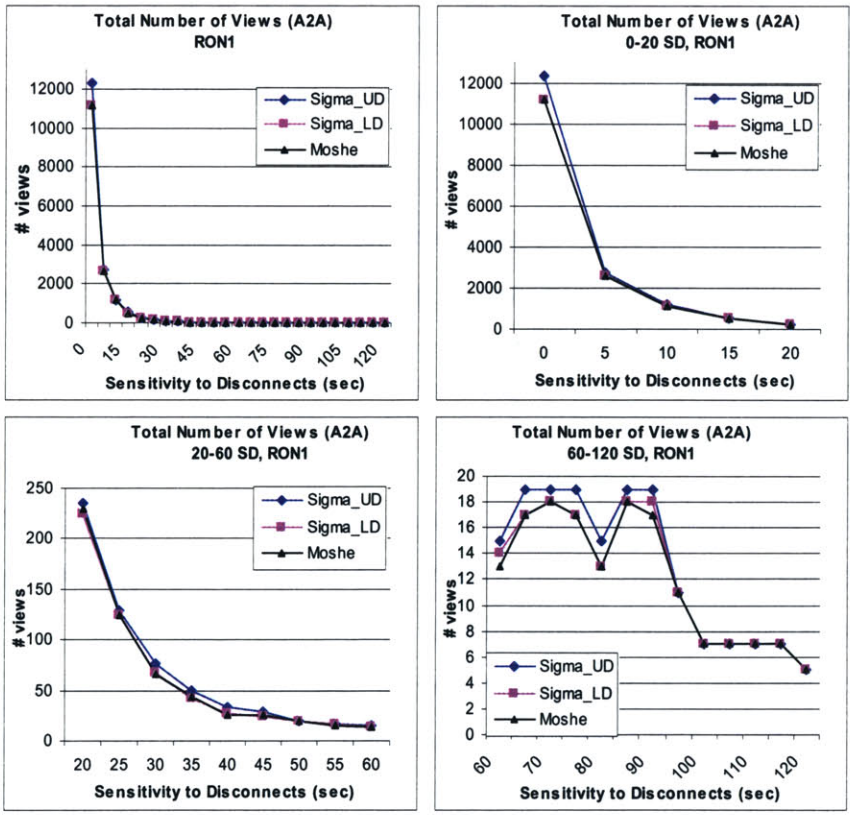

Figure 6-1. Total Number of Views, RON1. All-to-All Sigma vs. Moshe. (a) Overall picture (b,c,d) Piecewise analysis.

View duration, measured as the average number of seconds that each view lasts (seconds per view), is shown in Figure 6-2. We observe view duration to increase quickly with increasing SD. At the shortest SDs, view duration is on the order of seconds: views last 15s at SD = 0s, and views last 66s at SD = 5s. With SD = 35s, views last as long as an hour; with SD = 60s, views last four hours; by SD = 120s, views last for ten hours. The exponential decrease in number of views appears to be directly related to the increase in view duration with increasing SD.

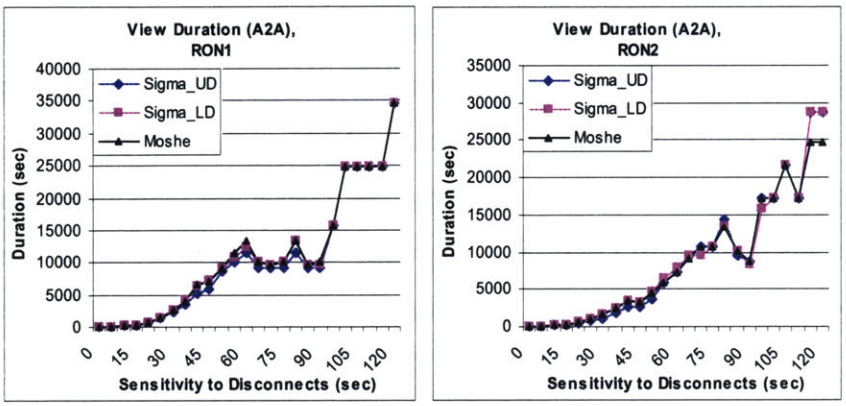

Figure 6-2. Duration of Views, All-to-All Sigma vs. Moshe (a) RON1 and (b) RON2.

The trends are very similar for the three algorithms, except that in RON2 Sigma_LD's views last 1.4 hours longer than Moshe's and Sigma_UD's views for SD = 115s and SD = 120s. This discrepancy is not reproduced in RON1, during which the trends overlap consistently for all SDs.

6.1.2 Agreement

Figure 6-3 compares the percentage of views in agreement for the A2A algorithms as a function of SD. The trends for the RON1 trace are shown in Figure 6-3(a), while the trends for RON2 are shown in Figure 6-3(b).

These trends show similarity between the agreement performance of Moshe and Sigma_LD. Moshe is most consistent, as expected, maintaining 100% agreement for all SDs in RON1, and for all but SDs 115 and 120 in RON2 where its performance drops uncharacteristically to 85%. Out of fifty total, forty-five of Sigma's data points represent 99% agreement or greater. All but one of the remaining five represent 95% agreement or

greater. The one other data point is 91% agreement at SD = 55s. For SDs ≥ 60s, Sigma_LD consistently achieves 100% agreement in both RON1 and RON2. In RON2, Sigma_LD also achieves 100% agreement for 25s ≤ SD ≤ 40s in RON2.

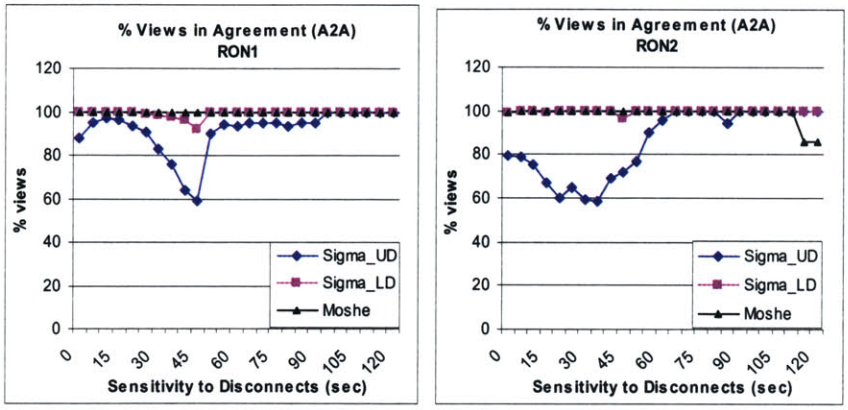

Figure 6-3. Percentage of Views in Agreement. (a) Trends for trace RON1 and (b) Trends for trace RON2. Both (a) and (b) demonstrate that while Sigma_UD offers unpredictable performance between 60 and 80% agreement for SD < 60s, Sigma_LD always performs close to or as well as Moshe in achieving agreement. Also of note is that for SD ≥ 60s, Sigma_UD consistently matches Sigma_LD and Moshe in achieving 100% agreement, or close to it.

Sigma_UD demonstrates lower agreement percentages for SD < 60s than both Sigma_LD and Moshe. Within this range, Sigma_UD's best performance occurs during the RON1 trace for the range where 10s ≤ SD ≤ 20s, achieving 99% agreement, but this pattern is not reproduced in the RON2 simulations, and therefore appears to be merely an artifact of the trace. Despite Sigma_UD's variability for SDs < 60 seconds, it may be of note that the percentage of views in agreement never goes below 60%. For SD ≥ 60s, Sigma_UD consistently remains in the range of 90% agreement or better during both traces, even achieving 100% in many data points. Specifically, Sigma_UD achieves 100% agreement for 65s ≤ SD ≤ 80s and SD ≥ 90s in RON2, and for SD ≥ 95s in RON1.

Figure 6-3 demonstrates a contrast between Sigma_UD and the other two algorithms, Sigma_LD and Moshe. However, judging by the raw number of agreed views, rather than percentages, as shown in Figure 6-4 for RON1, we observe that the difference is really quite subtle. We define *raw number of agreed views* as the actual number of views that achieved agreement. Figure 6-4(a) shows the complete raw agreement data for RON1 as an exponentially decreasing function of SD.

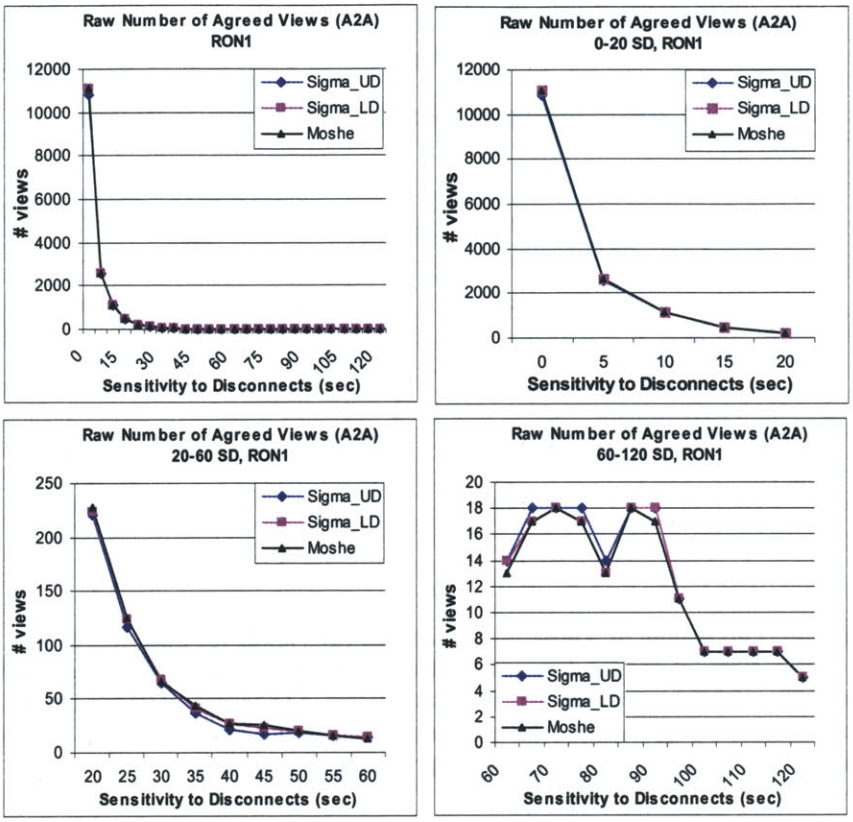

Figure 6-4. Raw Numbers of Agreed Views, All-to-All Sigma vs. Moshe, RON1. (a) Overall picture and (b,c,d) Piecewise analysis.

In Figure 6-4(b,c,d) we investigate the raw agreement patterns in a piecewise manner. Even at this scale, the trends are very similar, although slight differences are visible for Sigma_UD. Percentages as low as 60% occur for Sigma_UD in Figure 6-3 despite the subtle changes in raw agreement numbers, in part because the total number of views itself is relatively small for data points in the lower asymptote. These results are closely reproduced for RON2, as seen in Appendix C, Figure C-2.

6.1.3 Disagreement

Figure 6-5 shows the percentage of views in disagreement for the A2A algorithms, for (a) RON1 and (b) RON2. Both Sigma_LD and Moshe appear to operate with no disagreements at all. Any differences between the disagreement patterns between Sigma_LD and Moshe are negligible and therefore not significant. We mention this because there are, in fact, slight differences between Sigma_LD's disagreement performance and Moshe's, as shown in Appendix E for RON1. Specifically, Sigma_LD

produces a very small, albeit nonzero, amount of disagreement when SD is short enough. In RON1, disagreement is nonzero for SD ≤ 10s. In RON2, disagreement is nonzero for SD ≤ 15s. Nevertheless, Sigma_LD's nonzero disagreement percentages never exceed one half of one percent: 0.35% in RON1, and 0.42% in RON2. For all other SDs, Sigma_LD produces zero disagreement. Moshe produces zero disagreement for all SDs, in both traces, as expected.

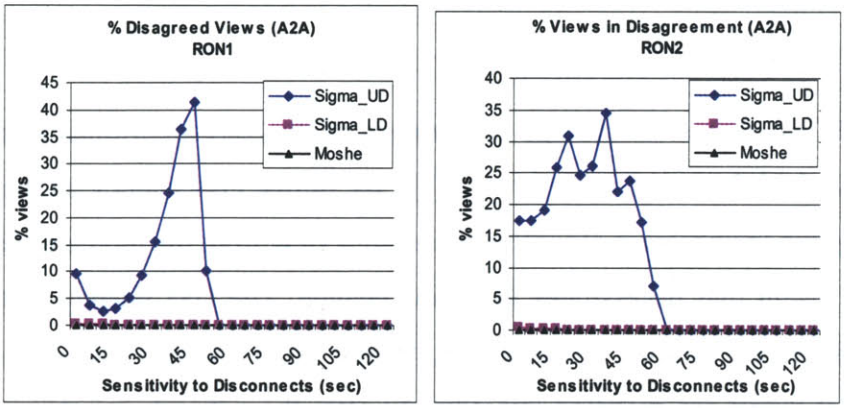

Figure 6-5. Percentage of views in disagreement. (a) RON1 and (b) RON2.

The patterns for Sigma_UD appear to complement the agreement percentages seen in Figure 6-3. For example, where Figure 6-3(a) shows a local maximum at SD = 15s, Figure 6-5(a) shows a matching local minimum. Also, the lowest agreement percentages in Figure 6-3 correspond to the highest disagreement percentages in Figure 6-5.

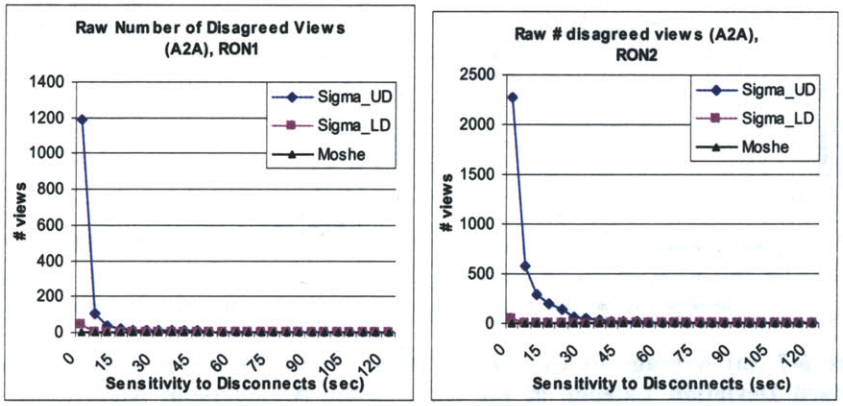

Figure 6-6. Raw Number of Disagreed Views. (a) RON1 and (b) RON2.

Sigma_UD produces a number of disagreed views, peaking at 45 seconds with 41% of views being disagreed. Again, this number is skewed by the fact that the total number of views decreases exponentially with increasing SD. For SD ≥ 55s, however, Sigma_UD produces no disagreement. Figure 6-6 confirms this fact by demonstrating an exponential decrease to zero in Sigma_UD's raw number of disagreed views, for the same points where Figure 6-5 shows disagreement percentages as high as 41%.

6.1.4 GM Latency

Figure 6-7(a) shows the average GM latency of the A2A algorithms as a function of SD, with standard deviation in Figure 6-7(b). We define GM latency as the time that it takes for a group membership algorithm to install a view from the moment that it receives the network event that necessitates this view change. The GM latency patterns for the three algorithms are similar. As expected, Sigma_UD's average latency is negligible, because most views are installed immediately after the network event is received. On average, Sigma_UD is faster than Sigma_LD by a range between 154ms at the longest SDs, to 189ms at the shortest SDs. Sigma_LD is faster than Moshe by an average of 30ms. These average GM latency results are closely reproduced in RON2, shown in Appendix C, Figure C-3.

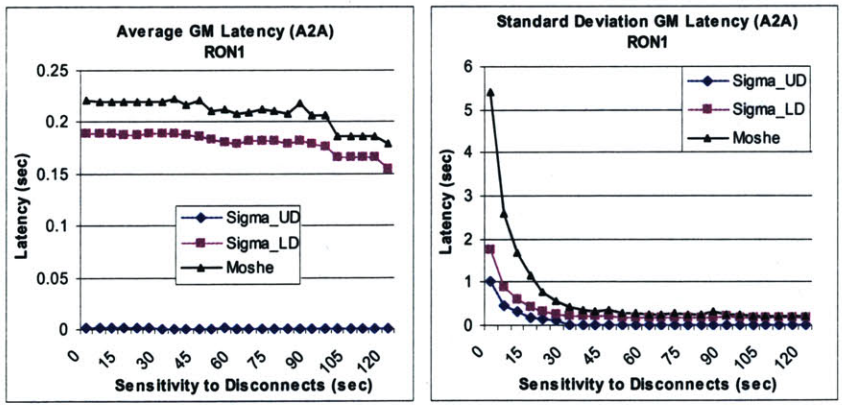

Figure 6-7. (a) Average Latency of the A2A Group Membership Algorithms, RON1. (b) Standard Deviation Latency of the A2A Group Membership Algorithms. On average, Sigma_LD is slightly less than 200ms slower than Sigma_UD, while Moshe is roughly 30ms slower than Sigma_LD. The 200ms difference between Sigma_UD and Sigma_LD is related to the bottleneck link latency in the simulation.

Average GM latency offers a general impression of the relative performance tendencies of each group membership algorithm. However, more important in our practical analysis of the algorithms' performance is the formulation and comparison of upper bound latencies, because upper bounds provide guarantees about worst-case performance. For this reason, we present the maximum GM latency for each algorithm as a function of SD, shown in Figure 6-8. The trends are similar for both RON1 and RON2. For the shortest SDs, Sigma_UD's maximum latency is on the order of Sigma_LD's maximum latency (for example, in RON1, Sigma_UD's max latency is 244 ms at SD = 0, and Sigma_LD's is 253 ms), but becomes negligible as SD is increased beyond 45 seconds. Sigma_LD's maximum latency remains fairly constant between 200 and 250ms. Moshe's range of maximum latencies is much wider, and appreciably higher, ranging between 250ms and 530ms, with higher latencies corresponding to shorter SDs.

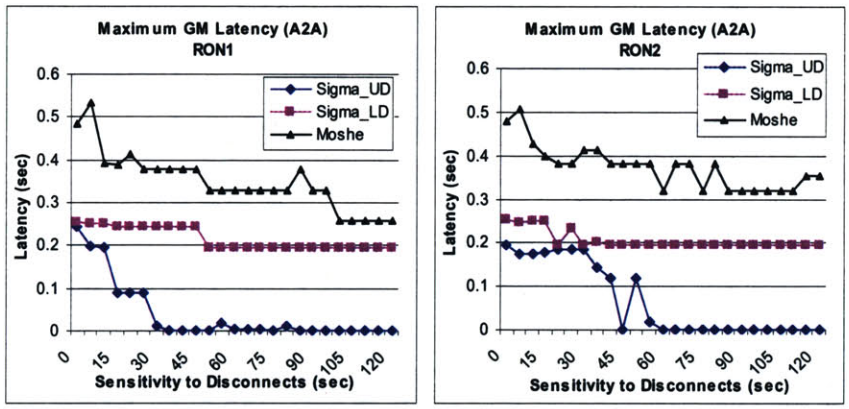

Figure 6-8. Maximum Latency, All-to-All Sigma vs. Moshe. Maximum latency of Sigma_UD, Sigma_LD, and Moshe execution during (a) RON1 and (b) RON2. Both traces show similar patterns. Sigma_UD's maximum latency is similar to Sigma_LD's for the shortest SDs, but becomes negligible beyond 45 second SDs. On average, in (a) Sigma_LD's max latency is 176ms longer than Sigma_UD, and Moshe's is 135ms longer than Sigma_LD. In (b), Sigma_LD's max latency is 138ms longer than Sigma_UD, and Moshe's is 169ms longer than Sigma_LD.

6.1.5 Message Overhead

Figure 6-9 shows the average message overhead for the A2A algorithms, as a function of SD, for the RON1 trace. From Figure 6-9(a), we observe that the average message overhead is an exponentially decreasing function of SD. This is consistent with the result in Section 6.1.1 in which we observe that the total number of views delivered is also an exponentially decreasing function of SD. Each view delivery is a culmination of

an exchange of messages; thus, the message overhead is related to the number of view deliveries.

For SD ≤ 20s, Moshe has a much higher message overhead than Sigma_UD and Sigma_LD. For example, at SD = 0, each Moshe member sends an average of 300,000 messages during the RON1 trace. On the other hand, each Sigma_LD member sends an average of 153,918 messages, half the messages that Moshe sends. Sigma_UD members send 115,657 messages at SD = 0s.

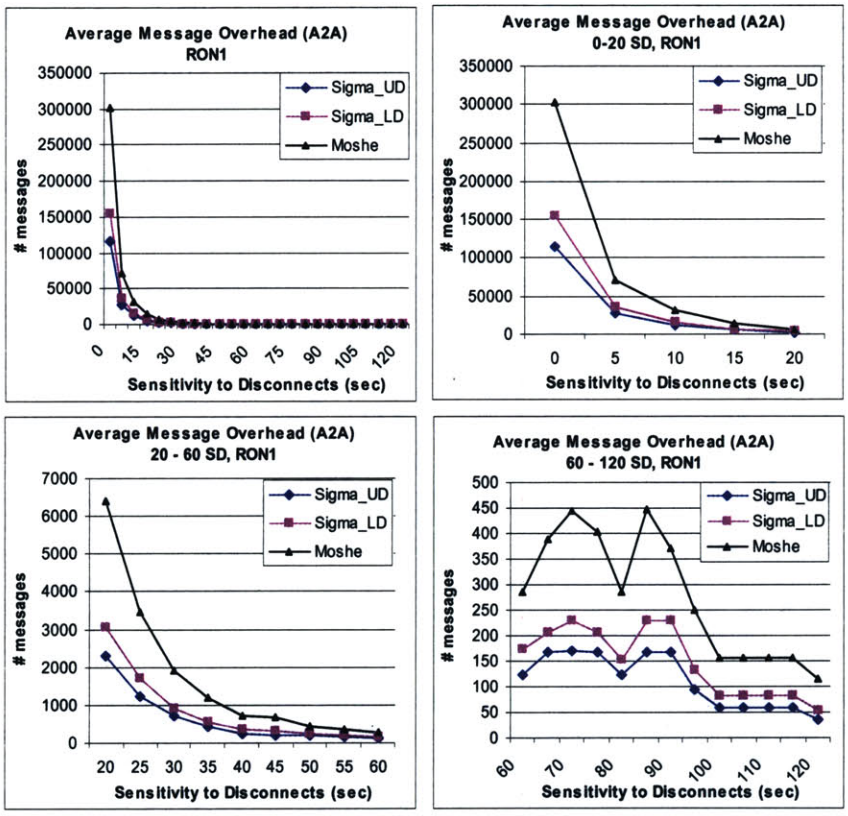

Figure 6-9. Average Message Overhead, RON1, All-to-All Sigma vs. Moshe. (a) Complete picture of average message overheads as an exponentially decreasing function of SD, for Sigma_UD, Sigma_LD, and Moshe. For the shortest SDs, this graph shows Moshe to have a higher message overhead than Sigma_UD and Sigma_LD. (b,c,d) Clarifying the trends in the lower asymptote shows Moshe to consistently maintain a significantly higher message overhead than both Sigma variants. Sigma_LD has a slightly higher message overhead than Sigma_UD.

The exponential nature of average message overhead trends in Figure 6-9(a) hinders a clear analysis of the trends, especially in the lower asymptote, starting from SD = 20, because at this scale they overlap. We investigate message overhead trends in a piecewise manner in Figure 6-9(b,c,d); by isolating data points that share the same order of

magnitude we can obtain a clearer picture of the actual trends. Figure 6-9(b) confirms our observations regarding Moshe's higher message overhead for SD ≤ 20s, as well as the exponentially decreasing relationship between message overhead and SD for all three algorithms.

Figure 6-9(c) shows message overhead trends for 20 ≤ SD ≤ 60 seconds. The trends in this range continue to decrease exponentially as a function of SD. Here, and also in Figure 6-9(d), which completes the piecewise analysis, Moshe consistently has two times (more precisely, by a factor of 1.95 on average) the message overhead of Sigma_LD. Throughout Figure 6-9, Sigma_LD consistently shows a slightly larger message overhead than Sigma_UD, by a factor of 1.4 on average. The patterns observed in Figure 6-9 during the RON1 trace are reproduced during RON2. The latter results are provided for completeness in Appendix C, Figure C-4.

6.2 LB Study: Leader-Based Sigma vs. Ensemble

In this section, we present the results of the Leader-Based (LB) simulations, in which we compare Leader-Based Sigma_UD, Leader-Based Sigma_LD, and Ensemble. Overall, we observe the results in the LB study to be similar to the results of the A2A study:

- The total number of views delivered drops exponentially, as view duration increases, with increasing SD.
- Leader-Based Sigma_LD's agreement levels are close to the 99-100% agreement of Ensemble.
- While Leader-Based Sigma_UD produces more non-agreed views, thereby reducing agreement percentages, raw agreement trends are consistent among the three algorithms.
- Leader-Based Sigma_LD is faster than Ensemble, and Leader-Based Sigma_UD is faster than Leader-Based Sigma_LD.
- Leader-Based Sigma_UD and Leader-Based Sigma_LD share the same message overhead, which is smaller than the message overhead of Ensemble.
- We also observe several contrasts between All-to-All and Leader-Based Sigma, most notably that Leader-Based Sigma reduces message overhead by tenfold.

6.2.1 Number and Duration of Views

In Figure 6-10, we observe the total number of views delivered by Leader-Based Sigma_UD, Leader-Based Sigma_LD, and Ensemble to be virtually the same in RON1, with the notable exception of the data point at SD = 0s. Note also that the total number of views follows an exponentially decreasing pattern as SD increases. The piecewise close-up shown in Figure 6-10(b,c,d) confirms the similarity of the trends for the three algorithms, clarifying the patterns in the lower asymptote. These results are reproduced in RON2, shown in Appendix D, Figure D-1.

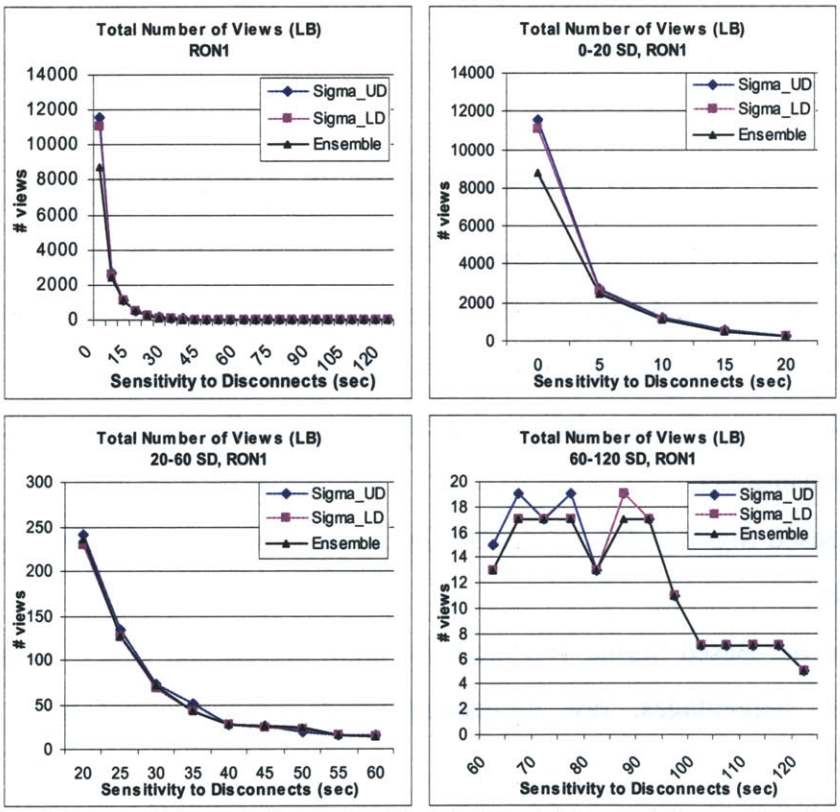

Figure 6-10. Total Number of Views, RON1, Leader-Based Sigma vs. Ensemble. (a) Overall picture (b,c,d) Piecewise close-up of trends.

Figure 6-11 shows the duration of views for Leader-Based Sigma. As described in Section 6.1.1, duration of views is measured as seconds/view, and describes how long each view lasts. Similarly to the All-to-All results, we observe that duration of views increases drastically with increasing SD, both for RON1 and RON2. When SD = 0, view

duration is 15 seconds; but when SD is increased to just 60 seconds, view duration becomes nearly 4 hours. When SD = 120, each view lasts for nearly 10 hours.

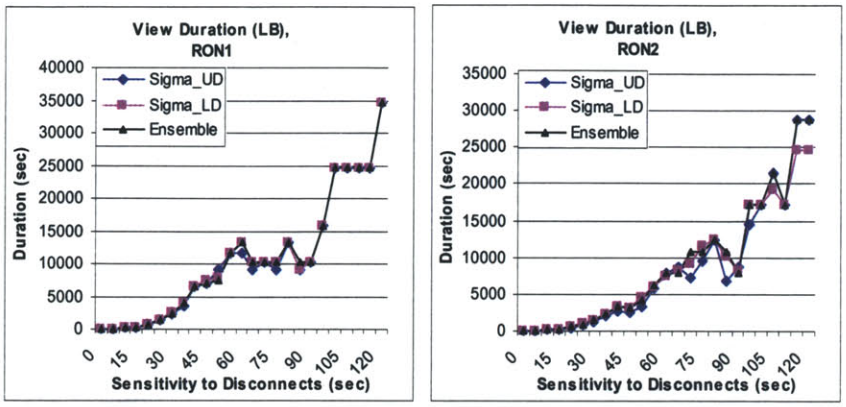

Figure 6-11. Duration of Views, Leader-Based Sigma vs. Ensemble. (a) RON1, (b) RON2.

6.2.2 Agreement

Figure 6-12 shows the Leader-Based agreement percentages. Ensemble maintains 99-100% agreement. Leader-Based Sigma_LD matches Ensemble's agreement performance with 95-100% agreement, except for the data points in RON2 (Figure 6-12(b)) at 80s, 95s, and 100s SD where its agreement drops to 85-90%. However, interleaved between these low points are the data points at 75s, 85s, and 90s SD for which LB Sigma_LD's agreement is at 100%. Judging by these interleaved strong points, as well as the consistent agreement performance in RON1 (Figure 6-12(a)), the three data points with 85-90% agreement in RON2 appear to be outliers, artifacts of the trace.

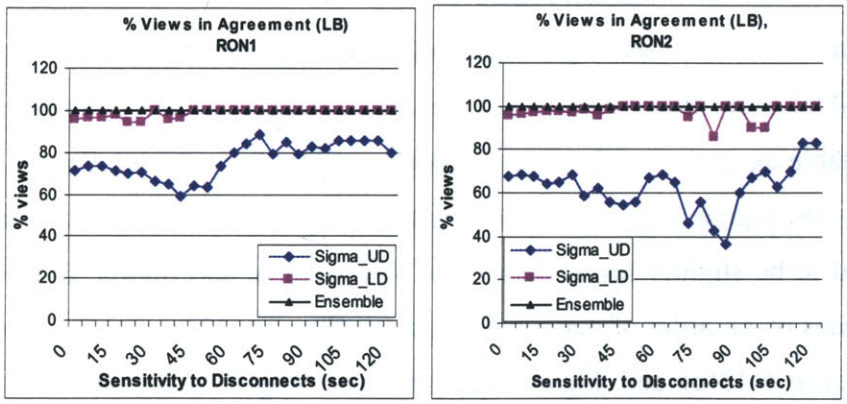

Figure 6-12. Percentage of Views in Agreement, Leader-Based Sigma vs. Ensemble. (a) RON1 and (b) RON2.

Leader-Based Sigma_UD, similarly to the All-to-All results for Sigma_UD, operates with a lower agreement percentage, ranging from a low of 36% at SD = 85s in RON2 to a high of 88% at SD = 70s in RON1.

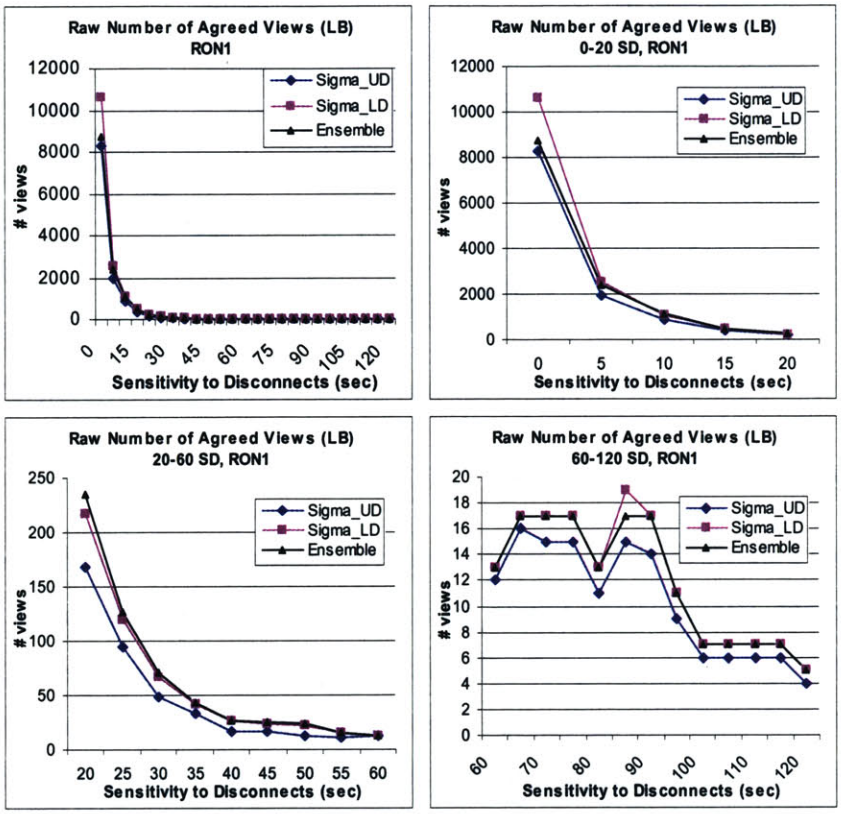

Figure 6-13. Raw Number of Agreed Views in RON1, Leader-Based Sigma vs. Ensemble. (a) Overall picture (b,c,d) Clarifying the trends by piecewise analysis of data points that share the same order of magnitude.

Similarly to the A2A results, Figure 6-13(a) shows the raw number of agreed views among the three leader-based algorithms to follow exponentially decreasing trends with increasing SD. Leader-Based Sigma_LD matches Ensemble's agreement numbers closely with the notable exception of the first datapoint at SD = 0s in RON1 (Figure 6-13(b)). In fact, at SD = 0s, Sigma_LD more closely matches Sigma_UD. The trends for Sigma_UD are observed to be slightly lower than those of Ensemble and Sigma_LD. Note that the scale decreases by an order of magnitude between Figure 6-13(b) and Figure 6-13(c), and between Figure 6-13(c) and Figure 6-13(d); thus "slight" differences as observed in Figure 6-13(b) are actually not so slight, on the order of thousands, when compared to slight differences in Figure 6-13(c), of less than one hundred, or Figure 6-13(d), where

differences are fewer than 10 views. The raw agreement trends are similar for RON2, shown in Appendix D, Figure D-2.

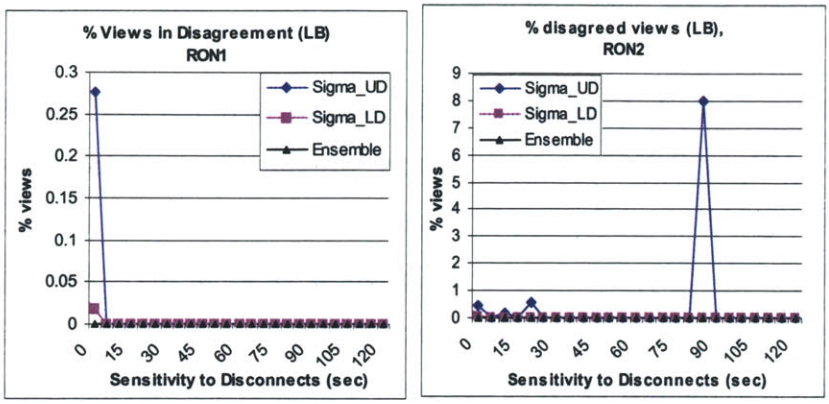

Figure 6-14. Percentage of Views in Disagreement, Leader-Based Sigma vs. Ensemble. (a) RON1 and (b) RON2.

6.2.3 Disagreement

In Figure 6-14, we observe the percentage of views that are disagreed among the three Leader-Based algorithms. The RON1 results in Figure 6-14(a) show zero disagreement for all but the first data point at SD = 0s, where Leader-Based Sigma_UD produces 0.28% disagreement (32 out of 11566 views), and Leader-Based Sigma_LD produces only 0.02% disagreement – only 2 views out of 11091 total are disagreed.

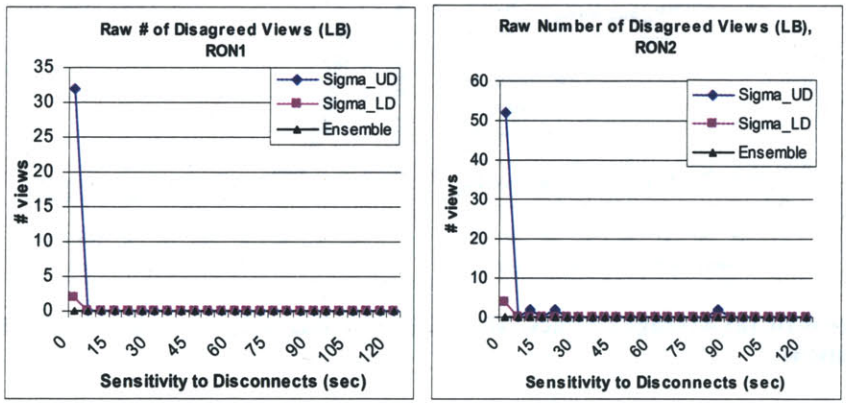

Figure 6-15. Number of Views in Disagreement, Leader-Based Sigma vs. Ensemble. (a) RON1 and (b) RON2.

The RON2 results in Figure 6-14(b) show zero disagreement for all three algorithms in all but four data points. Ensemble maintains zero disagreement for all data points.

Leader-Based Sigma_LD has nonzero disagreement only at SD = 0s with 0.04% disagreement. Leader-Based Sigma_UD has nonzero disagreement at the following four data points: 0.45% at SD = 0s, 0.15% at SD = 10s, 0.54% at SD = 20s, and 8% at SD = 80s.

The disproportionately large disagreement percentage of 8% at SD = 80s for Leader-Based Sigma_UD in RON2 deserves further insight. Figure 6-15 shows the actual numbers of disagreed views for RON1 (Figure 6-15(a)) and RON2 (Figure 6-15(b)). If we look at Figure 6-15(b), the number of disagreed views at SD = 80s is actually very small – only 2 views are disagreed. The high percentage in Figure 6-14(b) is skewed by the fact that the total number of views decreases exponentially with increasing SD.

6.2.4 GM Latency

Figure 6-16(a) shows the average GM latencies for the Leader-Based algorithms in RON1. The average GM latency appears to decrease slightly with increasing SD, most notably for Ensemble and Leader-Based Sigma_LD. Leader-Based Sigma_UD's average latency remains constant at an average of 31ms.

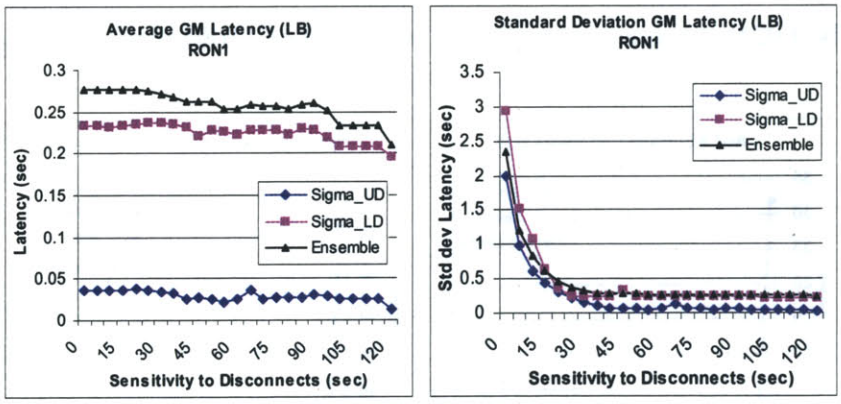

Figure 6-16. (a) Average Latency, RON1, Leader-Based Sigma vs. Ensemble. (b) Standard Deviations.

Leader-Based Sigma_LD's average latency remains within the range of 195ms at SD = 120s to 237ms at SD = 30s and SD = 45s. Ensemble's average latency runs from a low of 234ms at SD = 115s and SD = 120s to a high of 280ms at SD = 25s. Overall, Sigma_LD's average latency is 225ms, and Ensemble's average latency is 260ms. Thus,

on average, Leader-Based Sigma_UD is faster than Leader-Based Sigma_LD by 194ms, and Leader-Based Sigma_LD is faster than Ensemble by 35ms. For reference, the standard deviations of the GM latencies are shown in Figure 6-16(b). These results are reproduced for RON2 and are shown in Appendix D, Figure D-3.

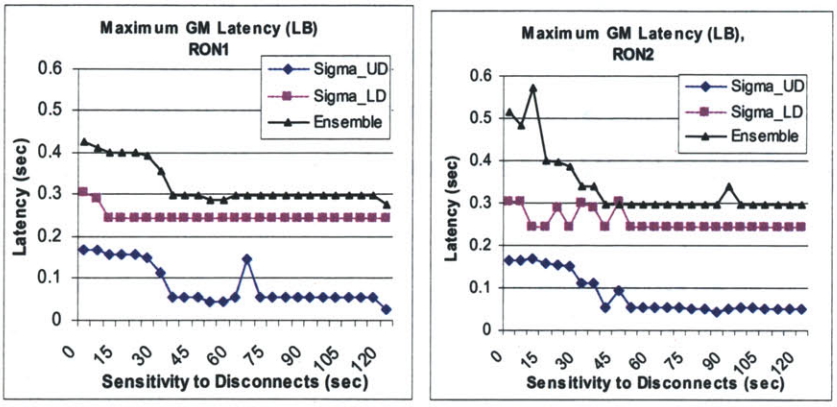

Figure 6-17. Maximum Latencies, Leader-Based. (a) RON1 and (b) RON2.

Similarly as discussed in the All-to-All case, average GM latencies only demonstrate relative overall differences in performance; more meaningful in our performance analysis are upper bounds – the differences in the worst-case behavior of the three leader-based algorithms. To this end, Figure 6-17 presents the maximum GM latencies for the leader-based algorithms. Again, Leader-Based Sigma_UD is the fastest, with maximum latencies ranging from 27ms at SD = 120s in RON1 to 167ms at SD = 5s in RON1 and SD = 10s in RON2. Leader-Based Sigma_LD maintains a maximum latency between 244ms and 300ms, with most data points coinciding with the former. Ensemble's maximum latency ranges from a low of 275ms at SD = 120s in RON1 to a high of 571ms at SD = 10 in RON2.

6.2.5 Message Overhead

Having observed much similarity between the All-to-All trends and the Leader-Based trends thus far, it is no surprise that the average message overhead trends for Leader-Based Sigma and Ensemble are also consistent with those seen for All-to-All in Chapter 6.1.5. Figure 6-18(a) shows the message overhead trends to decrease exponentially with increasing SD. Ensemble's message overhead is greater than that of

Leader-Based Sigma by 10000 messages at SD = 0. As evidenced by the piecewise analysis in Figure 6-18(b,c,d), Leader-Based Sigma has a consistently smaller average message overhead than Ensemble, although by a lesser margin than that seen in the A2A study. Leader-Based Sigma_UD and Sigma_LD share essentially the same average message overhead, their trends overlapping consistently in Figure 6-18(a,b,c). Only in Figure 6-18d, the differences between Sigma_UD and Sigma_LD emerge to be on the order of one or two messages in the common case, and at most 10 messages. These results are reproduced in RON2, and are shown in Appendix D, Figure D-4.

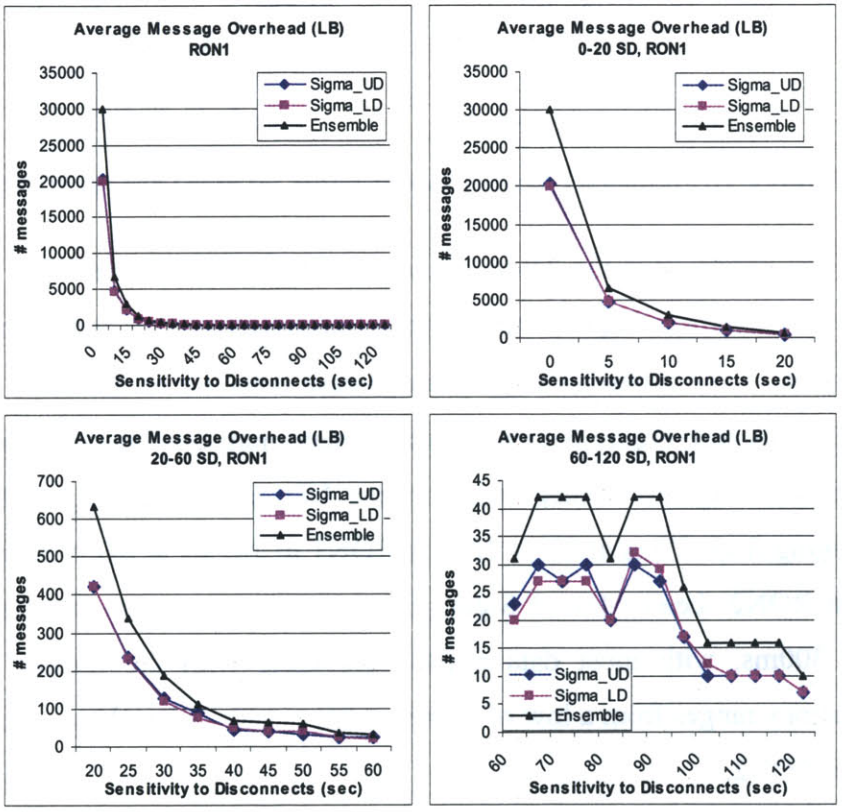

Figure 6-18. Average Message Overhead during RON1, Leader-Based. (a) Overall picture (b,c,d) Piecewise analysis.

Chapter 7

Discussion and Conclusions

Our performance analysis evaluates Sigma's practical potential and in doing so, explores the extent to which GM algorithms can be optimized. While a GM protocol is handling changes in groups' membership, higher-level applications are blocked, waiting to receive a new view. After receiving the new view, they have to synchronize with the other members to make sure everyone has received it, and to agree on a consistent state. The performance of GM applications therefore directly depends on (1) how long the underlying GM protocol takes to form new views and (2) how frequently these new views are created.

In Chapter 5, we quantified Sigma's performance in these terms, both in its original all-to-all form and as a leader-based version, in the context of its predecessors, all-to-all Moshe and leader-based Ensemble. We have measured the GM latency, message overhead, and duration of views of these algorithms. Our investigation confirms that both A2A Sigma and LB Sigma are faster and have a smaller message overhead than their respective GM predecessors.

One of our goals in this analysis has also been to evaluate the effectiveness of a filter for limiting disagreement. For this purpose, we have presented accuracy measurements in terms of both agreements and disagreements. The fact that Sigma_LD and Leader-Based Sigma_LD match the accuracy of Moshe and Ensemble suggests that such a filter is indeed very effective. Furthermore, Sigma_UD's less than optimal accuracy in the absence of a filter confirms that a filter is not only effective, but also necessary for Sigma to achieve practical accuracy.

We have also examined Sensitivity to Disconnects (SD) as a parameter of value to GM algorithms, and experimented with a range of SDs. Our results reveal that SD as short as 60s effectively increases view duration by several hours. In addition, we have observed that SD itself acts as a type of filter, by ignoring transient events – the most common source of disagreement. Thus, equipped with a Sensitivity to Disconnects of

sufficient length, Sigma_UD is able to achieve much higher accuracy, approaching that of Sigma_LD, Moshe, and Ensemble.

7.1 View Formation Time and Overhead

The results show that All-to-All Sigma and Leader-Based Sigma are significantly more efficient than Moshe and Ensemble, respectively, both in terms of latency and message overhead. All-to-All Sigma_LD is faster on average than Moshe by 30ms, and Leader-Based Sigma is faster on average than Ensemble by 33ms. In terms of worst-case latency, All-to-All Sigma_LD is faster than Moshe by as much as 260ms, and Leader-Based Sigma_LD is faster than Ensemble by as much as 327ms.

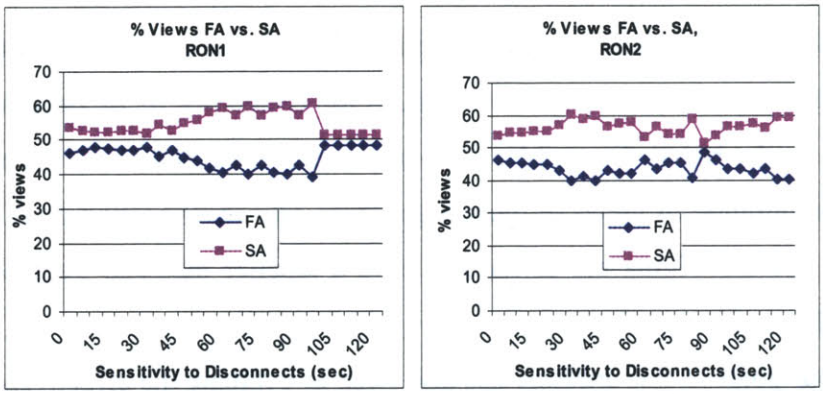

Figure 7-1. Percentage of Moshe's Views delivered in FA vs. SA. (a) RON1. (b) RON2.

Figure 7-1 shows the percentage of Moshe's views that were delivered with fast agreement (FA) versus slow agreement (SA). From these results we see that Moshe runs SA between 50 and 60% of the time, which means that Moshe delivers the majority of its views after two or more rounds. In contrast, All-to-All Sigma is guaranteed to deliver views within one round. Similarly, Ensemble always takes three communication rounds to deliver views, while Leader-Based Sigma is guaranteed to deliver views within two rounds.

Moshe's heavy use of SA (60%) in our simulations deserves further comment, because this result is very different from the performance of Moshe observed in [34], where less than 2% of all views were delivered using SA. This difference can be explained as follows. SA occurs more often in a network topology where more nodes are

directly connected to each other [34]. The experiments in [34] model only five nodes, comprising at most 20 direct connections. In contrast, our experiments model 16 nodes, for a total of 240 direct connections. The greater than tenfold increase of direct connections in our simulations can certainly account for SA difference. Indeed, Keidar et al. imply that their results can only generalize to large numbers of nodes under the assumption that the network topology is configured with no more than five direct connections [34].

Sigma_UD is faster on average than Sigma_LD by the average latency of the network topology. This is consistent with the fact that Sigma_UD delivers most views immediately, while Sigma_LD always delivers views at the end of one round. The case is similar for Leader-Based Sigma, except for an initial delay shared by both LD and UD due to the extra leader-based step. Our measurements show this delay to be 50ms on average for the link topology that we used. Accordingly, all timings in the LB study are prolonged by 50ms relative to the A2A measurements of average GM latency. In the worst case, Sigma_UD is faster in than Moshe by nearly 400ms, and by 230ms on average. With nearly identical results, Leader-Based Sigma_UD is faster than Ensemble by up to 404ms, and by 230ms on average.

A2A Sigma also has a significantly smaller message overhead than Moshe, by a factor of 1.95. Again, this has to do with the majority of Moshe's views being delivered in SA – more rounds mean more messages, which explains Moshe's larger message overhead. Similarly, for Sigma, fewer rounds mean fewer messages, a smaller message overhead. These results are reproduced when comparing Leader-Based Sigma with Ensemble, because Ensemble requires three rounds of message exchange, in contrast to LB Sigma's two rounds.

Appendix F presents data that distinguishes between Moshe's FA and SA message overhead. From this data, it is clear that Moshe's FA message overhead is identical to the message overhead of Sigma_LD, while SA adds its own substantial message overhead on top of that. This observation sheds light on how Moshe ends up with nearly twice the message overhead of Sigma.

A2A Sigma not only has a smaller message overhead than Moshe, but also a smaller message size. Besides the view id and membership set, Moshe's messages also include an

extra array that grows with the number of members: startChangeNums in view messages, and usedProps in proposal messages [34]. Sigma's messages, on the other hand, only carry the view id and membership set. Ensemble's message size is similar to that of Sigma.

7.2 Frequency of New Views

Our experimentation with Sensitivity to Disconnects has revealed this parameter to be useful, as both a means of configuring GM based on application preferences, as well as a technique for filtering out transient events that lead to disagreements. We discuss the former benefit in this section, and the latter in Section 7.3. As a configuration parameter, SD enables applications to fine-tune view duration according to their specific needs. In our measurements, we have found that frequency of views diminishes, and therefore view duration increases, greatly by increasing SD. A smaller frequency of view delivery (and accordingly greater view duration) means that the application spends less time resynchronizing and operates with less interruption.

Despite the efficiency benefits of a Sensitivity to Disconnects such as 60s $\leq$ SD $\leq$ 120s, leaving it as an optional, configurable parameter is important. Ultimately, applications must face the tradeoff between minimizing resynchronization overhead and maximizing accurate current representation of the group membership. One issue to consider when determining the right balance in this tradeoff is the duration of transient disconnects that the application is able, and would prefer, to tolerate without reacting to them. Different applications have different definitions of what "transient" means; SD enables applications to adjust the granularity of events to be perceived as transient. This tradeoff is especially pivotal for mission critical applications, in which both speed and moment-by-moment accuracy are of absolute importance.

For example, in our simulations, a mere five-second SD increases view duration to a minute, a reasonable duration for certain mission critical applications. Just 15 seconds of SD increases view duration to five minutes, which in relative network terms is a very long time; a 25 second SD increases view duration to 23 minutes – for most network applications, an eternity.

In any case, our results show that running GM without any SD at all is suboptimal for both sides of the tradeoff. In our simulations, this resulted in view changes every 15 seconds. Such frequent view changes cause constant interruption of the application's normal operation. Even though it might literally be the most accurate representation of the current membership, a short-lived view is meaningless to most applications, because a new one is delivered before the application has a chance to use it during normal operation. Besides, by the time the view change is done, the view could be outdated, because new views are delivered so frequently. The application would be at a virtual standstill, paralyzed by a cycle of interruption, reconfiguration, and obsoleteness. Thus, GM would be useful only to those applications that do not need GM to respond immediately to every network event – applications that can tolerate a Sensitivity to Disconnects.

7.3 Accuracy

So far, we have discussed Sigma's significant efficiency advantages and the benefits of using SD to optimize efficiency by increasing view duration and therefore decrease resynchronization overhead. However, even the greatest breakthroughs in efficiency would be meaningless, unless sufficient accuracy is achieved. In this section, we discuss Sigma's accuracy, in terms of agreement and disagreement.

The results show that Sigma can achieve good quality agreement and zero disagreement, when equipped with a filter for limiting disagreement. The limiting disagreement filter proposed in [39] is very effective for Sigma in our simulations. All-to-All Sigma_LD achieves accuracy on par with Moshe, consistently reaching 99-100% agreement and consistently producing zero disagreement. Even at the shortest SDs, Sigma_LD's disagreement percentages are negligible, less than half of one percent of all views. Similarly, Leader-Based Sigma_LD consistently matches Ensemble's agreement accuracy and zero disagreement.

On the other hand, Sigma_UD produces quite a bit of disagreement and visibly lower percentages of agreement, especially for SD $\leq$ 60s. This observation, in contrast to Sigma_LD's good quality agreement for the same SD range, confirms that the LD filter for limiting disagreement is indeed effective.

Note that Leader-Based Sigma_UD shows lower agreement percentages than All-to-All Sigma_UD for SD > 60 seconds, although it does show an increasing trend. This discrepancy may be due to the extra round in the leader-based version, which may require a slightly longer SD than 120 seconds for accuracy to improve sufficiently to match that of Leader-Based Sigma_LD and Ensemble.

There are other filters that can be used to limit disagreement. Sensitivity to Disconnects is one example, and it can complement the use of other filters. SD acts as a disagreement filter by ignoring transient events. Transient events are usually due to asymmetric network instabilities such as congestion, link failures, and message loss, which lead to disagreements. By filtering out transient events, SD eliminates this major source of disagreements. Unlike the LD filter, SD buffers GM and its applications from the adverse effects of transient network events without affecting the GM latency.

Another avenue of optimization for the process of limiting disagreement is to modify the LD filter to distinguish between voluntary and involuntary network events. Because the reliable FIFO communication service guarantees voluntary network events to be symmetric, they will never cause disagreements. Thus, we can optimize Sigma_LD's efficiency by running Sigma_UD for voluntary network events. Future directions for study include designing and evaluating such alternative limiting disagreement filters.

7.4 Scalability

In Chapters 3 and 4, we discussed the scalability benefits that Leader-Based Sigma can offer in theory. Comparing the message overhead graphs for All-to-All Sigma and Leader-Based Sigma, our simulation results confirm the theoretical figures with Leader-Based Sigma demonstrating at least a tenfold reduction in message overhead.

Despite the reduced message overhead, Leader-Based Sigma was still slightly slower in our simulations than All-to-All Sigma, by roughly 50ms on average. However, we believe that the benefits of a reduced message overhead are not fully reflected in our simulations, for two reasons. First, we do not simulate lossy links, in which spontaneous message losses occur. In lossy links, even relatively low loss rates are amplified by the greater volume of messages sent in an All-to-All algorithm. Secondly, our simulations are not run in the context of a larger encapsulating public network. Although the RON trace

realistically models a WAN environment, taking into account network congestion, network outages, and other causes of network instability, the message overhead due to our simulated GM does not contribute to any network congestion outside the 16 participating nodes. To fully explore the scalability benefits of Leader-Based Sigma over All-to-All Sigma, the next step would be to implement and test them on a real research network such as PlanetLab. Future research will continue to explore tradeoffs between Sigma and Leader-Based Sigma by implementing both in real network environments.

7.5 Conclusions

Our performance analysis of Sigma, the first single-round group membership algorithm, has brought forth some encouraging observations. We have confirmed that Sigma's underlying design – decoupling the goal of achieving agreement from that of limiting disagreement – effectively reduces latency, message overhead, and message size, while preserving good quality agreement on par with such practical algorithms as Moshe and Ensemble. Combining Sigma with a filter for limiting disagreement is effective, resulting in virtually all views being in agreement while preserving the single-round worst-case upper bound. Ignoring short-lived instabilities using a configurable Sensitivity to Disconnects (SD) is also useful to buffer GM and its applications from the adverse effects of transient network events without affecting the GM latency.

These results strongly indicate that Sigma is not just a theoretical result, but is indeed a result with important practical implications for Group Communication systems: the efficiency of GM applications can be significantly improved, without compromising accuracy, by using Sigma in place of Moshe, and Leader-Based Sigma in place of Ensemble. Although the optimal configuration details depend on the nature of the application and the network environment in which it runs, we hope that our results and discussions regarding SD, Leader-Based Sigma, and filters for limiting disagreement will provide insight towards Sigma's optimal deployment in a broad range of group-oriented applications, from data replication and collaboration systems in wired networks, to mission-critical operations such as group security in dynamic mobile military environments.

Bibliography

[1] Agarwal, D. A., Moser, L. E., Melliar-Smith, P. M., and Budhia, R. K. "The Totem Multiple-Ring Ordering and Topology Maintenance Protocol." ACM Transactions on Computer Systems, (1998), 16(2), pp. 93-132.

[2] Amir, Y., and Stanton, J. "The Spread Wide Area Group Communication System." Center for Networking and Distributed Systems, Johns Hopkins University. Technical Report CNDS 98-4, July, 1998.

[3] Amir, Y., Dolev, D., Kramer, S., and Malki, D. "Transis: A communication sub-system for high availability." *22nd IEEE Fault-Tolerant Computing Symposium (FTCS).* July, 1992.

[4] Amir, Y., et al. "The Secure Spread Project." http://www.cnds.jhu.edu/research/group/secure_spread/.

[5] Amir, Y., et al. "Secure Group Communication in Asynchronous Networks with Failures: Integration and Experiments." *20th IEEE International Conference on Distributed Computing Systems (ICDCS).* April, 2000.

[6] Amir, Y., Kim, Y., Nita-Rotaru, C. and Tsudik, G. "On the Performance of Group Key Agreement Protocols." *Proceedings of the 22nd IEEE International Conference on Distributed Computing Systems. Vienna, Austria.* June, 2002.

[7] Amir, Y., Moser, L. E., Melliar-Smith, P. M., Agarwal, D. A., and Ciarfella, P. "The Totem single-ring ordering and membership protocol." ACM Transactions on Computer Systems, (1995), 13(4).

[8] Amir, Y., Nita-Rotaru, C., Stanton, J., and Tsudik, G. "Scaling Secure Group Communication Systems: Beyond Peer-to-Peer." *In the Proceedings of DISCEX3. Washington DC.* April 22-24, 2003, 2002.

[9] Anderson, D. G., "http://nms.lcs.mit.edu/projects/ron/."

[10] Anderson, D. G., "http://nms.lcs.mit.edu/projects/ron/data/ron1-latency.desc."

[11] Anderson, D. G., Balakrishnan, H., Kaashoek, M. F., and Morris, R. "Resilient Overlay Networks." *Proceedings of the 18th ACM SOSP. Banff, Canada.* October, 2001.

[12] Anker, T., Breitgand, D., Dolev, D., and Levy, Z. "CONGRESS: Connection-oriented Group-address Resolution Service." *Proceedings of SPIE on Broadband Networking Technologies.* November 2-3, 1997.

[13] Anker, T., Shnayderman, I., and Dolev, D. "The Design of Xpand: A Group Communication System for Wide Area Networks." The Hebrew University of Jerusalem, Computer Science Department. Technical Report HUJI-CSE-LTR-2000-31, July, 2000.

[14] Bakr, O., and Keidar, I. "Evaluating the Running Time of a Communication Round over the Internet." *The 21st ACM Symposium on Principles of Distributed Computing (PODC). Monterey, CA.* July, 2002.

[15] Balenson, D. M., et al. "Dynamic Cryptographic Context Management (DCCM) Final Report." Cryptographic Technologies Group, Trusted Information Systems, NAI Labs, The Security Research Division of Network Associates, Inc. Report 4, April 6, 2000.

[16] Ban, B., et al. "JGroups -- A Toolkit for Reliable Multicast Communication." http://www.jgroups.org/javagroupsnew/docs/index.html.

[17] Birman, K. P. Building Secure and Reliable Network Applications. Greenwich, CT, Manning, 1996.

[18] Birman, K. P., and Joseph, T. A. "Exploiting Virtual Synchrony in Distributed Systems." *Proceedings of the 11th ACM Symposium on OS Principles. Austin, TX.* 1987.

[19] Birman, K. P., van Renesse, R., et al. "The Ensemble Distributed Communication System." http://www.cs.cornell.edu/Info/Projects/Ensemble/.

[20] Chandra, T. D., Hadzilacos, S.,Toueg, S. and Charron-Bost, B. "On the impossibility of group membership." *15th ACM Symposium on Principles of Distributed Computing (PODC).* May, 1996.

[21] Cheriton, D. R., and Zwaenepoel, W. "Distributed process groups in the V kernel." ACM Transactions on Computer Systems, (1985), 3(2), pp. 77-107.

[22] Chockler, G. V., Keidar, I., and Vitenberg, R. "Group Communication Specifications: A Comprehensive Study." ACM Comput. Surv., (2001), 33(4), pp. 1-43.

[23] Dolev, D., et al. "The Xpand Group Communication System." http://www.cs.huji.ac.il/labs/danss/xpand/.

[24] Dolev, D., Malkhi, D. "The Transis approach to high availability cluster communication." Communications of the ACM, (1996), 39(4), pp. 64-70.

[25] Fall, K., Varadhan, K., et al., "The ns Manual," in *The VINT Project*, December, 2003.

[26] Fekete, A., Lynch, N, and Shvartsman, A. "Specifying and Using a Partitionable Group Communication Service." *Proceedings of the Sixteenth Annual ACM Symposium on Principles of Distributed Computing.* August, 1997.

[27] Friedman, R., and van Renesse, R. "Strong and Weak Virtual Synchrony in Horus." Cornell University. Technical Report 95-1537, August 24, 1995.

[28] Hickey, T., Lynch, N. and van Renesse, R. "Specifications and Proofs for Ensemble Layers." TACAS, (1999).

[29] Jacobsen, K., Marzullo, K., and Zhang, X. "Group Membership and Wide-Area Master-Worker Computations." *Proceedings of the 23rd ICDCS.* 2003.

[30] Joint Chiefs of Staff. "Joint Command, Control, Communications, and Computer Systems (C4) Campaign Plan." Department of Defense. September, 2004.

[31] Keidar, I. "A Highly Available Paradigm for Consistent Object Replication." Master's Thesis. Institute of Computer Science, The Hebrew University of Jerusalem, 1994.

[32] Keidar, I., and Khazan, R. "A client-server approach to virtually synchronous group multicast: Specifications and algorithms." MIT Laboratory for Computer Science. Technical Report, 1999.

[33] Keidar, I., and Rajsbaum, S. "A Simple Proof of the Uniform Consensus Synchronous Lower Bound." Info. Processing Letters (IPL), (2003), 85(1), pp. 47-52.

[34] Keidar, I., Sussman, J., Marzullo, K., and Dolev, D. "Moshe: A group membership service for WANs." ACM Transactions on Computer Systems, (2002), 20(3), pp. 1-48.

[35] Khazan, R. "Group Communication as a base for a Load-Balancing, Replicated Data Service." Master's Thesis. Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, 1998.

[36] Khazan, R., and Lynch, N. "An Algorithm for an Intermittently Atomic Data Service Based on Group Communication." *Proceedings of the International Workshop on Large-Scale Group Communication. Florence, Italy.* October, 2003.

[37] Khazan, R., and Yuditskaya, S. "Using Leader-Based Communication to Improve the Scalability of Single-Round Group Membership Algorithms." *10th IEEE Workshop on Dependable Parallel, Distributed, and Network-Centric Systems (DPDNS).* April, 2005.

[38] Khazan, R., Fekete, A., and Lynch, N. "Multicast Group Communication as a Base for a Load-Balancing Replicated Data Service." *12th International Symposium on Distributed Computing (DISC). Andros, Greece.* September, 1998.

[39] Khazan, R. I. "Group Membership: A Novel Approach and the First Single-Round Algorithm." *24th ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing (PODC). Canada.* July, 2004.

[40] Lebras, Maj. Gen. P., "A Vision for the Future: It's Still About People," in *U.S. Air Force Spokesman Magazine*, September, 2003.

[41] Lynch, N. A. Distributed Algorithms, Morgan Kaufmann Publishers, 1996.

[42] Rodeh, O. "Secure Group Communication." Ph.D. Thesis. Computer Science Department, The Hebrew University of Jerusalem, 2001.

[43] Rodrigues, R. "Robust Services in Dynamic Systems." Ph.D. Thesis. Laboratory for Computer Science, MIT, 2005.

[44] Schiper, A. "Failure Detection vs. Group Membership in Fault-Tolerant Distributed Systems: Hidden Trade-Offs." *Proceedings of PAPM-ProbMiV'02, LNCS 2399.* 2002.

[45] Schiper, A., and Toueg, S. "From Set Membership to Group Membership: A Separation of Concerns." Ecole Polytechnique Federale de Lausanne. Technical Report 200371, November 13, 2003.

[46] Shamir, G. "Shared Whiteboard: A Java Application in the Transis Environment." The Hebrew University of Jerusalem. October, 1996.

[47] Steiner, M., Tsudik, G. and Waidner, M. "Diffie-Hellman Key Distribution Extended to Groups." *3rd ACM Conference on Computer and Communications Security.* March, 1996.

[48] Steiner, M., Tsudik, G. and Waidner, M. "CLIQUES: A New Approach to Group Key Agreement." *IEEE International Conference on Distributed Computing Systems.* May, 1998.

[49] Urban, P., and Schiper, A. "Comparing Distributed Consensus Algorithms." *Proc. IASTED Int'l Conf. on Applied Simulation and Modelling (ASM). Rhodos, Greece.* June, 2004.

[50] Urban, P., Shnayderman, I. and Schiper, A. "Comparison of Failure Detectors and Group Membership: Performance Study of Two Atomic Broadcast Algorithms." *Proc. International Conference on Dependable Systems and Networks (DSN).* 2003.

[51] van Renesse, R., Birman, K. P., and Maffeis, S. "Horus: A flexible group communication system." Communications of the ACM, (1996), 39(4), pp. 76-83.

[52] Wiesmann, M., Defago, X., and Schiper, A. "Group Communication Based on Standard Interfaces." *Proc. 2nd IEEE Intl. Symp. on Network Computing and Applications (NCA'03).* April, 2003.

[53] Khazan, R., and Yuditskaya, S. "A Wide Area Network Simulation of Single-Round Group Membership Algorithms." *Proc. 4th IEEE International Symposium on Network Computing and Applications (NCA'05). Cambridge, Massachusetts.* July, 2005.

Appendix A

Table A-1 shows an excerpt of the RON wide-area network traces that we used in our simulations. Each line is a distinct entry, and each entry consists of seven fields [33]:

- **source**, the originator of the probe.
- **dest**, the destination to which the probe was sent.
- **ron**, a flag to denote the RON link type. This was always 0 throughout the RON traces that we used, which means that probes were sent directly on the Internet. The other options are 1, in which the RON link is latency optimized, and 2, in which the RON link is a loss optimized path.
- **send1**, the time at which the source originally sends the probe to the destination.
- **rec1**, the time at which the probe was received on the interface at the destination.
- **send2**, the time at which the probe was sent back to the sender by the destination.
- **rec2**, the time at which the probe was received at source's interface.

The clocks on the RON servers were only roughly synchronized; thus, the time fields are accurate only in relation to other time entries on the same machine. For example, send1 and rec2 are compatible for relative analysis, because both times were computed by the same physical clock. In contrast, send1 and send2 cannot be compared, because they were each computed on different machines, and thus different physical clocks.

Table A-1. RON trace excerpt.

source	dest	ron	send1	rec1	send2	rec2
3237550090	2472938932	0	1027099710.15943	1027099710.38403	1027099710.38409	1027099710.54001
3469047693	3520452188	0	1027099710.25268	1027099710.46760	1027099710.46778	1027099710.48367
2607122173	2472938932	0	1027099710.32466	1027099710.43134	1027099710.43140	1027099710.53701
1115442534	2183470608	0	1027099710.35424	1027099710.42709	1027099710.42717	1027099710.51633
3433608487	304021648	0	1027099710.39149	1027099710.39597	1027099710.39619	1027099710.40035

Appendix B

This sample OTcl code was generated for just three nodes of the RON trace. It represents a subset of the actual code used for our simulation, which constructs a topology consisting of all 16 nodes.

```
set ns [new Simulator]
set n0 [$ns node]
set nesvc0 [new Application/NESvcTrc 3237550090]
set n1 [$ns node]
set nesvc1 [new Application/NESvcTrc 2472938932]
set n2 [$ns node]
set nesvc2 [new Application/NESvcTrc 3469047693]
set tcp1_0 [new Agent/TCP/GCSAgent]
$tcp1_0 set fid_ 10
$tcp1_0 set destin 2472938932
set sink1_0 [new Agent/TCPSink/GCSSink 2472938932 3237550090]
$sink1_0 set nesvc_ $nesvc1
$ns attach-agent $n0 $tcp1_0
$ns attach-agent $n1 $sink1_0
$ns duplex-link $n0 $n1 2Mb 189ms DropTail
$ns connect $tcp1_0 $sink1_0
$nesvc0 attach-agent $tcp1_0
set tcp2_0 [new Agent/TCP/GCSAgent]
$tcp2_0 set fid_ 20
$tcp2_0 set destin 3469047693
set sink2_0 [new Agent/TCPSink/GCSSink 3469047693 3237550090]
$sink2_0 set nesvc_ $nesvc2
$ns attach-agent $n0 $tcp2_0
$ns attach-agent $n2 $sink2_0
$ns duplex-link $n0 $n2 2Mb 98ms DropTail
$ns connect $tcp2_0 $sink2_0
$nesvc0 attach-agent $tcp2_0
set tcp0_1 [new Agent/TCP/GCSAgent]
$tcp0_1 set fid_ 01
$tcp0_1 set destin 3237550090
set sink0_1 [new Agent/TCPSink/GCSSink 3237550090 2472938932]
$sink0_1 set nesvc_ $nesvc0
$ns attach-agent $n1 $tcp0_1
$ns attach-agent $n0 $sink0_1
$ns duplex-link $n1 $n0 2Mb 189ms DropTail
$ns connect $tcp0_1 $sink0_1
$nesvc1 attach-agent $tcp0_1
set tcp2_1 [new Agent/TCP/GCSAgent]
$tcp2_1 set fid_ 21
$tcp2_1 set destin 3469047693
set sink2_1 [new Agent/TCPSink/GCSSink 3469047693 2472938932]
$sink2_1 set nesvc_ $nesvc2
$ns attach-agent $n1 $tcp2_1
$ns attach-agent $n2 $sink2_1
$ns duplex-link $n1 $n2 2Mb 130ms DropTail
$ns connect $tcp2_1 $sink2_1
$nesvc1 attach-agent $tcp2_1
set tcp0_2 [new Agent/TCP/GCSAgent]
$tcp0_2 set fid_ 02
```

```
$tcp0_2 set destin 3237550090
set sink0_2 [new Agent/TCPSink/GCSSink 3237550090 3469047693]
$sink0_2 set nesvc_ $nesvc0
$ns attach-agent $n2 $tcp0_2
$ns attach-agent $n0 $sink0_2
$ns duplex-link $n2 $n0 2Mb 98ms DropTail
$ns connect $tcp0_2 $sink0_2
$nesvc2 attach-agent $tcp0_2
set tcp1_2 [new Agent/TCP/GCSAgent]
$tcp1_2 set fid_ 12
$tcp1_2 set destin 2472938932
set sink1_2 [new Agent/TCPSink/GCSSink 2472938932 3469047693]
$sink1_2 set nesvc_ $nesvc1
$ns attach-agent $n2 $tcp1_2
$ns attach-agent $n1 $sink1_2
$ns duplex-link $n2 $n1 2Mb 130ms DropTail
$ns connect $tcp1_2 $sink1_2
$nesvc2 attach-agent $tcp1_2
$ns at 0.0 "$nesvc0 start"
$ns at 0.1 "$nesvc1 start"
$ns at 0.2 "$nesvc2 start"
$ns run
```

Appendix C

Figure C-1 through Figure C-4 present supplementary RON2 results for All-to-All Sigma vs. Moshe, that reproduce and confirm RON1 results presented in Chapter 6.1. We include these results here for completeness.

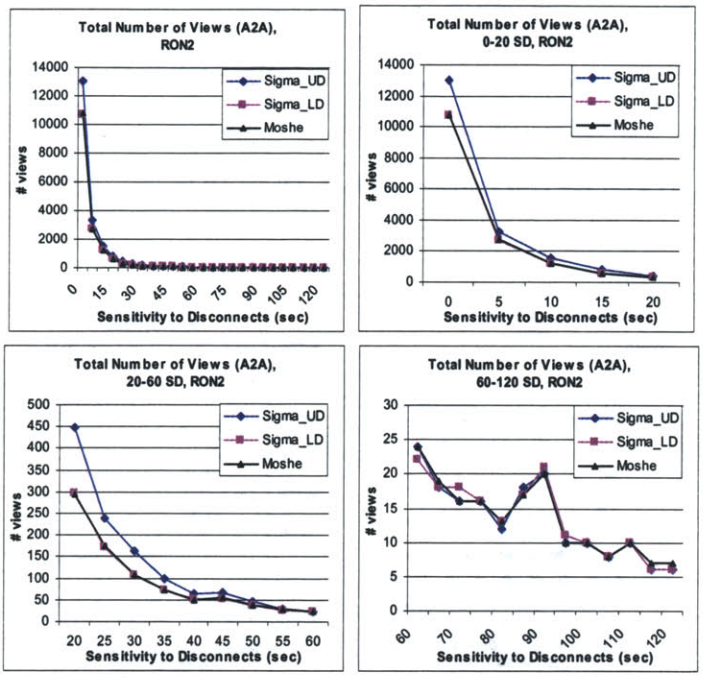

Figure C-1. Total Number of Views, RON2. All-to-All Sigma vs. Moshe. (a) Overall picture (b,c,d) Piecewise analysis.

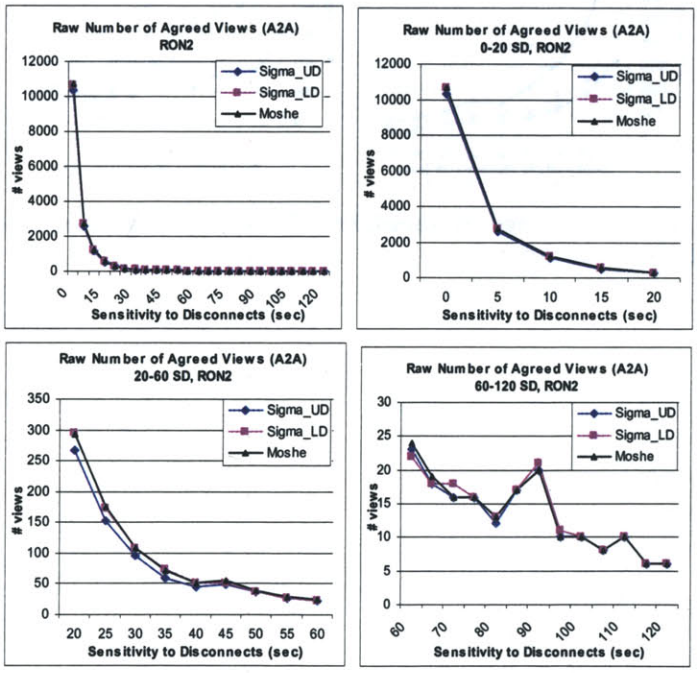

Figure C-2. Raw Numbers of Agreed Views, All-to-All Sigma vs. Moshe, RON2. (a) Overall Picture (b,c,d) Piecewise analysis.

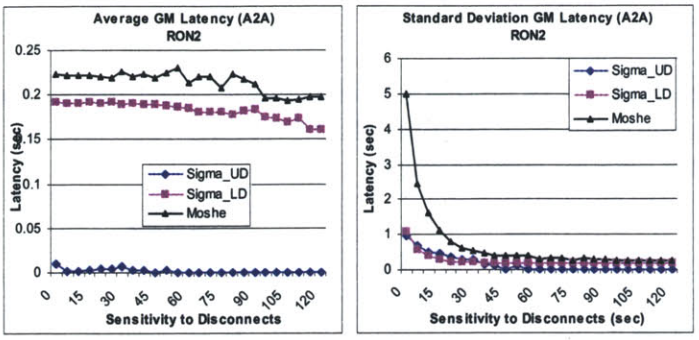

Figure C-3. (a) Average GM Latency, RON2, All-to-All Sigma vs. Moshe. (b) Standard deviations.

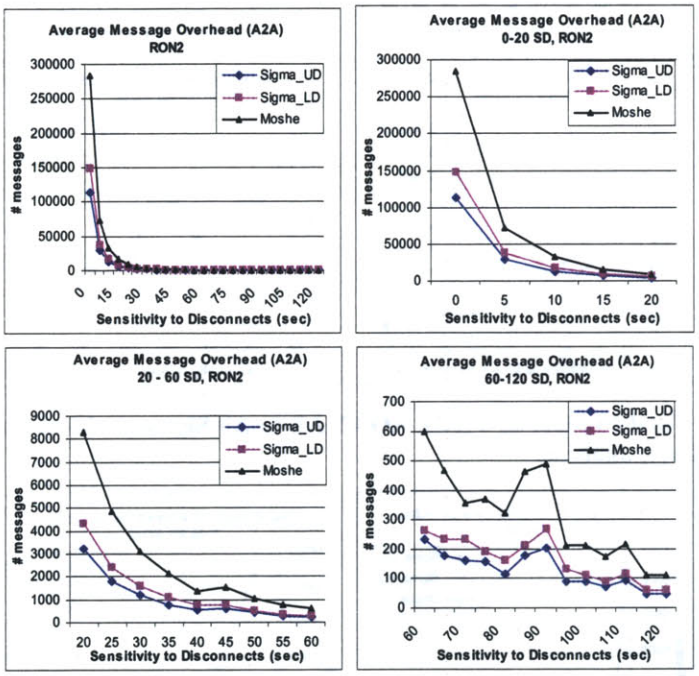

Figure C-4. Average Message Overhead, RON2, All-to-All Sigma vs. Moshe. (a) Overall picture (b,c,d) Piecewise Analysis.

Appendix D

Figure D-1 through Figure D-4 present supplementary RON2 results for Leader-Based Sigma vs. Ensemble, that reproduce and confirm RON1 results presented in Chapter 6.1. We include these results here for completeness.

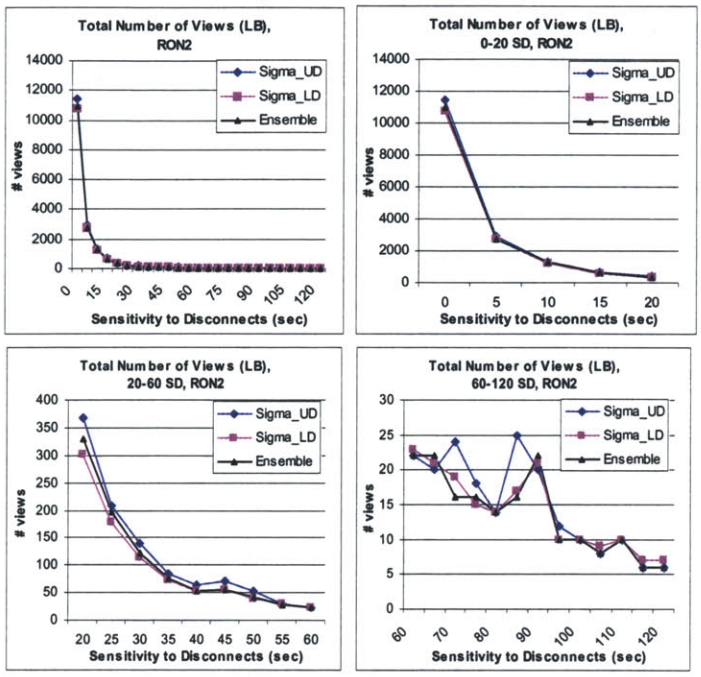

Figure D-1. Total Number of Views, RON2, Leader-Based Sigma vs. Ensemble. (a) Overall picture (b,c,d) Piecewise close-up of trends.

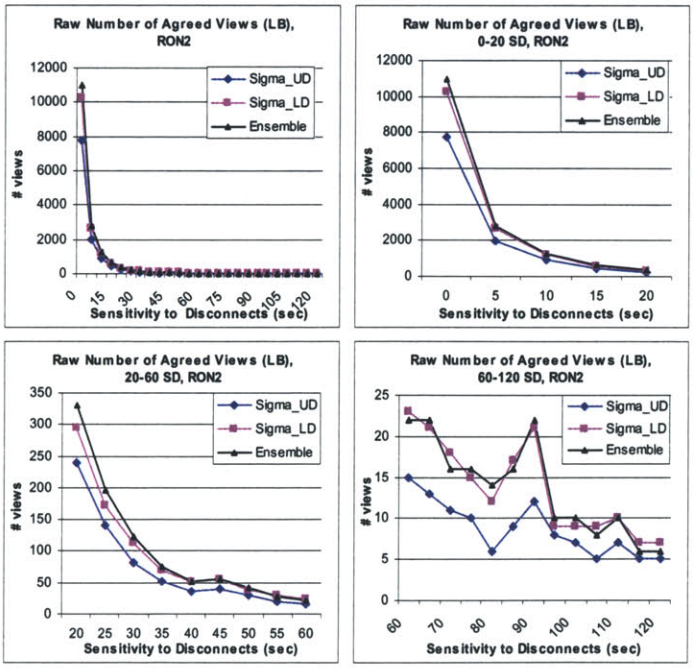

Figure D-2. Raw Number of Agreed Views in RON2, Leader-Based Sigma vs. Ensemble. (a) Overall picture (b,c,d) Clarifying the trends by piecewise analysis.

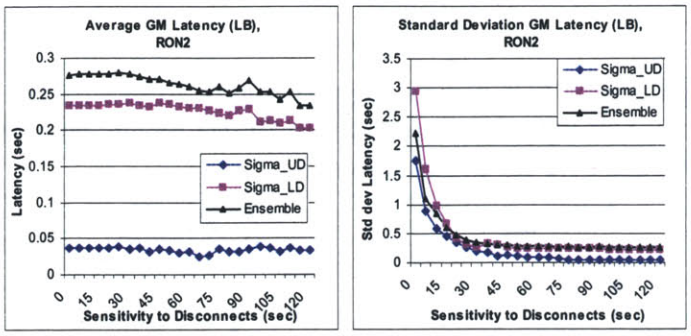

Figure D-3. (a) Average Latency, RON2, Leader-Based Sigma vs. Ensemble. (b) Standard deviations.

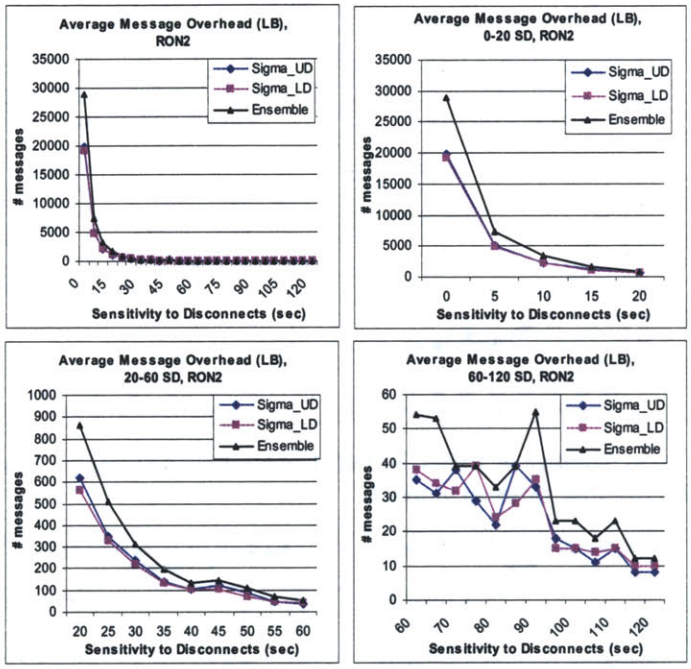

Figure D-4. Average Message Overhead, RON2, Leader-Based. (a) Overall picture (b,c,d) Piecewise analysis.

Appendix E

Table E-1. Raw Agreement and Disagreement Data (All-to-All, RON1).

	A2A Sigma_UD (RON1)			A2A Sigma_LD (RON1)			Moshe (RON1)		
SD	Agreed Views	Disagreed Views	Total Views	Agreed Views	Disagreed Views	Total Views	Agreed Views	Disagreed Views	Total Views
0	10834	1188	12306	11061	38	11117	11086	0	11128
5	2568	102	2707	2601	4	2608	2606	0	2611
10	1118	30	1157	1127	2	1131	1130	0	1130
15	495	16	513	503	0	504	504	0	504
20	220	12	235	223	0	224	228	0	229
25	117	12	129	124	0	125	125	0	125
30	64	12	77	67	0	68	66	0	66
35	37	12	49	41	0	42	44	0	44
40	21	12	33	26	0	27	26	0	26
45	17	12	29	22	0	24	25	0	25
50	18	2	20	19	0	19	19	0	19
55	16	0	17	16	0	16	15	0	15
60	14	0	15	14	0	14	13	0	13
65	18	0	19	17	0	17	17	0	17
70	18	0	19	18	0	18	18	0	18
75	18	0	19	17	0	17	17	0	17
80	14	0	15	13	0	13	13	0	13
85	18	0	19	18	0	18	18	0	18
90	18	0	19	18	0	18	17	0	17
95	11	0	11	11	0	11	11	0	11
100	7	0	7	7	0	7	7	0	7
105	7	0	7	7	0	7	7	0	7
110	7	0	7	7	0	7	7	0	7
115	7	0	7	7	0	7	7	0	7
120	5	0	5	5	0	5	5	0	5

Table E-2. Raw Agreement and Disagreement Data (All-to-All, RON2).

	A2A Sigma_UD (RON2)			A2A Sigma_LD (RON2)			Moshe (RON2)		
SD	Agreed Views	Disagreed Views	Total Views	Agreed Views	Disagreed Views	Total Views	Agreed Views	Disagreed Views	Total Views
0	10339	2266	13021	10669	46	10735	10694	0	10765
5	2587	574	3296	2710	4	2724	2722	0	2735
10	1133	288	1510	1198	2	1202	1206	0	1211
15	524	202	782	572	2	576	574	0	576
20	267	138	447	296	0	297	293	0	293
25	153	58	237	172	0	172	174	0	175
30	96	42	162	108	0	108	107	0	107
35	58	34	99	74	0	74	72	0	72
40	44	14	64	51	0	51	50	0	50

45	49	16	68	50	0	52	55	0	55
50	36	8	47	37	0	37	38	0	38
55	26	2	29	27	0	27	29	0	29
60	23	0	24	22	0	22	24	0	24
65	18	0	18	18	0	18	19	0	19
70	16	0	16	18	0	18	16	0	16
75	16	0	16	16	0	16	16	0	16
80	12	0	12	13	0	13	13	0	13
85	17	0	18	17	0	17	17	0	17
90	20	0	20	21	0	21	20	0	20
95	10	0	10	11	0	11	10	0	10
100	10	0	10	10	0	10	10	0	10
105	8	0	8	8	0	8	8	0	8
110	10	0	10	10	0	10	10	0	10
115	6	0	6	6	0	6	6	0	7
120	6	0	6	6	0	6	6	0	7

Table E-3. Raw Agreement and Disagreement Data (Leader-Based, RON1).

	LB Sigma_UD (RON1)			LB Sigma_LD (RON1)			Ensemble (RON1)		
SD	Agreed Views	Disagreed Views	Total Views	Agreed Views	Disagreed Views	Total Views	Agreed Views	Disagreed Views	Total Views
0	8255	32	11566	10629	2	11091	8760	0	8761
5	1966	0	2678	2516	0	2612	2432	0	2432
10	856	0	1163	1092	0	1137	1109	0	1109
15	373	0	524	495	0	507	499	0	499
20	168	0	241	216	0	229	235	0	235
25	94	0	134	119	0	126	127	0	127
30	48	0	73	67	0	67	71	0	71
35	33	0	51	41	0	43	43	0	43
40	16	0	27	26	0	27	27	0	27
45	16	0	25	23	0	23	25	0	25
50	12	0	19	22	0	22	23	0	23
55	11	0	15	15	0	15	15	0	15
60	12	0	15	13	0	13	13	0	13
65	16	0	19	17	0	17	17	0	17
70	15	0	17	17	0	17	17	0	17
75	15	0	19	17	0	17	17	0	17
80	11	0	13	13	0	13	13	0	13
85	15	0	19	19	0	19	17	0	17
90	14	0	17	17	0	17	17	0	17
95	9	0	11	11	0	11	11	0	11
100	6	0	7	7	0	7	7	0	7
105	6	0	7	7	0	7	7	0	7
110	6	0	7	7	0	7	7	0	7
115	6	0	7	7	0	7	7	0	7
120	4	0	5	5	0	5	5	0	5

Table E-4. Raw Agreement and Disagreement Data (Leader-Based, RON2).

	LB Sigma_UD (RON2)			LB Sigma_LD (RON2)			Ensemble (RON2)		
SD	Agreed Views	Disagreed Views	Total Views	Agreed Views	Disagreed Views	Total Views	Agreed Views	Disagreed Views	Total Views
0	7756	52	11441	10270	4	10707	10949	0	10949
5	1982	0	2892	2624	0	2725	2758	0	2758
10	884	2	1303	1176	0	1215	1268	0	1268
15	431	0	668	581	0	595	636	0	636
20	240	2	368	295	0	302	330	0	330
25	141	0	207	172	0	178	196	0	196
30	82	0	140	112	0	114	122	0	122
35	52	0	84	70	0	73	76	0	76
40	35	0	63	52	0	53	52	0	52
45	39	0	72	56	0	56	56	0	56
50	29	0	52	38	0	38	42	0	42
55	20	0	30	29	0	29	28	0	28
60	15	0	22	23	0	23	22	0	22
65	13	0	20	21	0	21	22	0	22
70	11	0	24	18	0	19	16	0	16
75	10	0	18	15	0	15	16	0	16
80	6	0	14	12	0	14	14	0	14
85	9	2	25	17	0	17	16	0	16
90	12	0	20	21	0	21	22	0	22
95	8	0	12	9	0	10	10	0	10
100	7	0	10	9	0	10	10	0	10
105	5	0	8	9	0	9	8	0	8
110	7	0	10	10	0	10	10	0	10
115	5	0	6	7	0	7	6	0	6
120	5	0	6	7	0	7	6	0	6

Appendix F

AMO = Average Message Overhead
FA = Moshe Fast Agreement
SA = Moshe Slow Agreement

Table F-1. Average Message Overhead Raw Data for All-to-All study, including FA vs. SA breakdown for Moshe. Note that Sigma_LD's AMO is very similar to Moshe's FA AMO, often with exactly the same number of messages per node.

	RON1					RON2				
SD	Sigma_UD AMO	Sigma_LD AMO	Moshe FA AMO	Moshe SA AMO	Moshe AMO	Sigma_UD AMO	Sigma_LD AMO	Moshe FA AMO	Moshe SA AMO	Moshe AMO
0	115657	153918	153918	147905	301823	112615	147626	147626	136656	284283
5	26861	35778	35826	35350	71177	28367	37142	37208	35555	72763
10	11652	15458	15441	15352	30794	12675	16620	16855	15930	32786
15	5123	6907	6845	6886	13732	6094	8070	8183	7718	15901
20	2308	3048	3200	3194	6394	3218	4288	4301	4011	8312
25	1234	1708	1751	1695	3447	1800	2409	2530	2317	4847
30	701	909	993	923	1916	1206	1599	1600	1484	3084
35	422	555	621	581	1202	770	1110	1062	1049	2111
40	244	360	360	343	703	532	737	686	652	1339
45	204	324	348	342	690	593	737	763	749	1512
50	179	239	239	218	457	414	497	538	519	1058
55	147	201	179	180	360	278	338	378	401	780
60	125	174	152	131	284	234	263	304	293	598
65	168	207	207	181	388	177	234	231	236	468
70	170	229	229	214	443	161	233	189	168	357
75	169	207	207	195	402	157	192	192	176	369
80	124	152	152	131	284	115	160	157	166	323
85	167	229	229	218	447	178	214	233	228	462
90	169	229	207	163	370	202	269	247	241	488
95	94	132	132	118	251	90	133	111	110	211
100	58	81	81	75	157	90	111	111	110	211
105	58	81	81	75	157	71	87	87	86	173
110	58	81	81	75	157	93	114	114	102	216
115	58	81	81	75	157	48	60	60	50	111
120	36	54	54	60	114	48	60	60	60	111

Table F-2. Average Message Overhead Raw Data for Leader-Based study. Provided for completeness.

.

	RON1			RON2		
SD	LB Sigma_UD AMO	LB Sigma_LD AMO	Ensemble AMO	LB Sigma_UD AMO	LB Sigma_LD AMO	Ensemble AMO
0	20375	19940	30002	19833	19191	28946
5	4748	4680	6625	5023	4880	7303
10	2065	2038	3003	2264	2201	3349
15	927	911	1348	1146	1105	1671
20	423	421	631	621	561	862
25	234	232	339	349	329	510
30	126	118	187	235	218	313
35	86	76	111	141	135	196
40	45	48	68	105	102	133
45	40	41	62	121	103	144
50	31	40	58	85	68	107
55	24	24	36	49	49	70
60	23	20	31	35	38	54
65	30	27	42	31	34	53
70	27	27	42	38	32	39
75	30	27	42	29	39	39
80	20	20	31	22	24	33
85	30	32	42	39	28	39
90	27	29	42	33	35	55
95	17	17	26	18	15	23
100	10	12	16	15	15	23
105	10	10	16	11	14	18
110	10	10	16	15	15	23
115	10	10	16	8	10	12
120	7	7	10	8	10	12