Practical Wide Area Group Communication

by

Jonathan Robert Stanton

A dissertation submitted to Johns Hopkins University in conformity with

the requirements for the degree of Doctor of Philosophy

Baltimore, Maryland

May, 2002

©Jonathan Robert Stanton

All rights reserved

Abstract

Group communication systems have a successful history of providing useful services for the development of distributed applications. Several different semantic models for such systems have been proposed, analyzed, and studied, and a number of systems have been built and used in both commercial and research software systems. However, group communications has also been viewed by many as a niche service whose limitations make it useful for only very restricted classes of applications. Two of the most significant limitations have been their dependence on low-latency, high-bandwidth networks and the complexity or inflexibility of the systems.

This thesis presents a formal model specifying the services that a wide-area group communication system could support. It then discusses the challenges wide-area networks, such as the Internet, present for group communications and how these challenges were met by the design and implementation of a group messaging system called "Spread." Finally the performance of the system is evaluated and the impact of specialized network protocols is discussed.

Advisor: Professor Yair Amir

Reader: Professor Scott Smith

Acknowledgments

This work would not have been possible without the support of many people. To all of you, Thank You.

To my wife, Allison. You have always been beside me, supporting me with your love, ideas, and friendship.

To my parents who have provided me with love, encouragement, direction, and help when I needed it. To my brother, Jamie who is the best brother and friend I could want. To Stewart, Jane and Sandy for welcoming me into their family and giving me a second home during this journey.

To my advisor Yair who has taught me not only how to research and teach, but through many long conversations more about the world and people and friendship then I ever expected to learn in graduate school. To his wife Michal who is a partner in Spread and without whom this work would not be a success.

To Prof. Scott Smith who graciously agreed to serve as a reader on my dissertation and has been extremely supportive over the years. To Prof. Baruch Awerbuch who has taught me the importance of ideas and the power of great research.

To all of my friends and collegues in the lab, Theo for his friendship and engaging conversations; Cristina for her discipline, support and critical thoughts; Claudiu for our always interesting discussions and work together; Ciprian for his focus on precise thinking and help with administering the lab; John for long conversations on the semantics of groups and his help over the years; Jacob for keeping us all together; Ryan B., David S., Herb,

Dave H., Chuck and Tripurari for your making the lab a better place.

To my fellow students from the department who showed me that graduate students can have fun and be successful at the same time - Rich, Grace, Andy, Beth, Lidia, Andrew, and Mihai.

Table of Contents

Abstract ii

Acknowledgments iii

1 Introduction 1

1.1 Summary of Contributions 4

1.2 Structure of thesis 5

1.3 Related work 6

1.3.1 Group Communication Systems 6

1.3.2 Semantics and Models 8

1.3.3 Group Membership Algorithms 9

1.3.4 Applications of Group Communication 10

1.3.5 Reliable multicast and network protocols 10

2 Specifications and Environment 12

2.1 Failure Model 13

2.2 System and Layer Specifications 15

2.2.1 Basic Model and Definitions 18

2.2.2 System Assumptions 22

2.2.3 Extended Virtual Synchrony 23

2.2.3.1 Membership Safety Properties 24

2.2.3.2 Message Properties 25
2.2.3.3 Liveness Properties 29
2.2.4 EVS-Client Semantics 30
2.2.5 EVS-Server Semantics 31
2.2.6 Network Specification 33
2.2.7 Reliable Link Semantics 34
2.2.8 Unreliable Multicast Semantics 35
2.3 Discussion 35
2.3.1 Minimal vs. “Best Effort” Implementations 35
2.3.2 Service Interactions 35
2.3.3 Obsolete Views 36

3 Spread System Architecture 38
3.1 History 39
3.2 Architecture 40
3.3 Overlay Network, Routing, and Reliability 43
3.4 Group Scalability and Services 46
3.4.1 Light-Weight Groups 49
3.4.2 Message Pruning 51
3.5 End-To-End Principle 53

4 Network Protocols 57
4.1 Ring Protocol 58
4.2 TCP Protocol 61
4.3 Hop Protocol 62
4.4 Evaluation 66
4.4.1 General Comparison 68
4.4.2 Performance Comparison 68

5 Global Ordering and Safety **73**

5.1 Ordering Algorithm . . . 76

5.1.1 Proof of Ordering Properties . . . 79

5.2 Performance . . . 81

6 Daemon and Group Membership **86**

6.1 Daemon Membership Algorithm . . . 87

6.1.1 States . . . 88

6.1.2 Types of Messages . . . 88

6.1.3 Possible Events . . . 90

6.1.4 Start State . . . 93

6.1.5 OP State . . . 95

6.1.6 Segment State . . . 95

6.1.7 Represented State . . . 99

6.1.8 Gather State . . . 99

6.1.9 Statetrans State . . . 103

6.1.10 EVS state . . . 104

6.2 Proof of Daemon Membership . . . 110

6.3 Evaluation of Daemon Membership . . . 123

6.4 Process Group Membership Algorithm . . . 124

6.4.1 States . . . 124

6.4.2 Types of Messages . . . 125

6.4.3 Possible Events . . . 125

6.4.4 GOP State . . . 127

6.4.5 GTRANS State . . . 128

6.4.6 GGATHER State . . . 136

6.4.7 GGT State . . . 141

6.5 Proof of Group Membership . . . 143

7 Conclusion **158**

References **160**

Curriculum Vita **169**

List of Tables

4.1 Throughput using TCP and several network configurations. 67

4.2 Link Latency (Mae East to UCSB). 69

5.1 Latency of Agreed Messages. 83

List of Figures

2.1 System Architecture 16

3.1 Sites and Links 40

3.2 Spread Software Architecture 42

3.3 Overlay Network Architecture 43

3.4 Scalability with Groups 50

3.5 Scalability with Groups 51

4.1 Hop Link: Sender protocol 63

4.2 Hop Link: Receiver protocol 65

4.3 Experimental Network for Tree Network 66

4.4 Map of Chain Network 70

4.5 Latency of Messages under Load (Comparing TCP and Hop) 71

5.1 Deliver_Mess function 79

5.2 Deliver_Safe_Mess function 80

5.3 Wide Area Chain Network 84

5.4 Wide Area Latency of Ordered Messages 85

6.1 Membership State Machine 94

6.2 Daemon Membership – Start State 94

6.3 Daemon Membership – OP State 96

6.4 Daemon Membership – ProcessNewMemb function 97

6.5 Daemon Membership – DeclareMembLoss function 97
6.6 Daemon Membership – SendJoin function 98
6.7 Daemon Membership – Seg State 98
6.8 Daemon Membership – Rep State 100
6.9 Daemon Membership – Gather State 102
6.10 Daemon Membership – Gather State (Part 2) 103
6.11 Daemon Membership – Create_Send_NewMemb function 103
6.12 Daemon Membership – SendStateTrans function 104
6.13 Daemon Membership – Statetrans State 105
6.14 Daemon Membership – Statetrans State (Part 2) 106
6.15 Daemon Membership – EVS State 106
6.16 Daemon Membership – DeliverMembership function 107
6.17 Group State Machine 125
6.18 Group Membership – GOP State 129
6.19 Group Membership – GOP State (Part 2) 130
6.20 Group Membership – GTRANS State 133
6.21 Group Membership – GTRANS State (Part 2) 134
6.22 Group Membership – GTRANS State (Part 3) 137
6.23 Group Membership – GGATHER State 138
6.24 Group Membership – ComputeNotify function 139
6.25 Group Membership – GGATHER State (Part 2) 140
6.26 Group Membership – GGT State 142
6.27 Group Membership – GGT State (Part 2) 142

Chapter 1

Introduction

> *Observe constantly that all things take place by change, and accustom thyself to consider that the nature of the universe loves nothing so much as to change things which are and to make new things like them. For everything that exists is in a manner the seed of that which will be.*
>
> – The Meditations of Marcus Aurelius (Book 4 - XXIX)

One of the main difficulties in distributed algorithms and systems is that the environment in which distributed software runs is very complex and has an extremely large number of possible failure scenarios. Group communication systems attempt to hide that complexity by providing a model that constrains the diversity of failures and enforcing a type of synchronous behavior on messages and failures. Although Group communication systems succeed in the theoretical sense, in that they can correctly provide well-defined message semantics that are simpler then the unbounded behavior of asynchronous, multi-node distributed systems, they have not clearly succeeded in the practical sense. Group communication systems are not widely used in real applications or viewed by developers outside the research community as a well-understood, reliable building block, upon which complex applications can be built.

Several plausible reasons exist for this lack of acceptance. First, although group com-

munications is a well-developed research field, it is quite young, only having begun 15 years ago. The beginning of the field is usually marked by the Virtual Synchrony paper by Birman and Joseph in 1987 and the ISIS system [Bir86] in 1986. Flexible, general purpose systems that are mature, well-tested, and evolved have not had much time to develop. Practical systems existed from the very beginning of the field, starting with the very successful ISIS system. The solutions offered by systems, however, were often narrowly focused on certain applications, or specific, limited classes of problems.

Second, even now, agreement on what the appropriate models and semantics that systems should support does not exist. All early models have been shown to be flawed, or limited in some way [ACBMT95]. More recent specifications [BDM99, FLS97] avoid these flaws, however, they enforce properties that limit the performance of the system, and are, therefore, non-optimal. More generally, an understanding of what properties are needed for applications other than consistent replication has not been developed.

The debate over the usefulness of causal and totally ordered virtually synchronous communication systems has occured since the first such system. Cheriton and Skeen [CS93] claimed causal and total order communication was not applicable to several types of distributed applications. Birman [Bir93b] responds by providing a number of applications that can take advantage of ordered communication to become fault-tolerant and have a simplified programming model. Schneider's survey [Sch90] presents how to use total ordered messages to manage a system with distributed state consistently without locks.

When examining distributed systems that function over wide-area networks, one of the most important constraints is the latency of communications. While in local area networks, message latency is significantly slower than local memory latency. It is still low enough that for many applications involving user interaction or moderate amounts of messaging, the latency does not contribute to user-experienced delays or become the bottleneck of the system. In wide-area networks the network latency becomes critical for any application involving interaction and can significantly affect even non-interactive applications by lim-

iting their throughput and consuming large amounts of memory to store messages.

Network latency critically impacts the design of network and wide-area distributed systems not only because it causes delay, but also, and more importantly, because the delay can not be overcome by future technological advances, faster hardware, or better designed networks. The delay is fundamental to wide-area networks, as long as the physical propagation delay of information remains limited by the speed of light. Sound engineering and improved network hardware can reduce the latency to the minimum propagation delay. The minimum delay, however, is still very significant in that it can be hundreds of times the local area network latency even for networks just within the USA.

Many distributed algorithms, effective in networks with low latency, become slower in proportion with the network latency. This does not mean that all distributed algorithms increase in cost as the network latency increases. Many algorithms may increase only slowly, or may be modified to increase slowly.

Supporting wide area group communication systems involves not only the obvious scaling to more nodes, but more importantly, it means adapting to a more hostile networking and process environment in which less can be trusted. In such environments, latency can be high and widely varying. The system can not assume a flat (uniform) process group with no regards to network locality and resource availability.

An effective implementation of a general group communication specification is not a trivial problem. Because group communication systems require both providing network services and implementing distributed algorithms, even simple specifications often require significant effort to develop functional implementations. The high messaging and computation costs required by the first generation of systems implementing group communication specifications, often led observers to believe that strong semantics group messaging was fundamentally impractical due to the high cost. This belief was not based on fundamental performance constraints of the model, but rather the performance of the initial implementations. Later work [AMMS+95, ADKM92, FvR97] improved the performance of group

communication systems significantly and laid the groundwork for efficient, high performance, scalable systems such as Spread.

A useful group communication system needs to provide a simple and consistent API for membership, messaging, ordering, and reliability. It also needs flexible service semantics that can be customized to fit application needs. At the same time it is more difficult to maintain and understand a monolithic system. One approach is to provide one external interface, with a number of internal interfaces between self-contained components, each providing a portion of the services that make up the entire system. This gives some of the benefit of modularity without losing the simplicity and ease of use of a single system.

The usefulness of group communication systems in supporting consistent replicated state and reliable multi-party coordination is well known. Debate over the usefulness of group communication systems for other applications and as a general model for distributed applications is widespread. Many argue the strong semantics of traditional group communication models makes them unscalable or too slow for many applications. Researchers generally agree that most existing group communication systems interfaces and semantics are too complex and too difficult to understand.

The problem this thesis addresses is how to provide a useful set of services, specifically wide-area reliable multicast, totally ordered messages, and a view-oriented membership service for group-oriented messaging, in the context of wide-area networks.

1.1 Summary of Contributions

This dissertation studies a method for constructing scalable group communication systems and presents a new architecture for building high performance group communication systems. This architecture includes algorithms for strong group membership, ordering and reliability. The architecture is realized in the implementation of a group messaging toolkit called Spread which provides high performance messaging on both local and wide area

networks.

This dissertation makes the following contributions to distributed systems and networking research in the field of Computer Science:

- the first formal specification of a complete Extended Virtual Synchrony-based model for reliable group communications including light-weight FIFO and reliable messaging services, and scalable, dynamic joins and leaves of processes.
- a flexible architecture and programming interface that supports both simple (reliable multicast without membership) and complex (Extended Virtual Synchrony) services.
- the design of a scalable, hierarchical membership service and a proof that this design correctly provides the Extended Virtual Synchrony specification.
- a scalable overlay network for routing, dissemination, reliability, and flow control of group multicast messages.
- a high performance messaging system, Spread, that provides group communication services to large numbers of clients across a wide area network.

1.2 Structure of thesis

This thesis is structured to present three views of a wide-area group communication system. One might characterize these views as a journalist, "What is a group communication system?"; "How does one build a correct, high-performance group communication system?"; and "Why is a wide-area group communication system useful?". First, we explore the "What" by presenting the theoretical specification of a model for group membership, message ordering, and reliability supporting wide-area networks. This model is independent of any particular system, algorithm or protocol and specifies the behavior that applications using the service can expect, as well as the behavior of certain sub-components that make up a group communication system. Second, the "How" is explained in a detailed discussion of the actual architecture, protocols, and implementation of a complete group communica-

tion system implementing the model. Third, the "Why" is addressed through experimental evaluation of the system.

Chapter 2 includes the complete specification of several models of group communication systems and network protocols. In Chapter 3 Spread is introduced as the wide-area group communication system that implements the above specification. Chapter 3 also includes Spread's history, overall architecture, and design. Chapter 4 explains the algorithms and implementation of a wide-area reliable overlay network that disseminates messages between the servers. Chapter 5 presents the algorithm for global ordering of messages, and discuss its performance during intervals when the set of participating servers is stable. Chapter 6 presents, and proves the correctness of, two complete group membership protocols that implement the EVS specification among first the set of servers, and then among all of the processes in the system. Finally Chapter 7, concludes and discusses the overall impact of this work.

1.3 Related work

1.3.1 Group Communication Systems

Group communication systems have a well developed history beginning with the seminal work on Virtual Synchrony by Birman and Joseph[BJ87] and the development of the ISIS System[Bir86, Bir93a, BR94] in the late 1980's and early 1990's. Several other early systems such as Amoeba [KT91], Consul [MPS93], xAMp [RV92], Phoenix [MFSW95], and Newtop [EMS95] provided services such as total order messages, membership, and some supported Virtual Synchrony.

Since then a number of systems have been developed such as Transis [ADKM92], Horus [RBM96], Totem [AMMS^{+}95, Aga94], RELACS [BDGB94] Ensemble [Hay98], and RMP [WMK94].

Wide Area Group Communications A few of these systems have added some type of support for either wide-area group communication or multi-LAN group communication. The Hybrid paper [RFV96] discusses the difficulties of extending LAN oriented protocols to the more dynamic and costly wide-area setting. The Hybrid system has each group communication application switch between a token based and symmetric vector based ordering algorithm depending on the communication latency between the applications. While their system provides a total order using whichever protocol is more efficient for each participant, Hybrid does not handle partitions in the network, or provide support for orderings other then total.

Several new group communication protocols designed for such wide area networks have been proposed [KK00, AMMSB98, AS98, ADS00, JJS99] and continue to provide the traditional strong semantic properties such as reliability, ordering, and membership.

These systems predominantly extend a flow control model previously used in local area networks, such as the Totem Ring protocol[AMMSB98], or adapt a window-based algorithm to a multi-sender group[HvR95, ADS00].

Totem The Multiple-Ring Totem protocol [AMMSB98] allows several rings to be interconnected by gateway nodes that forward packets to other rings. This system provides a substantial performance boost compared to a single-ring on large LAN environments, but keeps the assumptions of low loss rates and latency and a fairly similar bandwidth between all nodes that limit its applicability to wide-area networks. The latency of the Totem multiple ring protocol has been theoretically analyzed in [TMMS97].

Transis The Transis wide-area protocols Pivots and Xports by Nabil Huleihel [Hul96] provide ordering and delivery guarantees in a partitionable environment. Both protocols are based on a hierarchical model of the network where each level of the hierarchy is partitioned into small sets of nearby processes and each set has a static representative who is also a member of the next higher level of the hierarchy. Messages can be contained in

any subtree if the sender specifies that subtree. The Congress work [ACDK98] approaches the problem of providing wide-area membership services separately from actual multicast and ordering services, and provides a general membership service that can provide different semantic guarantees.

Inter-group Router The inter-group router model[JJS99] aggregates local ordering protocols into global ordering guarantees. Each local instance of a group is connected with other instances through wide-area routers which provide ordered channels and groups. The paper provides analysis of which orderings are required in the wide area groups in order to preserve FIFO and Total order among the aggregated groups.

1.3.2 Semantics and Models

Several different models for what properties group communication systems should provide were developed. The first, was Virtual Synchrony [BJ87] which initiated the idea of reliable groups with membership. Later other researchers developed Extended Virtual Synchrony [MAMSA94] that provided semantics for partitionable systems.

One of the most fundemental problems in group communication systems is providing an agreed membership view. Providing such a view requires some sort of consensus which has been proven to be impossible in asynchronous systems [FLP85]. The idea of Failure detectors as a way to solve Consensus was first proposed by Chandra and Toueg [CT96].

A number of specifications [JFR93, BDM99, BDMS98, DMS96, FLS01, MAMSA94, HLvR99] for a group membership service have been developed.

One of the limitations of virtual synchrony is that the application must not send messages prior to a membership change for a period of time whose length is dependent on the latency of the network. This result, along with a potential solution, appears in[FvR95]. Here the models of Strong and Weak Virtual Synchrony are defined and the approach of sending a “suggested view” a superset of the final view is proposed as a method to allow

messages to be sent at any time around a membership change.

The concept of a light-weight group that exists on top of a heavy-weight group membership protocol was explored in [Pow91, RGS+96]. The advantage of light-weight groups is they improve the scalability of the system in terms of the number of groups that can efficiently be supported. Each group does not need to run a full membership algorithm every time the set of members changes but rather the heavy-weight algorithm runs once, and translates the results for all of the light-weight groups. The main disadvantage that was identified was the risk of interference from failures in the heavy-weight group.

Structured VS [GVvR96] presents a technique that added a two-level hierarchy to the Horus implementation of Virtual Synchrony and decreased the number of messages required to compute message stability so the system could support more processes.

1.3.3 Group Membership Algorithms

More recent work in this area focuses on scaling group membership to wide-area networks [ACDK98, KSMD00].

The client-server approach taken by [KSMD00] uses a set of membership servers who maintain membership views on the behalf of clients and introduces a novel virtual synchrony membership algorithm. This algorithm can achieve agreement on the view identifier in only one round of network communication in the common cases and three rounds in the worst case.

This approach separates the membership service from the associated multicast, ordering, and reliability services. The Congress [ACDK98] and Moshe [KSMD99, Now98] systems take this approach. One distinction between the Moshe work and this thesis is that in the Congress and Moshe approach, membership tracks client processes, while in this thesis the membership task is split into maintaining a stable set of servers and a light-weight client membership that does not require participation of the clients.

1.3.4 Applications of Group Communication

Schiper and Raynal[SR96] argue that to easily support fault-tolerant transactions with Group communication systems, the Group communication systems need to support multi-group multicast.

Other typical applications are state-machine replication [FV97a, KD96, ADMSM94, Ami95, FLS97, KFL98], replicated distributed objects [FV97b, MDB00, Mon00, MMSN98, Maf95], distributed transactions and database replication [GS95, KA98, Kei94, KA00], resource allocation [SM98], system management [ABCD96], distributed logging [ASSS01], monitoring [ASAWM99], and highly available servers [MP99, ADK99].

Group communication systems are also used for audio and video conferencing [CHRC97] and collaborative computing [BFHR98, ACDK97, RCHS97, KCH98].

Shared memory using GC [Fri95]. The Oasis system implements distributed shared memory on top of a group communications system[WLF00, WLF01].

1.3.5 Reliable multicast and network protocols

IP-Multicast is actively developed to support Internet wide unreliable multicasting and to scale to millions of users. Many reliable multicast protocols which use IP-multicast have been developed, such as SRM [FJL+97], RMTP [LP96], Local Group Concept (LGC) [Hof96], and HRMP [GGLA97].

The development of reliable protocols over IP-Multicast has focused on solving scalability problems, such as ACK or NACK implosion and bandwidth limits, and providing useful reliability services for multimedia and other isochronous applications. Several of these protocols have developed localized loss recovery protocols. SRM uses randomized timeouts with back-off to request missed data and send repairs, which minimizes duplicates. SRM has enhancements to localize the recovery, using the TTL field of IP-Multicast to request a lost packet from nearer nodes first, and then expands the request if no one close has the packet. Several other variations in localized recovery such as using administrative

scope and separate multicast groups for recovery, are discussed in [FJL+97].

Other reliable multicast protocols like LGC use the distribution tree to localize retransmits to the local group leader who is the root of some subtree of the main tree. RMTP also uses "Designated Receivers" (DR) who act as the head of a virtual subtree to localize recovery of lost packets and provides reliable transport of a file from one sender to multiple receivers located around the world. RMTP is based on the IP-Multicast model, but created user-level multicast through UDP and modified *mrouted* software. RMTP did not examine the tradeoffs in link protocols discussed in this paper because RMTP handles reliability over the entire tree, with the DR's only acting as aggregators of global protocol information. The system described in this thesis also provides localized recover. The proposed system has additional information about the group membership and how messages are disseminated, allowing more precise local recovery to get the packet from the nearest source.

HRMP [GGLA97] is a reliable multicast protocol which provides an efficient local reliability based on a ring, while using standard tree-based protocols such as ack trees to provide reliability between rings. This work theoretically analyzes the predicted performance of such a protocol and shows it to be better then protocols utilizing only a ring or a tree. This thesis actually uses a ring protocol for local area networks for many of the same reasons as HRMP.

Chapter 2

Specifications and Environment

This chapter presents a formal specification of a group communication system and how that specification interacts with the surrounding environment. This formal specification of the system allows a developer or researcher to understand the properties of the system without having to understand, or examine the implementation. The specification also provides a formal model for reasoning about the correctness of the system. The protocols can then be studied to see whether or not they actually provide the specified properties.

Often no one "right" formal specification of a distributed system can be defined. Each specification involves tradeoffs among the semantics provided, how strictly those semantics are enforced, and the performance of an implementation of the specification.

The specification provided here focuses on several commonly accepted properties of group communication systems and combines them into a complete specification that is lighter weight then existing specifications. It is more flexible and so it is able to be implemented in a more efficient way. The specification is still, however, strong enough to solve any of the traditional fault-tolerant consistent replication problems discussed in the literature [Ami95, AT01].

Group communication systems usually support one of two group models, open-groups or closed-groups. In the open-group model clients can send messages to a group without

joining the group. The client will not receive these messages since they have not joined the group to which the messages were sent. Despite the sender not being in the group, the messages will support the same semantics with regards to the members of the group as a message sent by one of the current members. In a closed-group model, clients can only send messages to groups in which they are members.

Many specifications of group membership and virtual synchrony models assume a closed-group model. This limits the usefulness of the model as open groups map efficiently to many common designs such as client-server or request-response applications. Open groups are a more general model as they can efficiently emulate closed-groups by not sending to groups an application has not joined, while closed-groups cannot emulate open-groups send-without-receiving service or without the sender receiving messages it does not need or want.

This specification adds connect and disconnect events to support the open-group model. Instead of the model assuming joining a group is the initial event a process sees, the first event is a connect event which is not related to any particular group. After the connect is complete, the client can send messages, but will not receive any messages.

2.1 Failure Model

This thesis discusses a *distributed system*. A *distributed system* can be defined as a group of processes executing on one or more computers and coordinating actions by exchanging messages [Bir96]. Processes communicate via asynchronous multicast and unicast messages. Messages can be lost.

The system can experience process crashes and recoveries. A crash of any portion of the running process, such as the application or a communication library is considered a process crash. A machine crash can be treated as the crash of all the application processes as well as the group communication server. Each process can detect the crash of any other

process on the same machine using standard operating system services.

Additionally, the system can suffer network partitioning failures in which the network splits into disconnected subnetworks. When the network recovers from such a failure, the disconnected components merge into a larger connected component. Unlike processes from connected components, processes from disconnected components can not exchange messages.

More formally, given a set of processes S, and potential bidirectional links between every pair of processes in S, a process p can be either alive or dead and a link $\overline{pq}$ where $p, q \in S$ can be either up or down.

A *connected component* C is defined as a set of alive processes that have direct bidirectional communication links in the up state to all of the other processes in C but all communication links to processes in $S \setminus C$ are in the down state.

This thesis assumes that message corruption is masked by a lower layer. Byzantine failures are not considered.

It is well known that solving the consensus problem is impossible in an asynchronous network [FLP85]. The Group Membership problem has also been shown to be unsolvable in asynchronous networks with failures [CHTCB96]. Both of these results rely on the fact that certain loss or failure patterns are possible, and in executions with those patterns, the membership or consensus algorithm will fail. We avoid these results by making the specification conditional on certain failure patterns not occurring. If the patterns do occur, since they are possible in an asynchronous network, certain specified properties will not hold. The only property that will be violated in this case is the Membership Precision property. Since the condition on certain failure patterns is external to the actual algorithm, the algorithm does not know in what executions the failures will prevent a successful result. So the algorithm must always make a best effort to provide a correct solution and this conditional property does not allow the algorithm to take any shortcuts.

2.2 System and Layer Specifications

This specification divides the task of providing a complete messaging and group communication system into several components. Each component, or layer, relies on the semantics and services provided by lower layers and provides a well-specified set of semantics and services to layers above it.

Dividing the system into well-defined components both improves the analysis and makes a quality implementation easier. The separation of network-focused tasks such as providing a reliable point-to-point link protocol from more complex distributed algorithms, like membership, creates more specific tasks that each component must accomplish. This specification also maps quite naturally onto an actual software implementation with separate software modules providing the different services at each layer.

Four boundary levels exist: first, between the physical network and the network link layer; second, between the network link layer and the protocol layer; third, between the protocol layer and the session layer; and fourth, between the session layer and the application. Each of these boundary levels has a well defined specification of the semantics that layers above the boundary can rely on.

The boundary levels are not walls. Higher layer components can directly access any lower layer if needed. One case which requires layer skipping is probing for new servers. No current network link can exist to a server that has never been contacted before, so the probe must be sent using the lowest layer, unreliable UDP messages.

The boundary levels are marked B1, B2, B3, and B4 in Figure 2.1. Boundary B1 represents a minimum required network service level of unreliable point-to-point and multicast datagrams. Multicast datagrams are only assumed to work within local area networks and are not routed. This level is clearly provided by the standard IP Internet protocol of UDP.

Boundary B2 provides the abstraction of a reliable datagram link between two or more machines. The link guarantees that any messages sent through it will arrive at the other end, or if they cannot, the link will declare itself failed. The link also provides flow control

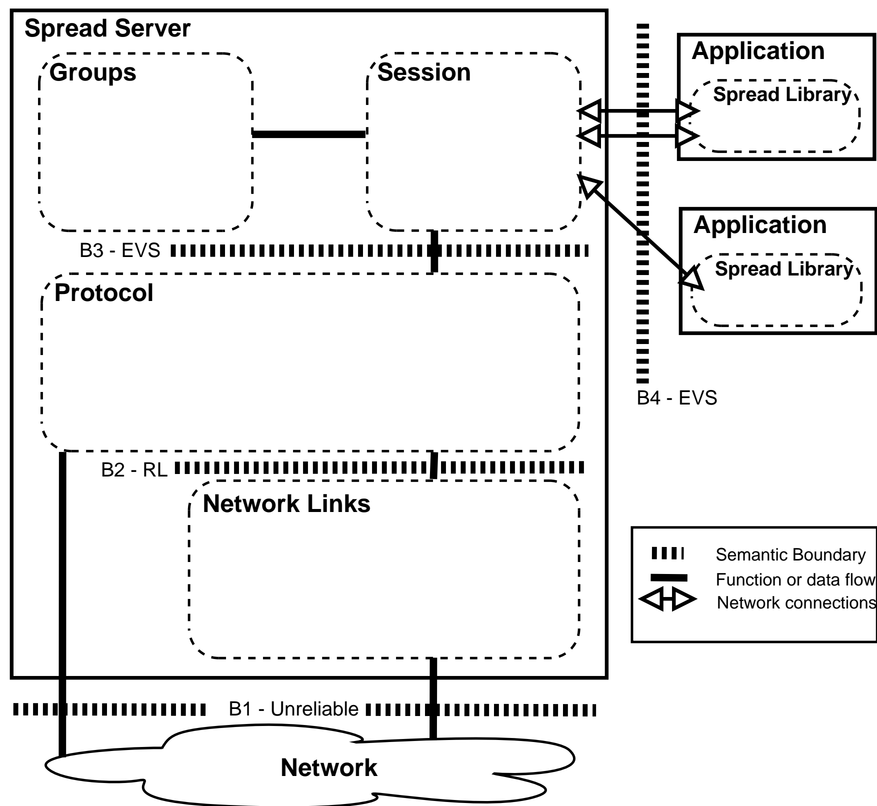

Figure 2.1: System Architecture

and congestion control sufficient to meet the requirements of Internet standards. For this work, this set of semantics has the name “Reliable Link” semantics, or RL semantics.

Boundary B3 provides full EVS semantics for the group of servers. These semantics are defined below in Section 2.2.5. This set of semantics has the name “EVS-S” or EVS-Servers.

Boundary B4 provides full EVS semantics for all of the clients and all of the groups they use. These semantics are defined below in Section 2.2.4. This set of semantics has the name “EVS-C” or EVS-Clients.

An important observation is that both the EVS-S and EVS-C provide essentially the same set of semantic guarantees. The difference between the two semantic layers is in what assumptions they make about the layers underneath them. For example, the EVS-S layer only assumes the existence of unreliable and reliable communication links between hosts, but does not assume anything regarding the reliability of messages sent to the entire system, or the coherency of the server state. The EVS-C layer on the other hand does assume that messages it sends have the ordering, safety, and membership guarantees of the EVS model, since those services are provided by the EVS-S layer below it.

Despite the fact that the EVS-C layer already assumes a complete EVS-S layer below it, the design and implementation of the EVS-C layer is definitely non-trivial. It is difficult for two reasons. First, the EVS-C layer has to map the EVS guarantees from a set of hosts (the EVS-S group) to a very large set of client members and groups. Second, the EVS-C layer provides a stronger service than the EVS-S layer in terms of performance guarantees. EVS-C guarantees that a group join or leave operation will not require any additional synchronization beyond one totally ordered message, while the EVS-S layer requires a full flush of messages and state synchronization when a join or leave operation occurs. To maintain this better performance guarantee, the EVS-C layer must maintain the consistent state of the groups while both network level changes occur (views delivered by the EVS-S layer) and group level changes occur (client joins and leaves).

2.2.1 Basic Model and Definitions

This specification draws on the IO atomota model developed by Lynch and Tuttle [LT87, Lyn96] and previous GCS specifications by Schultz[Sch01].

This section provides a set of definitions and properties.

The following symbols are used throughout the rest of this work.

N set of natural numbers.

C set of client processes.

S set of server processes.

M set of sent client messages.

P set of sent packets on the network.

G set of groups joined by clients.

Vid set of delivered view ids in strict partial order.

SVid set of delivered server view ids in strict partial order.

MT set of message types $\{R, F, A, S\}$.

In this thesis the variables a,b,i,j,k, and l are members of N, variables named c or d are members of C, variables named s or t are members of S, variables named g or h are members of 2^G, variables named m are members of M, variables named D and T are members of 2^P, and variables named id are members of Vid or SVid.

The external signature of the EVS-C specification consists of the actions:

- input $connect(c, s), c \in C, s \in S$
- input $disconnect(c, s), c \in C, s \in S$
- input $send(c, s, m), c \in C, s \in S, m \in M$
- input $join(c, s, g), c \in C, g \in 2^G$
- input $leave(c, s, g), c \in C, g \in 2^G$
- output $deliver(c, s, g, m), c \in C, s \in S, g \in 2^G, m \in M$

- output $view(c, s, g, id, D, T), c \in C, s \in S, g \in 2^G, id \in Vid, D \in 2^P, T \in 2^P$
- output $transsig(c, s, g, id), c \in C, s \in S, g \in 2^G, id \in Vid$

The complete signature also includes the actions:

- input $sdeliver(s, m)$
- input $sview(s, id, D, T)$
- input $stranssig(s, T)$
- output $ssend(s, m)$

These actions provide the interface between the EVS-S specification and the client level EVS specification.

Notice that this specification does not provide any actions referring to the failure or recovery of a process or a network link. This is because the EVS-C specification assumes an underlying EVS-S specification that will provide `view` events with information about network partitions and host crashes and recoveries. A client treats the detected crash of the server it is connected to as a disconnect. As when a client crashes, the data in buffers between the server and client is considered lost.

Client processes can also crash and recover, but for the signature of the GCS these events do not matter. The server treats the crash of a client process as a disconnect. The server handles a disconnect by initiating a leave of all the groups the client process had joined. The recovery of a client consists of initiating a connect event and then joining whatever groups it requires. Any additional state synchronization or recovery is outside of the GCS.

DEFINITION 2.1 (EVENT)
The occurrence of an action from the automaton's external signature.

DEFINITION 2.2 (TRACE)
A sequence of events.

Each event occurs at a single client process connected to a single server. Thus, a function cid mapping events to the client process for which they occurred and a function sid mapping events to the server process on which they occurred are well defined.

DEFINITION 2.3 (CID)
The cid of an event t_a is the client process at which the event t_a occured. Formally:

$$\begin{aligned} cid(t_a) = c \quad \textit{if} \quad & \exists s \,|\, t_a = connect(c,s) \vee t_a = disconnect(c,s) \\ \vee \quad & \exists m \exists g \,|\, t_a = send(c,s,g,m) \vee t_a = deliver(c,s,g,m) \\ \vee \quad & t_a = join(c,s,g) \vee t_a = leave(c,s,g) \\ \vee \quad & \exists id \exists D \exists T \,|\, t_a = view(c,s,g,id,D,T) \vee t_a = transsig(c,s,g,id) \end{aligned}$$

DEFINITION 2.4 (SID)
The sid of an event t_a is the server process at which the event t_a occurred. Formally:

$$\begin{aligned} sid(t_a) = s \quad \textit{if} \quad & \exists c \,|\, t_a = connect(c,s) \vee t_a = disconnect(c,s) \\ \vee \quad & \exists m \,|\, t_a = sdeliver(s,m) \vee t_a = ssend(s,m) \\ \vee \quad & \exists g \exists m \,|\, t_a = send(c,s,g,m) \vee t_a = deliver(c,s,g,m) \\ & t_a = join(c,s,g) \vee t_a = leave(c,s,g) \\ \vee \quad & \exists id \exists D \exists T \,|\, t_a = view(c,s,g,id,D,T) \vee t_a = transsig(c,s,g,id) \\ \vee \quad & t_a = sview(s,id,D,T) \vee t_a = stranssig(s,T) \end{aligned}$$

Each event occurs within the context of a view. The symbol $\perp$ represents the initial view for any process and the view at a client after disconnecting. The view in which event $t_k = view$ occurs is not t_k itself, but rather the previous view for that process. The function vid returns the view in which an event occurred at a client. Definition 2.5 provides the formal specification.

DEFINITION 2.5 (VID)
The vid of an event t_k at a process c is the view identifier delivered in a view event t_i at c which precedes t_k such that there are no view or connect events between t_i and t_k at c. If there is no such view event then the vid is the null view identifier, $\perp$. Formally:

$$\begin{aligned} vid(t_k,c,g) = \quad & id \quad \textit{if} \;\; \exists i \; \nexists j \exists s \exists D \exists T \,|\, i < j < k \wedge t_i = view(c,s,g,id,D,T) \wedge \\ & \qquad (t_j = connect(c,s) \vee \exists id' \exists D' \exists T' \,|\, t_j = view(c,s,g,id',D',T')) \\ & \perp \quad otherwise \end{aligned}$$

The causal order as defined by [Lam78] provides a strict partial order on events in a trace.

DEFINITION 2.6 (→)
The → relation defines an irreflective, anti-symmetric, transitive partial order. Formally:

$$\begin{aligned} t_i \rightarrow t_k \quad \equiv \quad & (cid(t_i) = cid(t_k) \wedge i < k) \\ & \vee \quad (t_i = send(c, s, g, m) \wedge t_k = deliver(d, s', g, m)) \\ & \vee \quad (\exists j \mid t_i \rightarrow t_j \textbf{ and } t_j \rightarrow t_k) \end{aligned}$$

All messages sent have an assigned, absolute order within the system. This order can be considered a one-to-one function from the set of messages to the set of natural numbers. This order is not necessarily the order in which the group communication system delivers the messages to clients, and clients do not have access to this order.

DEFINITION 2.7 (ORD)
The ord function is a one-to-one mapping from M to the set of natural numbers that is consistent with the causally precedes strict partial order of send events. Formally:

$$\begin{aligned} ord(m) = n \quad | \quad & (ord(m) = ord(m') \iff m = m') \wedge \\ & (t_a = send(c, s, g, m) \wedge t_i = send(d, s', g', m') \wedge t_a \rightarrow t_i \Rightarrow ord(m) < ord(m')) \end{aligned}$$

To assist in specifying some of the following properties, a few terms are defined concerning the order of events with regards to the client.

DEFINITION 2.8 (FIRSTEVENT)
The event t_j is the first event that occurs at a client c.

$$firstevent(t_j, c) \equiv \nexists i \mid i < j \wedge cid(t_i) = cid(t_j) = c$$

DEFINITION 2.9 (NEXTEVENT)
The event t_k is the next event after t_i at client c.

$$nextevent(t_k, t_i, c) \equiv i < k \wedge cid(t_i) = cid(t_k) = c \wedge \ \nexists j \mid cid(t_j) = c \wedge i < j < g$$

DEFINITION 2.10 (PREVEVENT)
The event t_i is the previous event prior to t_k at client c.

$$prevevent(t_i, t_k, c) \equiv i < k \wedge cid(t_i) = cid(t_k) = c \wedge \nexists j \,|\, cid(t_j) = c \wedge i < j < g$$

DEFINITION 2.11 (FIRSTGROUPEVENT)
The event t_j is the first event referring to group g that occurs at client c.

$$firstgroupevent(t_j, c, g) \equiv \nexists i \,|\, i < j \wedge cid(t_i) = cid(t_j) = c \wedge (t_i = join(c, s, g) \vee t_i = leave(c, s, g))$$

DEFINITION 2.12 (LASTGROUPEVENT)
The event t_j is the last event referring to group g that occurs at client c.

$$lastgroupevent(t_j, c, g) \equiv \nexists k \,|\, j < k \wedge cid(t_j) = cid(t_k) = c \wedge (t_k = join(c, s, g) \vee t_i = leave(c, s, g))$$

We also need a similar set of terms for the order of events with regards to the servers.

DEFINITION 2.13 (SFIRSTEVENT)
The event t_j is the first event that occurs at a server s.

$$sfirstevent(t_j, s) \equiv \nexists i \,|\, i < j \wedge sid(t_i) = sid(t_j) = s$$

DEFINITION 2.14 (SNEXTEVENT)
The event t_k is the next event after t_i at server s.

$$snextevent(t_k, t_i, s) \equiv i < k \wedge sid(t_i) = sid(t_k) = s \wedge \nexists j \,|\, sid(t_j) = s \wedge i < j < g$$

DEFINITION 2.15 (SPREVEVENT)
The event t_i is the previous event prior to t_k at server s.

$$sprevevent(t_i, t_k, s) \equiv i < k \wedge sid(t_i) = sid(t_k) = s \wedge \nexists j \,|\, sid(t_j) = s \wedge i < j < g$$

2.2.2 System Assumptions

To provide a more precise representation of the environment the model makes several assumptions about the behavior of the surrounding environment and the system.

ASSUMPTION 2.1 (SERVER EXECUTION STRUCTURE)
The first event at a server process is a recover event. If a recover event occurs at a server process, then it is either the first event at that process or the previous event was a crash event. The next event that occurs at a server process after a crash event is a recover event.

$$sfirstevent(t_j, s) \Rightarrow t_j = recover(s)$$
$$t_j = recover(s) \Rightarrow sfirstevent(t_j, s) \vee (sprevevent(t_i, t_j, s) \wedge t_i = crash(s))$$
$$t_i = crash(s) \wedge snextevent(t_j, t_i s) \Rightarrow t_j = recover(s)$$

ASSUMPTION 2.2 (CLIENT EXECUTION STRUCTURE)
The first event at a client process is a connect event. The last input event at a client process is a disconnect event. The first event at a client that refers to a group is a join event. The last input event at a client process for each group is a leave event, the last output event is a view.

$$firstevent(t_j, c) \Rightarrow t_j = connect(c, s)$$
$$\nexists j \,|\, nextevent(t_j, t_i, c) \Rightarrow t_i = disconnect(c, s)$$
$$firstgroupevent(t_j, c, g) \Rightarrow t_j = join(c, s, g)$$
$$lastgroupevent(t_j, c, g) \Rightarrow t_j = leave(c, s, g) \vee t_j = view(c, s, g, id, D, T)$$

Each message sent by a client to the same group is assumed to be unique. The client can easily guarantee this property by appending a unique sequence number on every message.

ASSUMPTION 2.3 (CLIENT MESSAGE UNIQUENESS)
Any two send events of the same message body to the same group are actually the same event.

$$t_i = send(c, s, g, m) \wedge t_j = send(c', s', g, m) \Rightarrow i = j$$

2.2.3 Extended Virtual Synchrony

These properties define the general Extended Virtual Synchrony properties that both the EVS-C and EVS-S specifications have in common. Sections 2.2.4 and 2.2.5 discuss the properties that are unique to the EVS-C specification and the EVS-S specification, respectively. The properties are formally defined in terms of the EVS-C primitives specified above, as that is the specification provided to the applications using the system. The differences in the EVS-S specification are either obvious (as in replacing $vid(t_i, c, g)$ with

$svid(t_i, s)$) or are discussed in Section 2.2.5.

2.2.3.1 Membership Safety Properties

These properties define the safety guarantees of the Extended Virtual Synchrony Model.

PROPERTY 2.1 (INITIAL VIEW EVENT)
Every deliver and transsig event at a process occurs within some view.[1]

$$t_i = deliver(c, s, g, m) \lor t_i = transsig(c, s, g, id) \Rightarrow vid(t_i, c, g) \neq \bot$$

PROPERTY 2.2 (SELF INCLUSION)
If a process p installs a view, then p is a member of the membership set.

$$t_i = view(c, s, g, id, D, T) \Rightarrow c \in D$$

The Self Inclusion property provides a constraint on the set of members by giving a base membership that always exists as long as the process has not crashed itself. The usefulness of a membership view which excluded the process is also very limited.

PROPERTY 2.3 (LOCAL MONOTONICITY)
If a process p installs a view with identifier id' after installing a view with identifier id, then id' is greater then id.

$$t_i = view(c, s, g, id, D, T) \land t_j = view(c, s, id', D', T') \land i < j \Rightarrow id < id'$$

PROPERTY 2.4 (TRANSITIONAL SET)

1. *The transitional set for the first view installed at a process following either a connect or recover event is the empty set.*
2. *If a process p installs a view in a previous view, then the transitional set for the new view at p is a subset of the intersection between the two views' membership sets.*
3. *If processes p and q install the same view, then q is included in p's transitional set for that view if and only if p's previous view was identical to q's previous view.*

[1] For closed-groups, every send event also occurs within some view, but in open groups a send event can occur at a process who is not in the group

4. *If processes p and q install the same view in the same previous view, then they have the same transitional sets in their new views.*

2.2.3.2 Message Properties

PROPERTY 2.5 (NO DUPLICATION)
A process never delivers a message more than once.

$$t_i = deliver(c, s, g, m) \wedge t_j = deliver(c, s, g, m) \Rightarrow i = j$$

PROPERTY 2.6 (DELIVER INTEGRITY)
A deliver event in a view is the result of a preceding send event[2].

$$t_j = deliver(c, s, g, m) \Rightarrow \exists i \exists c' \exists s' \mid i < j \wedge t_i = send(c', s', g, m)$$

Several later properties use the idea of processes "moving together" from one view to another. A formal definition of "moving together" follows:

DEFINITION 2.16 (VSYNCHRONOUS_IN)
If processes c and d both install the same view in the same previous view and d is in c's transitional set, then they were virtually synchronous in that previous view.

$$\begin{aligned} vsynchronous_in(c, d, g, id) \equiv\ & \exists i \exists id' \exists s \exists D \exists T \exists j \exists s' \mid t_i = view(c, s, g, id', D, T) \\ & \wedge t_j = view(d, s', g, id', D, T) \wedge vid(t_i, c, g) = vid(t_j, d, g) \\ & \wedge d \in T \end{aligned}$$

PROPERTY 2.7 (VIRTUAL SYNCHRONY)
If processes p and q are virtually synchronous in a view (as defined above), then any message delivered by p in that view is also delivered by q.

$$\begin{aligned} & vsynchronous_in(c, d, g, id) \wedge t_i = deliver(c, s, g, m) \wedge vid(t_i, c, g) = id \\ \Rightarrow\ & \exists j \exists s' \mid t_j = deliver(d, s', g, m) \end{aligned}$$

[2]In a closed-group model the deliver must be the result of a send by a member in that view.

PROPERTY 2.8 (TRANSITIONAL SIGNALS)

1. *At most one transsig event occurs at a process during a view.*

$$\begin{aligned} & t_i = transsig(c,s,g,id') \wedge vid(t_i,c,g) = id \\ \Rightarrow \quad & \nexists j \mid j \neq i \wedge t_j = transsig(c,s,g,id') \wedge vid(t_j,c,g) = id \end{aligned}$$

2. *If two processes p and q are virtually synchronous in a view v and p has a transsig event occur in v, then q also has a transsig event occur in v and they both deliver the same sets of agreed messages before and after their transsig events in v.*

$$\begin{aligned} & vsynchronous_in(c,d,g,id) \wedge t_b = transsig(c,s,g,id') \wedge vid(t_b,c,g) = id \Rightarrow \\ & \exists j \mid t_j = transsig(d,s',g,id') \wedge vid(t_j,d,g) = id \\ & \wedge (\exists a \exists m \mid a < b \wedge t_a = deliver(c,s,g,m) \wedge vid(t_a,c,g) = id \wedge agreed(m) \\ & \iff \exists i \exists m \mid i < j \wedge t_i = deliver(d,s',g,m) \wedge vid(t_i,d,g) = id \wedge agreed(m)) \\ & \wedge (\exists l \exists m' \mid b < l \wedge t_l = deliver(c,s,g,m') \wedge vid(t_l,c,g) = id \wedge agreed(m') \\ & \iff \exists k \exists m' \mid j < k \wedge t_k = deliver(d,s',g,m') \wedge vid(t_k,d,g) = id \wedge agreed(m)) \end{aligned}$$

A number of GCS provide a property called Sending View Delivery which guarantees that messages will be delivered in the same view as they were sent in. However, providing this guarantee requires that a flush of all outstanding messages occur at every membership change which is very costly for dynamic systems. The EVS specification provided here does not require Sending View Delivery. As a result, the general EVS properties from Section 2.2.3 are not sufficient to require reasonable behavior as to which view a message is delivered in. One could imagine that all processes deliver a message in the same view, but it is a view that is much later then expected or earlier then one would think possible. The following properties constrain in which views a message is delivered and are provided by all known EVS implementations.

PROPERTY 2.9 (SANE VIEW DELIVERY)

1. *A message m with $m.type \in \{A, S\}$ is not delivered in a view earlier than the one in which it was sent.*

$$\begin{aligned} & t_i = send(c,s,g,m) \wedge vid(t_i,c,g) = id \wedge m.type \in \{A,S\} \\ & \quad \wedge t_j = deliver(d,s',g,m) \wedge vid(t_j,d,g) = id' \Rightarrow id \leq id' \end{aligned}$$

2. *If a process p sends a message m, crashes and later recovers in a view v and a process*

q delivers m, then m is delivered in a view before v.

$$t_i = send(c, s, g, m) \wedge t_k = view(c, s, g, id, D, T) \wedge vid(t_k, c, g) = \perp$$
$$\wedge\, i < k \wedge t_j = deliver(d, s', g, m) \Rightarrow vid(t_j, d, g) < id$$

The different types of messages have well-defined properties that they maintain. The message types form a hierarchy where each message type contains all of the properties of those types lower in the hierarchy, as well as some additional properties. All of the message types are consistent with lower types. Each message sent may have its own type, so if a client sends message m_1 and then message m_2, m_1 may be of type AGREED and m_2 may be of type FIFO, in which case m_2 may not be delivered before m_1 because AGREED is also FIFO. The use of a hierarchy makes Spread use *weak incorporated* [WS95] delivery semantics when delivering messages of two different types.

PROPERTY 2.10 (RELIABLE MESSAGES)
The Self-Delivery, Sane View Delivery and Virtual Synchrony properties define the safety properties of Reliable messages.

$$reliable(m) \equiv m.type \in \{R, F, A, S\}$$

PROPERTY 2.11 (FIFO MESSAGES)

1. *FIFO messages are Reliable messages.*

$$fifo(m) \equiv m.type \in \{F, A, S\}$$

2. *If a process sends a FIFO message after sending a previous message, then all processes which deliver both messages deliver them in the order in which they were sent.*

$$t_a = send(c, s, g, m) \wedge t_b = send(c, s, g', m') \wedge a < b \wedge fifo(m')$$
$$\wedge\, t_i = deliver(d, s', g, m) \wedge t_j = deliver(d, s', g', m')$$
$$\Rightarrow \quad i < j$$

PROPERTY 2.12 (CAUSAL MESSAGES)

1. *Causal messages are FIFO messages.*

$$causal(m) \equiv m.type \in \{A, S\}$$

2. *If a process sends a causal message m' such that the send of another message m causally precedes the send of m', then any process that delivers both messages delivers m before m'.*

$$\begin{aligned} & t_a = send(c,s,g,m) \wedge t_b = send(c',s,g',m') \wedge t_a \rightarrow t_b \wedge causal(m') \\ & \quad \wedge t_i = deliver(d,s',g,m) \wedge t_j = deliver(d,s',g',m') \\ \Rightarrow \quad & i < j \end{aligned}$$

PROPERTY 2.13 (AGREED MESSAGES)

1. *Agreed Messages are causal messages.*

$$agreed(m) \equiv m.type \in \{A, S\}$$

2. *If a process p delivers an agreed message m, then after that event it will never deliver a message that has a lower ord value.*

$$t_a = deliver(c,s,g,m) \wedge t_b = deliver(c,s,g,m') \wedge agreed(m') \wedge ord(m) < ord(m') \Rightarrow a < b$$

3. *If a process p delivers an agreed message m' before a transsig event in its current view, then p delivers every message with a lower ord value than m' delivered in that view by any process.*

$$\begin{aligned} & t_k = deliver(c,s,g,m') \wedge agreed(m') \\ & \quad \wedge (\nexists j \,|\, j < k \wedge t_j = transsig(c,s,g,id) \wedge vid(t_j,c,g) = vid(t_k,c,g)) \\ \Rightarrow \quad & \forall a \forall d \forall m \,|\, t_a = deliver(d,s',g,m) \wedge vid(t_a,d,g) = vid(t_k,c,g) \wedge ord(m) < ord(m'); \\ & \quad \exists i \,|\, t_i = deliver(c,s,g,m) \end{aligned}$$

4. *If a process p delivers an agreed message m' after a transsig event in its current view, then p delivers every message with a lower ord value than m' sent by any processes in p's next transitional set that were delivered in the same view as m'.*

$$\begin{aligned} & t_i = transsig(c,s,g,id) \wedge t_k = deliver(c,s,g,m') \wedge t_l = view(c,s,g,id',D',T') \\ & \quad \wedge i < k < l \wedge agreed(m') \wedge vid(t_i,c,g) = vid(t_k,c,g) = vid(t_l,c,g) = id \\ \Rightarrow \quad & \forall a \forall d \in T' \forall m \forall b \forall c' \,|\, t_a = send(d,s,g,m) \wedge t_b = deliver(c',s,g,m) \\ & \quad \wedge vid(t_b,c',g) = id \wedge ord(m) < ord(m'); \\ & \exists j \,|\, t_j = deliver(c,s,g,m) \end{aligned}$$

PROPERTY 2.14 (SAFE MESSAGES)

1. *Safe messages are agreed messages.*

$$safe(m) \equiv m.type \in \{S\}$$

2. *If a process p delivers a safe message m before a transsig event in its current view id, then every member of that view delivers m, unless that member crashes in id.*

$$\begin{aligned} & t_i = view(c,s,g,id,D,T) \wedge t_k = deliver(c,s,g,m) \wedge safe(m) \wedge vid(t_k,c,g) = id \\ & \quad \wedge \nexists j \,|\, i < j < k \wedge t_j = transsig(c,s,g,id) \\ \Rightarrow \quad & \forall d \in D; \exists a \exists D' \exists T' \exists b \,|\, t_a = view(d,s',g,id,D',T') \\ & \quad \wedge (t_b = deliver(d,s,g,m) \vee (t_b = disconnect(d,s') \wedge vid(t_b,d,g) = id)) \end{aligned}$$

3. *If a process p delivers a safe message m after a transsig event in its current view id, then every member of p's transitional set from p's next view delivers m, unless a member crashes in id.*

$$\begin{aligned} & t_i = view(c,s,g,id,D,T) \wedge t_j = transsig(c,s,g,id) \wedge t_k = deliver(c,s,g,m) \\ & \quad \wedge t_l = view(c,s,g,id'',D'',T'') \wedge safe(m) \wedge j < k \\ & \quad \wedge vid(t_j,c,g) = vid(t_k,c,g) = vid(t_l,c,g) = id \\ \Rightarrow \quad & \forall d \in T''; \exists a \exists D' \exists T' \exists b \,|\, t_a = view(d,s',g,id,D',T') \\ & \quad \wedge (t_b = deliver(d,s,g,m) \vee (t_b = disconnect(d,s) \wedge vid(t_b,d,g) = id)) \end{aligned}$$

A SAFE message is not an atomic message, in that the delivery of a safe message does not guarantee all or none of the applications will receive the message. It is a guarantee that if no processes crash, but the network partitions in arbitrary ways, then all of the daemons and applications will receive the message.

2.2.3.3 Liveness Properties

The liveness properties of Extended Virtual Synchrony are determined by the liveness properties of the failure detector as well as the algorithm.

The properties that define the liveness guarantees are below.

PROPERTY 2.15 (SELF DELIVERY)

If a process p sends a message m to group g, and p is a member of g and does not leave

group g, then m is delivered to p unless p crashes[3].

$$
\begin{aligned}
& t_i = send(c,s,g,m) \wedge \exists l \,|\, l < i \wedge t_l = view(c,s,g,id,D,T) \wedge c \in D \\
& \quad \wedge \nexists j \,|\, i < j \wedge (t_j = disconnect(c,s) \vee t_j = leave(c,s,g)) \\
\Rightarrow \quad & \exists k \,|\, t_k = deliver(c,s,g,m)
\end{aligned}
$$

PROPERTY 2.16 (MEMBERSHIP PRECISION)
If a stable component S exists, then a view v exists with member set equal to S, and every process p in S installs v and does not install any later views.

PROPERTY 2.17 (TERMINATION OF DELIVERY)
If a process p sends a message m in view v, then for each member q of view v, either q delivers m, or p installs a view v' such that $vid(v') = v$.

They are conditional on the networks behavior and thus they are only required to hold in runs when a stable component exists.

2.2.4 EVS-Client Semantics

The EVS-C specification includes all the standard EVS semantics as specified above, with one exception. The exception is that Same View Delivery does not hold for RELIABLE or FIFO messages.

Often reliable and FIFO ordered messages are not even included in group communication specifications because the traditional users of such systems, such as consistent replication, required causal or total ordered multicast. Also, the benefit of reliable and FIFO messages are that they may be delivered with less latency then messages with stronger ordering. However, to preserve Same View delivery for them and to allow the EVS-C implementation to optimize group join and leave events such that they do not require any synchronization, would require that they not be delivered until they were totally ordered, so the entire latency benefit would be lost. By weakening the specification for reliable and

[3]If the server p is connected to crashes, then p will detect that it must treat the crash the same as if p had crashed itself, such as an explicit disconnect event, with regards to data and state recovery.

FIFO messages, the specification allows an implementation that optimizes both group join and leave events as well as provides low-latency reliable and FIFO messages.

2.2.5 EVS-Server Semantics

For the specification of the EVS-S semantics a different set of actions are possible:

- input $ssend(s, m), s \in S, m \in M$
- input $nreceive(s, m), s \in S, m \in M$
- input $ureceive(s, m), s \in S, m \in M$
- input $crash(s), s \in S$
- input $linkfailed()$
- internal $recover(s), s \in S$
- output $mcast(\{s_1, s_2, \ldots\}, m), s_i \in S, m \in M$
- output $nmcast(m), m \in M$
- output $sdeliver(s, m), s \in S, m \in M$
- output $sview(s, id, D, T), s \in S, id \in SVid, D \in 2^P, T \in 2^P$
- output $stranssig(s, T), s \in S, T \in 2^P$

Just like the EVS-C specification, the EVS-S specification provides that every event occurs in the context of some view, which may be the null view $\perp$. The views seen at the EVS-S layer are a subset of the views deliverd by the EVS-C layer as the view changes caused by clients joining or leaving groups do not cause view changes at the EVS-S layer. Because of this a new svid function must be defined, corresponding to the vid function 2.5 defined earlier.

DEFINITION 2.17 (SVID)
The svid of an event t_c at a server s is the view identifier delivered in a view event t_a at s which precedes t_c such that there are no view or recover events between t_a and t_c at s. If

there is no such view event then the svid is the null view identifier, $\perp$. Formally:

$$svid(t_c, s) = \begin{cases} id & \text{if } \exists a \ \nexists b \exists D \exists T \mid a < b < c \land t_a = sview(s, id, D, T) \land \\ & \quad (t_b = recover(s) \lor \exists id' \exists D' \exists T' \mid t_b = sview(s, id', D', T')) \\ \perp & otherwise \end{cases}$$

The EVS-Server specification actually provides a stronger semantic then general EVS, namely Same View Delivery. This is specified below:

PROPERTY 2.18 (SAME VIEW DELIVERY)
If servers s and t both deliver a message m then they both deliver m in the same view.

$$t_i = sdeliver(s, m) \land svid(t_i, s) = id \land t_j = sdeliver(s', m) \land svid(t_j, s') = id' \Rightarrow id = id'$$

The EVS-Server specification is also slightly stronger with regards to the Sane View Delivery property as well. Specifically the guarantees about messages not being delivered in earlier views then they were sent is guaranteed for all messages types, not only Agreed and Safe messages.

PROPERTY 2.19 (SANE VIEW DELIVERY)

1. *A message m is not delivered in a view earlier than the one in which it was sent.*

$$t_i = send(c, s, g, m) \land vid(t_i, c, g) = id \\ \land t_j = deliver(d, s', g, m) \land vid(t_j, d, g) = id' \Rightarrow id \leq id'$$

2. *If a process p sends a message m, crashes and later recovers in a view v and a process q delivers m, then m is delivered in a view before v.*

$$t_i = send(c, s, g, m) \land t_k = view(c, s, g, id, D, T) \land vid(t_k, c, g) = \perp \\ \land i < k \land t_j = deliver(d, s', g, m) \Rightarrow vid(t_j, d, g) < id$$

PROPERTY 2.20 (MEMBERSHIP AGREEMENT)
If a process p installs a view with identifier id and a process q installs a view with the same identifier, then the membership sets of the views are identical.

$$t_i = view(c, s, g, id, D, T) \land t_j = view(c', s', g, id, D', T') \Rightarrow D = D'$$

Membership Agreement provides a unique mapping from view ids to membership sets. This allows clients to compare view id histories and construct a consistent picture of the membership of views in which the clients were both present. This is only strictly provided by the EVS-S algorithm as in the EVS-C algorithm member sets that partition away from each other can deliver some view id's that are identical but do not contain identical sets of members. These occur due to single member joins or leaves while a heavy-weight membership change is in process. An application can work around this by including in the unique view id the membership set delived with that view. Then if two views ids are identical and the membership sets are identical, then the two processes historically delivered the same view.

2.2.6 Network Specification

The network specification is a very simple model that represents a reliable overlay network. The possible actions are:

- input $ninit(\{s_1, s_2, \ldots\}), s_i \in S$
- input $nmcast(m), m \in M$
- output $nreceive(s, m), s \in N, m \in M$
- output $linkfailed()$

The purpose of this model is to capture the dissemination and routing properties of an overlay network upon which messages can be multicast. The properties of this model only hold during an window of time, which may be infinite, during which the network maintains the same connectivity.

PROPERTY 2.21 (NETWORK SERVER SET)
N equals the set of servers in the **`ninit`** *event.*

$$N = \{s_1, s_2, \ldots\}$$

PROPERTY 2.22 (RELIABLE NETWORK MULTICAST)
Every **`nmcast`** *event will have a corresponding* **`nreceive`** *event at all servers in N in the network unless a* **`linkfailed`** *event occurs.*

$$t_i = nmcast(m) \Rightarrow (\forall s \in N t_j = nreceive(s, m) \wedge i < j) \vee (t_k = linkfailed() \wedge i < k)$$

PROPERTY 2.23 (NETWORK END)
No events occur between a **`linkfailed`** *event and the next* **`ninit`** *event.*

$$t_i = linkfailed() t_k = ninit(s_i, \ldots) \Rightarrow \nexists j \,|\, i < j < k \wedge (t_j = nmcast(m) \vee t_j = nreceive(s, m))$$

Although no liveness or performance properties for the Network specification are provided, in practice the performance of this model is critical to the performance of the entire system. The actual implementation and algorithms provide a "best effort" solution to this problem.

2.2.7 Reliable Link Semantics

The Network model uses a number of Reliable Links, both point-to-point and multicast, to present a wide-area reliable multicast service. These links are specified below and have the following actions:

- input $rmcast(\{p_1, p_2, \ldots\}, m), p_i \in P, m \in M$
- output $rreceive(p, m), p \in P, m \in M$
- output $linkfailed()$

All links are uniquely identified by the endpoints they connect, either two IP addresses for point-to-point links, or a multicast IP address for multicast links. All actions on links are parameterized by the link id of the link, even though that parameter is not shown in the action specification.

For point-to-point links the rmcast parameter p_i must be a singleton set of one process.

2.2.8 Unreliable Multicast Semantics

The unreliable multicast channel provided by UDP datagrams is modeled by an IO automaton with the following events in its signature.

- input $mcast(\{p_1, p_2, \ldots\}, m), p_i \in P, m \in M$
- output $ureceive(p, m), p \in P, m \in M$

2.3 Discussion

2.3.1 Minimal vs. "Best Effort" Implementations

In any system, even one with strong specifications, a system implementation that provides exactly the required specifications, while correct, may not be useful. This occurs because it is very difficult to specify liveness and performance constraints that are sufficiently strong, while still being implementable in arbitrary, or even common, networks. Also, just as with most software, significant differences exists between different implementations of the same systems because of differences in quality of software design and engineering or in the myriad of detailed design and implementation choices which are not specified.

Systems which will be useful must not only provide the required specifications, they must also do so in a high-quality implementation that provides not just the minimal required service, but rather a "best effort" service. This "best effort" will, although only guaranteed to provide what was specified, actually provide better service in most scenarios and practical environments (usually in performance and liveness properties).

2.3.2 Service Interactions

Traditionally, most group communication systems have been used to send Agreed (total) ordered messages. Spread attempts to provide not only the strongly ordered Agreed message service, but also less-ordered FIFO and Reliable message service. FIFO and Reliable

have the advantage in that they provide lower latency, from send to receive, then Agreed order messages. These messages can be delivered when they meet a specific set of constraints. For Reliable, the only constraint is the complete receipt of the message. For FIFO, the constraints are that the entire message must have arrived and that all messages prior to this one, from the same client must have already been delivered.

Since each message sent may request a different ordering or delivery guarantee the way differently ordered messages interact acquires importance.

2.3.3 Obsolete Views

When designing membership specifications for wide-area and large distributed systems, the issues of whether or not to deliver obsolete views, and second, if they are delivered, how often to deliver them, has significant interest. An obsolete view is a view delivered by the system that is known by the membership algorithm to not represent the most current information about the underlying hosts and network.

The question is when the network is unstable, should the group communication system tell the client about the interim views during the unstable period or should it block or wait until things stabilize and then tell the client only the final result.

If one assumes that the clients act as blind responders to view events, then obsolete views can cause significant performance problems. Clients will have to run a synchronization protocol in response to the view event, which will turn out to be unnecessary or redundant because of a rapidly following new view event. In essence, the view that the application received was already obsolete even when it received it because the lower level group communication system already knew of additional group changes.

However, in larger networks the possibility of longer stabilization times and more frequent view changes make waiting until the network stabilizes a strategy with high live-lock potential, as the network may take a long time to stabilize sufficiently. Many applications which do not require strict replica consistency can still make use of the group messaging

system during these periods of instability if they are given some indication of what is going on. They can use the information in obsolete views to adapt their own behavior to the network changes. Many applications would also prefer a system that 'never' blocked, but instead occasionally gave inaccurate information rather than one that was always accurate but could block for noticeable periods of time.

Chapter 3

Spread System Architecture

The Spread messaging toolkit is a concrete realization of the hierarchical group communication architecture discussed in this thesis. The goal of the architecture is to provide scalable and strong wide-area group communication services, as well as light-weight messaging services. The architecture is designed to be flexible, allowing network protocols, service guarantees, and groups to be selected to best support each application.

The overall design philosophy was, to borrow from the common saying about the Perl programming language, “Easy things should be easy and efficient, hard things should be correct, useful, and as efficient as possible.”

When discussing distributed systems efficiency usually refers to efficiency in the number of messages or efficiency in the amount of latency added by the protocol. In the case of wide-area group communication, both are important with latency having a higher priority. Latency is the critical resource that must be optimized because it provides the fundamental lower-bound on the performance of an application. Faster networks and better hardware can improve the bandwidth of wide-area networks to an almost unbounded degree, but the latency of the network is already approaching the basic speed-of-light limits and is not expected to improve in the future. Therefore, in the Spread architecture when a tradeoff between bandwidth and latency is required, the choice made is to decrease latency even at

the cost of some increase in the required bandwidth.

Spread is a client-server system. The core component in Spread is the Spread daemon or server. It is called a daemon because in the Unix lexicon a daemon is a long-running background process that provides some system service. The daemon acts as a local server to a possibly large number of client connections and coordinates the messages sent by those clients, and the actions requested by those clients, such as joins and leaves of a group. In addition to supporting directly connected clients, the daemon also provides message routing and reliability for messages sent by other daemons. Finally, the daemon detects network and host failures and reconfigures the system to work around those failures.

A client of the Spread system is an application program that communicates with the daemon over a local IPC mechanism or a remote TCP connection. A standard Spread library implements the client-server protocol and can be linked into an application.

The Spread system as presented in this and the following chapters provides a complete implementation of the specification given in Chapter 2. As well as providing a provably correct functional implementation, Spread also provides a high-performance implementation with a variety of experimental validation and evaluation.

Spread is implemented in a general modular framework that permits a number of different network protocols to be added to the system with minimal changes. This modularity extends to the creation of new message types for specific purposes, and different configurations for different network and application environments.

3.1 History

Spread goes back to research done by my advisor Professor Yair Amir and was originally an attempt to make a high quality, production ready group communication system based on academic and individual research. It provided primitive support for wide area networks and a highly-optimized protocol for local area networks.

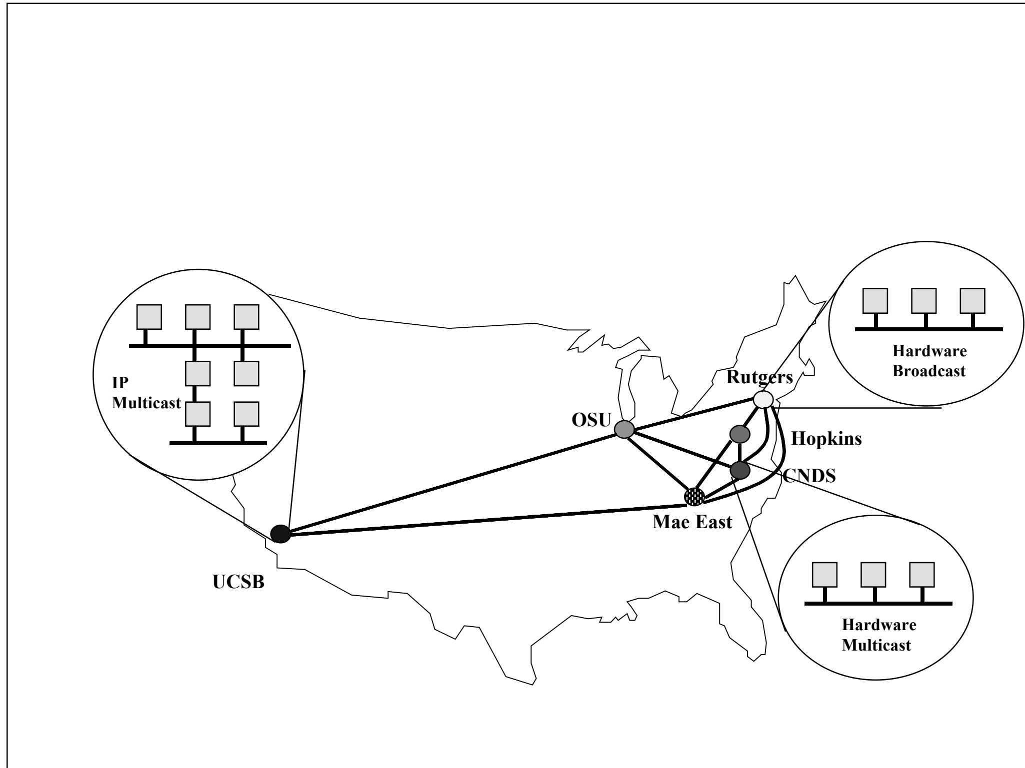

Figure 3.1: Sites and Links

Over the course of my research Spread has evolved into a complete system with several independent implementations of certain key components and has been continually improved in terms of performance and scalability.

This work presents the first complete architecture for wide area group communication and its implementation in the Spread system.

3.2 Architecture

The overall architecture of the Spread system is a three-level hierarchy of client applications, communication servers, and network sites.

The Spread system uses generally long-running servers to establish the basic message dissemination network and provide basic membership and ordering services, while user applications link with a small client library, can reside anywhere in the network, and will con-

nect to a server to gain access to the group communication services. There is a small cost to using a server-client architecture. The cost is extra context-switches and inter-process communication. On modern systems, however, this cost is minimal in comparison with wide-area latencies. The benefit of using a client-server architecture is vastly increased scalability both in terms of clients connected and the efficiency of the messaging and distributed membership protocols.

A “site” in Spread consists of a collection of servers which can all communicate over a broadcast or multicast domain. This is usually limited to a local area network. We will use the term “site” to refer to this collection of locally connected servers as a whole. Each site selects one server, based on the current membership of the site, that acts as a gateway, connecting all the members of the site to other sites. An example of an actual Spread configuration is shown in Figure 3.1 where sites with multiple local hosts exist at UCSB, Rutgers and Johns Hopkins CNDS lab, while sites with single hosts exist at OSU, Mae East, and Johns Hopkins Computer Science Department.

The Spread system is implemented as a server process and a client library. The overall software architecture is shown in Figure 3.2. The applications link directly with the client library. The client library is is fairly small. Its primary purpose is to provide a good API and to translate client requests and messages and server responses between the API provided to the client application and the message interface expected by the server.

The server implements most of the algorithms and protocols of the system. It contains three identifiable layers: a session layer, a server protocol layer, and a network layer.

These three layers define the structure of both the architecture and the software implementation. The first, and lowest level task, is to provide a reliable wide-area network multicast service that provides dynamic routing depending on the set of available servers. This task is discussed below in Section 3.3 and Chapter 4.

The second task is to provide the various supported message ordering services and a membership service across the set of servers who are active in the network. This task is

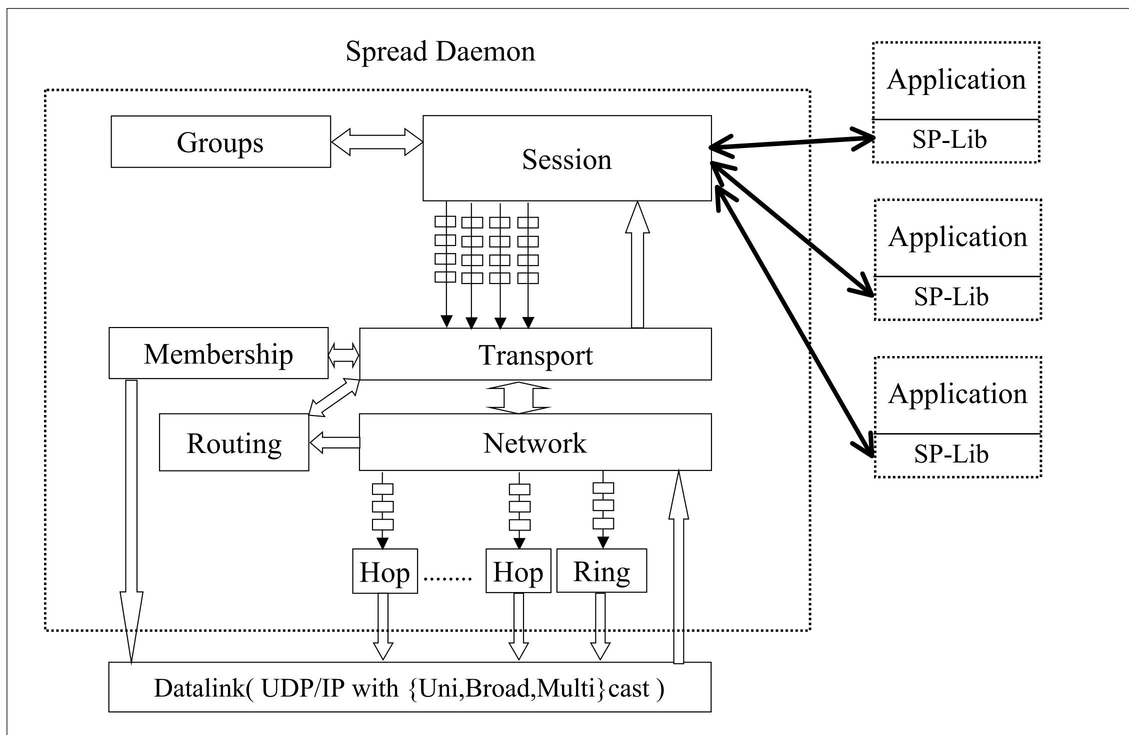

Figure 3.2: Spread Software Architecture

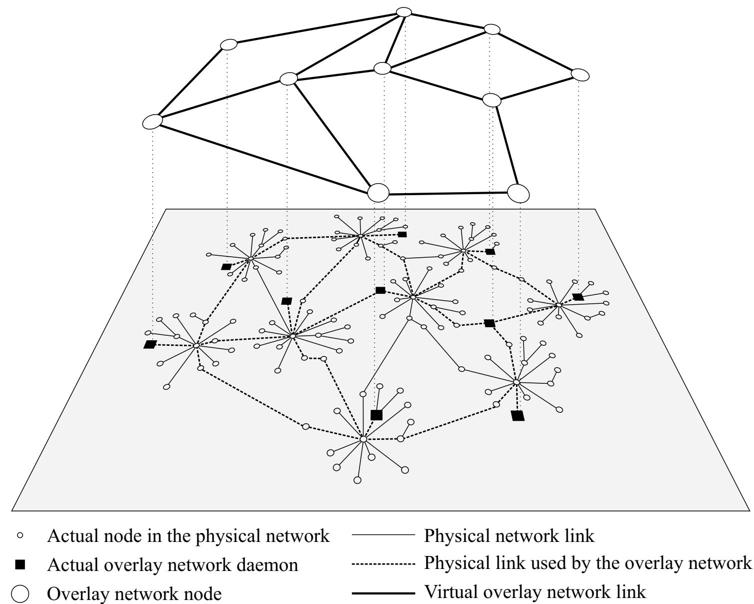

Figure 3.3: Overlay Network Architecture

discussed in Chapter 5 and the first half of Chapter 6.

The third task, at the session layer, is to provide the full set of group services to applications who use the group communication system. This allows many different users to share one server and permits them to create, join and leave groups and send messages to those groups efficiently. This task is discussed below in Section 3.4 and in the second half of Chapter 6.

3.3 Overlay Network, Routing, and Reliability

An "overlay network" is defined as a virtual network constructed so that each link connects two edge nodes in an underlying physical network, such as the Internet. Each virtual link in the overlay network can translate into several hops in the underlying network and the cost attached to a virtual link is some aggregated cost of the underlying network links over which it travels. A sample overlay network is shown in Figure 3.3. In the figure, the Spread daemons are located at hosts located in leaf networks.

Spread constructs an overlay network between all the sites that currently have active servers. This network is constructed based on information contained in the static configuration file given to Spread, the current server membership list, and any available network cost information. These sources produce a network that dynamically changes as servers start or crash, and as partitions or merges occur. The configuration file provides information about all potential machines in the Spread system, but does not constrain which members are currently running.

The overlay network is used to calculate source-based optimal routing paths from each source to all other Spread servers. Source-based routing over the shortest path[NB97] produces better routes than either a single shared multicast tree or incrementally constructed source routes. The cost of per-source shortest path routing is the computational calculation of routes and the necessity of maintaining complete knowledge of current members and the costs between them. In a group communication system, such as Spread, this is an acceptable cost because the number of nodes in the routing graph is limited, the system must maintain accurate, current membership anyway to provide the guaranteed semantics. Link costs are not too expensive to maintain for a limited number of nodes. After a membership change a new overlay network is constructed and the routing trees are recalculated based on the new network.

The membership service provides each server with an identical view of the current global membership and the link weights of the current overlay network. Then, each Spread server independently calculates a shortest path multicast tree from each site to every other currently connected site. Since each server uses identical weights and an identical graph they are guaranteed to compute the same routing trees. The routes are calculated by applying Floyd-Warshall[Flo62] all-pairs shortest path algorithm to the set of sites and link-costs.

The major cost of using an overlay network is that since the overlay is constructed only between end nodes in the underlying network, inefficiencies exist in the routing paths. Some experimental work [ABKM01, hCRSZ01, SCH+99] has recently shown that the in-

efficiency is actually very small. This disadvantage is outweighed by several key benefits provided by the overlay architecture: First, the algorithms used in the overlay network can be easily changed and do not require changes to basic network infrastructure (e.g. routers). Second, routers can be made simpler and faster, while complex protocols and processing can occur on end nodes where more abundant resources exist. For example, as a result of the difficulties encountered while deploying and upgrading IP-Multicast in routers, most of the work on high level multicast services, such as reliability, uses an overlay network approach.

In general, Spread decouples the dissemination and local reliability mechanisms from the global ordering and stability protocols. This decoupling allows messages to be forwarded on the network immediately despite losses or ordering requirements. The only place where messages are delayed by Spread is just before delivering them to the clients, when delay is needed to preserve the semantic guarantees. Decoupling local and global protocols also permits pruning, where data messages are only sent to the minimal necessary set of network components without compromising the strong semantic guarantees.

Spread allows different low level protocols to be used to provide reliable dissemination of messages, depending on the configuration of the underlying network. Each protocol can have different tuning parameters applied to different portions of the network. In particular, Spread integrates three low-level protocols: one for local area networks called Ring, and two for the wide area overlay network connecting the local area networks; the standard TCP, and our new protocol called Hop.

As discussed in Chapter 1, one of the motivations for this approach to wide-area group communication is the realization that the main limiting factor for distributed algorithms in wide-area networks is the unavoidable latency required to communicate between nodes. While the physical network latency is unavoidable, the latency experienced by applications is often much higher then the required physical latency. This led to one of the main design criteria for this system: the additional latency imposed by the membership, ordering, and

dissemination protocols must be minimal. Ideally, the latency of a message should be the physical latency of the required dissemination network plus a variable delay to establish correct semantic guarantees. This might require delaying the message until its ordering has been established, or until it is known to be stable in the case of SAFE messages. The process of determining both ordering and stability should also require little additional latency. Ideally, the process should run in parallel with the dissemination, so that when a message arrives at a destination node, the information necessary to deliver it arrives within a small δ not much more than the delay required for necessary information flow.

A second result of the minimal latency goal is that as many actions as possible should be able to occur independently of any other system action. Independent actions minimizes the problem of dependency chains of actions that each require earlier actions to complete. These chains increase latency because dependent actions must receive information from the actions they depend on. The flow of information may include high latencies to cross the wide-area network. Avoiding chains also improves the potential throughput of the system by allowing more actions to occur at the same time in different areas of the network.

The key idea is to not delay messages prior to them arriving at receivers. If there is a delay, to delay them only if the delay is necessary to maintain the semantics the application requested. One of the discoveries of this work is that the delay caused by dissemination, flow control and routing is significant. Using custom network protocols to minimize this can provide significant advantages.

3.4 Group Scalability and Services

One of the fundamental scalability limits of any strong group communication system is the requirement that delivery of messages is tied to the membership of the group. Thus, all members of a group must at all times have accurate knowledge of the complete group membership. The costs of maintaining this state can be significant. One can decrease the costs

by recognizing that the failure of a process and the failure of a machine do not have the same consequences in real systems. The assumptions of an asynchronous network model are actually much weaker in practice then the actual behavior of a machine's operating system. Even if we maintain the asynchronous model for communication between machines, the model for processes interacting on one machine can be stronger.

The failure of a process is accurately detectable by the operating system kernel. Although some 'long' delays are possible if swapping and virtual memory are used, the kernel can have accurate, and timely information about the state of the system, even following a failure. The kernel therefore is able to reliably notify other local processes about the failure. In the case of a communication server, this could be by marking the socket connecting the server with the failed process as a closed socket. The difference between a very slow process, and a crashed process becomes detectable. Some forms of process failures, such as Byzantine failures, can cause behavior which is characterized as failed, but does not result in a crash of the process. Most operating systems provide protection against some types of misbehavior by processes. Higher level monitoring can sometimes detect such process misbehavior, although not in all cases.

When the processes and servers reside on different hosts, one no longer receives these stronger guarantees. Instead, one can use the existing failure detection built into standard Internet protocols such as TCP/IP. This is not reliable in the same sense as local kernel based fault detection. It is an accepted level of accuracy, however, with known behaviors and misbehaviors and is a usable substitute in practice when it is necessary or desirable to locate client processes on hosts separate from servers.

If a machine fails upon which a process is running, then processes running on different machines will have to detect the failure in the traditional way and will have no accurate means of differentiating a slow from a failed machine.

Maintaining the per-group state in the servers allows each server to fully synchronize with other servers only when a machine running a server fails and not when a machine

running a client fails or when a client process fails and the machine keeps running. Each server can track the status of the clients directly connected to it and delegate to other servers the tracking of all the other clients.

Using a client-server system to provide clients with the traditional View Synchrony or Virtual Synchrony model would provide some benefit over using a purely process based approach. However, maintaining View Synchrony requires synchronizing, sometimes called flushing, the message state at each process whenever a group membership change occurs. As a result, even though the servers can use faster, more reliable failure detectors and do not have to resynchronize membership state when a process failure occurs, they still have to synchronize the message state at all of the group member processes. This will make group join and leave events costly and require applications to consider them as heavy-weight events that must occur rarely. Yet, many applications need to join or leave groups often or may be of such size that joins and leaves will occur often despite attempts by the application to minimize them to avoid the scalability limits of the system.

Because of the high cost of group membership changes in View Synchrony, this thesis argues that using Extended Virtual Synchrony (EVS) as the base group membership model is a better design than using View Synchrony. EVS does not require end-to-end acknowledgments and synchronization of message state when group joins or leaves occur. EVS provides slightly weaker guarantees to the client application but improves the performance of basic operations such as join, leave, so to be useful for both weak-semantic and strong-semantic applications. Additionally, if View Synchrony semantics are needed or useful for a particular application, they can be provided by an additional library running on top of EVS at essentially no more cost then implementing native View Synchrony. For details see[Sch01].

The application will always need to synchronize some, so avoiding synchronization and end-to-end acknowledgments at the group layer does not avoid them entirely. However, applications can tightly control and limit their synchronization to exactly that required

by their particular services while the group layer imposes whatever synchronization it provides on every application that uses it. So, a general purpose group layer must minimize the synchronization and costs it imposes while still providing a useful set of semantics. Minimizing synchronization and costs in a network layer for distributed systems has the same advantages as it does in networking protocols.

3.4.1 Light-Weight Groups

The group abstraction provided by the system is fundamental to both the semantics and performance of a group messaging system. Groups can define many different aspects. For example, a group is often defined as a set of recipients of a message. That definition while accurate, does not capture a number of group properties. For example, does the group have a name by which others can address it? Or is the name just the list of members? The answer distinguishes a group which is just an aggregated target for a message from a group which has meaning outside the delivery of a particular message; i.e. are the recipients the most important component or is the group identity itself the most important concept.

If the group is considered as more then just a set of recipients, additional, stronger properties can be provided by the group. These properties enhance the system model from multicast-oriented to a group-oriented.

Once a group has state, for example, the set of group members or the set of messages that have been delivered to the group successfully, one can consider how strongly the group should enforce guarantees about its state and the messages that are sent to the group.

Light-weight groups provide two advantages:

1. Traditional advantage – only run membership and synchronization algorithms between nodes ONCE for all the groups. This is normal heavy/light weight groups.

2. Fast, lightweight Joins and Leaves – New – Each join or leave of a group is only an Agreed message and requires only a small local computation once messages is

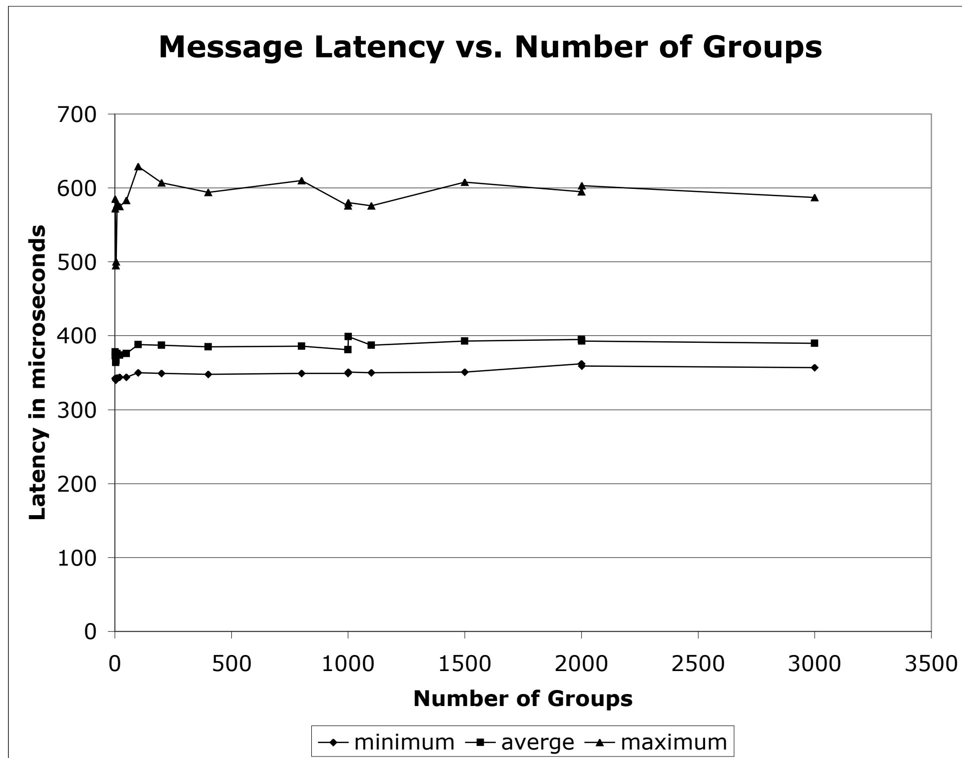

Figure 3.4: Scalability with Groups

received.

Scalability of the system as the number of groups increases is an important characteristic. Figure 3.4 shows the latency to send and receive a reliable message when the number of groups in the system increases. Each datapoint represents 10 messages and the system was running on a 100 Mbps Ethernet with two daemons.

When the number of connections to the system is increased as well as the number of groups the latency to send a message increases close to linearly, as is shown in Figure 3.5. This is mostly because of the overhead of using the "poll" system call on linux to check which of the connections currently have packets waiting to be handled. The poll call scales linearly, or worse then linearly, with the number of file descriptors it is checking. With more connections, checking what packets are ready for the daemon to process takes longer. This performance could be improved in several ways. First, some operating systems have a

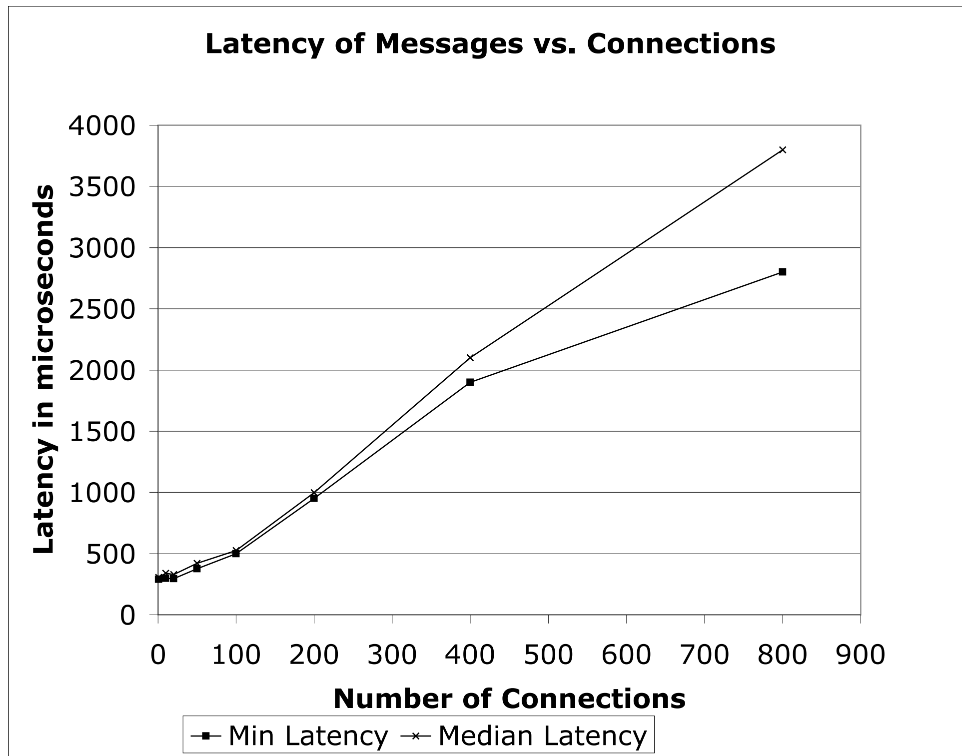

Figure 3.5: Scalability with Groups

different interface that is more scalable than poll to receive notifications of network socket activity. Furthermore, in a system with lots of busy connections instead of just one, the overhead of poll will be amortized over more actual work. As the system gets busier, the overhead will decrease.

3.4.2 Message Pruning

The two disadvantages of this light-weight group model are; first, the interference caused by the failure of hosts who are participants in the system but who have no members of a particular group; and second, the requirement that all hosts in the system receive all messages, even those who have no local clients.

The first disadvantage implies that if one instance of the system is used for many different purposes, for example different applications, the stability of the the system for all of

the applications will approximate the stability of the most unstable host running a daemon. This disadvantage can easily be mitigated by running separate sets of daemons for each non-interacting application. Then the set of hosts whose stability will affect the application is no larger then the set of hosts required for the application.

The second disadvantage of a light-weight group model occurs because all messages in the system are part of a single global total order. Thus, for any daemon to know the order of messages sent to some groups it has to also know about all of the messages sent to other groups. Clearly this is a tradeoff. If lots of groups have similar sets of members, then this is quite efficient. If most groups have disjoint sets of members, then messages must be sent to lots of locations that would otherwise not need them, thereby significantly increasing the message overhead. The advantage of a single global order is both a decrease in the state required to be tracked, from per-group sequence counts to a single sequence count, as well as a decrease in state exchange during membership changes, as a single number sent to other daemons communicates the set of messages instead of sending a number for every group.

It turns out that the cost of sending actual messages to all daemons can be avoided. Only a small set of meta-data about each message needs to be sent to all daemons. The actual message content, which is usually much larger then the meta-data, only needs to be sent to those daemons who have local clients who have joined the group. Thus, the realized overhead of sending a message is fairly small, even when only a small number of daemons have group members.

"Pruning messages" is the process of sending a message content only to the daemons who have clients who are members of the group. The multicast tree that disseminates messages from the source site to all of the other sites in the system prunes off branches which have no clients on them and instead of sending the data message, sends a small meta-data message that only contains the sequence numbers of the message and the originating daemon identity, but none of the additional information about the message. It is known that

no daemon down this branch will need any of the additional information.

On face, it is not obvious that this optimization is correct. If a process were to join a group at a daemon, say d_1 who had not previously had a member of that group, until all of the daemons were to learn of that join, daemon d_1 would not receive any of the data messages destined for that group and would not be able to deliver them to the new member. This problem does not actually occur because when the set of sites who should receive a message is calculated at the initiator of the message, the set is computed conservatively. This means that all sites who might need this message, even if they do not need it yet, will receive it. When a new member joins a group, that join does not take effect until the message containing the join becomes globally ordered as an Agreed message. That member, however, is added to the set of sites who receive messages for the group as soon as the join is seen.

3.5 End-To-End Principle

Many subscribe to the Internet's "end-to-end" principle that states any service that can be provided "efficiently and correctly" at a higher layer of the networking stack should be provided at that higher layer and not in any lower layer.

The "end-to-end" principle directly applies to distributed systems. Even if a lower level service could provide, for example, persistent storage of messages so that when a node fails and then recovers some other node can resend the messages the failed node missed, the application still must have its own stable message store to correctly match the state of the failed node with the continuously running nodes. The application must have its own knowledge of which messages were actually acted upon by the application. Over the last ten years, two different approaches have been proposed. The first integrates the networking protocols with the application, either as a library or a separate service that gives the application tight control on the service's state. The second decouples the lower-

level distributed services from the application, but provides well-defined guarantees to the application which allows it to minimize the additional work the application must do to synchronize.

The first approach, by giving the application complete knowledge and control of the state of the network layer, enables the application to make decisions based on network state and integrate application and network buffers. This coupling requires both the application protocol and network protocols to work end-to-end. Scalability limitations, caused by a lack of hierarchy, is the cost of working strictly end-to-end.

This approach was applied in the "Application Level Framing"(ALF) work developed by Floyd et al, [FJL+97]. ALF tried to solve this problem by requiring the communication protocol, in this case a reliable IP-Multicast protocol called "SRM", to give the application control of how messages were framed and stored. This allowed the application to not require its own storage since it directly controlled the networking layer.

The second approach maintains separation between the applications and network protocols. The network protocols run end-to-end on the underlying physical network, however, the application protocols establish a structure of processes providing different services. This structure allows aggregation of information and decreases in the number of parties involved in each service, thereby improving scalability. Spread follows this second approach.

Spread runs at the application level of the network protocol stack and requires minimal services from the network needing only unreliable, point-to-point datagram service and can use an unreliable local area multicast service if it is available. Spread thus avoids any specialized networking equipment or the deployment of any new networking protocols. From the network protocol stacks perspective Spread is an end-host application.

Applying the end-to-end principle to distributed systems shows potential. The principle clearly makes sense in a multi-layer distributed system. If a lower layer provides some service that all the upper application or layers do not need then it probably imposes unnecessary costs on those layers. However, the layers of a distributed system seem much less

well-defined then networking layers, and vary significantly between applications. Thus, the question becomes how to define appropriate, general purpose layers in a distributed system. Given the amount of discussion and disagreement that occurred in the past about the layered network model (OSI or Internet), universal agreement on a common set of layers for distributed systems will prove difficult if not impossible. Instead, layers can be used as an analytical tool that assist a system designer in understanding which services should be implemented in each part of the software. The layers appropriate for a particular application or set of applications can be specified by the designer. Those layered services then can be mapped to different pieces of software in order to provide a good balance of performance and flexibility to the application.

For example, a number of group-based applications might have a set of specified layers like this:

- Application
- Stability, Safety, Acknowledgments
- Ordering
- Groups, Message Delivery and Addressing
- Message dissemination

Message dissemination is probably a network level service, either provided natively by the operating system such as IP-Multicast, or by a network toolkit handles dissemination and reliability. The Ordering service may be done either by the application if the ordering constraints are very specialized, or by a messaging toolkit below the application. If the application does the ordering it gets fully control over what happens but at the cost of increased overhead and coordination costs because every application instance must participate in ordering its own and other's messages.

Stability and safety information can provide several different types of guarantees depending on the application's requirements. If the application requires knowledge that a

message has been acted upon by another application, then the acknowledgments and stability service must be implemented by the application itself, because only the application will know when it has completed acting upon a message. This type of stability is very costly, however, as every application must send an acknowledgment to every other application for every message. One of the key observations of researchers in group communication systems is that a layer of software running below the application can provide a type of stability information that is much less expensive to compute and can still provide the same correctness guarantees as end-to-end stability services. Therefore it might make sense for Stability to be split into two components, Network Stability, and Application Stability like Spread provides.

Chapter 4

Network Protocols

Thus far this thesis has presented the arguement that overlay networks are a sound architecture for building wide-area group communication systems. This chapter discusses the ideas and algorithms developed for Spread to construct and disseminate reliable messages over an overlay network.

Overlay networks must provide a way to both route messages and provide reliable delivery of messages. The goal is to extract the nodes from a network that are actively involved in a service, in this case those hosts running Spread servers, and have the nodes construct a network of links that connect them. The network creates routing tables so that a message sent at any one of them will be forwarded to all of the others. Additionally, if a packet is lost between two hosts, the packet or the complete message will be recovered by the network from a host who does have the message and it will eventually reach all of the destinations.

To provide this service each link between two hosts provides local reliability, guaranteeing that packets sent to the other side are either received or the link is declared failed. The overall overlay network is constructed by instantiating a set of such links sufficient to reach from every host to every other host in some reasable shortest path.

Since often some of the hosts participating in a group-oriented application may be collocated in the same local area network, the existance of LANs should be taken into account

when the overlay network is constructed. The approach taken here is to treat an entire local area network as one single host on the wide-area overlay network. This allows local area reliability and multicast protocols to run on the local area networks where they are very efficient and specific wide-area protocols are used on the high latency links.

The next several sections present the specific protocols used for reliability in both local area multicast networks, the Ring protocol, and wide-area point-to-point network links, the Hop and TCP protocols.

For all of these protocols, the discussion below is done in terms of individual control packets or data packets. However, as a performance optimization all of the actual protocols fill packets with as many data and control messages as possible, without introducing delay, so as to send full-sized packets.

4.1 Ring Protocol

The Ring protocol provides a high throughput, efficient reliable, ordered multicast protocol for local-area networks. Each site in the Spread architecture runs an instance of the Ring protocol among all of the servers in the site. Each server in the site has an identifier, their IP address, that is considered static. The identifier of a particular host, however, can change between a crash and restart as Spread does not maintain any state across a crash.

The Ring protocol is a modification of the ring protocol used in Totem and Transis. In Totem and Transis, the ring protocol provides reliability, global flow control and global ordering.

Spread uses the Ring protocol for one main purpose: packet-level reliable multicast and flow control within a site, and one secondary purpose: message-level stability detection for all the servers in a site. The same circulating token packet is used for both of these functions. In each cycle of the token the ring ARU calculation algorithm updates both the packet and message ARU fields. Therefore, the cost of using the token to maintain site

wide message stability knowledge is minimal.

In contrast to Totem and Transis, global ordering and flow control are provided by higher level protocols described in Chapter 5. The Ring protocol is very efficient in low-latency environments, but becomes extremely costly as latency increases. At the extreme, the entire collection of servers can be configured as one site, connected by routed IP-Multicast. This configuration would not take advantage of the optimized wide-area reliability protocols and would have poor performance.

The rotating token has the following fields:

link_seq The highest sequence number of any reliable packet sent on the ring.

link_aru The sequence number of which all reliable packets have been received up to by all members of the ring. It is used to control when a link can discard any local references to a packet.

flow_control A count of the number of packets sent on the ring during the last rotation of the token, including retransmits.

rtr list A list of all the sequence numbers that the previous token holder is asking to be retransmitted.

site_seq The highest sequence number of any reliable message originating on this ring. This sequence is local to the site and combined with the site_id provides a unique identifier of every message sent in the system.

site_lts The highest LTS value seen by any member of the ring so far. This is used to provide a causally consistent ordering for Agreed and Safe messages.

site_aru_modifier The identifier of the ring member that last modified the site_aru value on the token.

Upon receiving the token a server will handle any retransmits requested by the previous token holder then process messages received from client applications, send packets up to the limit imposed by the ring flow control. Finally, the server will update the token with new information and send it to the next server in the ring. After sending the token the server will attempt to deliver any messages it can to the client applications. For each message processed into the system, a new site_seq and site_lts value is assigned and the counters are incremented. For every reliable packet sent, a unique link_seq is assigned and the counters incremented.

To update the link and site aru values on the token, a server compares the local aru values with those on the token. If the local value is less, then the token value is lowered to be the local value and the site_aru_modifier field is set to the id of the server. If the local value is equal to or higher then the token value, and the site_aru_modifier on the token is equal to this servers id or is zero, then the daemon raises the token value to be equal to the local value. The site_aru_modifier value being zero indicates that no server lowered the token value during the last rotation. By following this algorithm, all members of the ring can calculate the highest aru value they can locally use by taking the lesser of the recently calculated token aru value and the token aru value from the previous round of the token.

The Ring protocol provides flow control within the site by limiting the number of packets each member can send during each rotation of the token. The number of packets which can be sent on the entire ring per round, and the limit on how many packets each individual member can send per round are tunable parameters. The server simply sends the lessor of its individual limit and the total limit minus the value in the flow_control field plus whatever it sent last time.

4.2 TCP Protocol

TCP is a very mature protocol and is the standard reliable transport protocol of the Internet. TCP provides stream based reliable transport with flow and congestion control. Because it is stream based TCP delivers data in a FIFO order. Using TCP as the point-to-point protocol in an overlay network is an obvious choice. It provides good performance, is very stable and has well-understood properties in wide-area networks. It provides a gold standard to which other protocols may be compared. For these reasons, the Spread overlay network supports using TCP as a link protocol.

It is fairly obvious that TCP provides the properties required by the Reliable Link specification in Section 2.2.7. By simply adding a length field to the beginning of every message written to the TCP connection and discarding any partial messages when a connection is closed, TCP can emulate the datagram-style service specified. TCP detects the failure of connections by either timeouts or active resets and so the link can declare an **`linkfailed`** event. On the overlay network links the **`rmcast`** events have only one destination. TCP clearly provides the necessary reliability and congestion control guarantees.

Messages can be forwarded out of order in overlay networks as the order of delivery is constructed at the end hosts who have additional information about how messages need to be ordered. The network is permitted to deliver them out of order.

When multiple TCP connections are chained together, as they are in an overlay network, the FIFO order on messages that TCP delivers can limit the end-to-end throughput and increase the latency experienced by messages. This occurs because when a packet is lost all later packets, even if they belong to different "messages," are also delayed because TCP will not deliver data out of order. However, while TCP is recovering the lost packet the next link in the chain of TCP links is idle, even though messages are available, waiting to be relayed by the available links. This idle link is an inefficient use of bandwidth and will limit the throughput of the system. The increased latency occurs because messages are blocked in queues behind other messages that contained packets dropped by the network

and awaiting recovery. If these later messages were not blocked, they could have passed on and reached their destinations in less time, and thereby experience less latency.

4.3 Hop Protocol

The second point-to-point protocol used to connect sites in the overlay network is the Hop protocol. The Hop protocol operates over an unreliable datagram service such as UDP/IP. The Hop protocol attempts to provide the lowest latency and highest throughput possible when transferring packets across wide-area networks. Three ideas characterize the Hop protocol.

1. *Non-Blocking:* packets are forwarded despite the loss of packets ordered earlier.

2. *Lazy-Selective-Retransmits:* nacks are sent for specific lost packets after a short delay to avoid requesting data which was not lost but merely arrived out of order or is sequenced after lost data.

3. *TCP fair flow and congestion control:* a TCP style flow regulator provides fair behaivor with competing TCP flows.

The Hop protocol establishes a bidirectional connection between every two servers who should connect on the overlay network. Each pair of servers maintains several counters and a table of open packets which have not yet been acknowledged. To establish reliable transmission in the presence of losses, Hop uses selective nacks where the receiver requests specific data packets (identified by their sequence number) when loss is detected. The receiver continues to request lost packets until it has recovered them.

Hop has several ways of detecting when a link fails. In all of these cases Hop will trigger a membership change when it detects a failure. The first method of failure detection involves a failure to recover a lost packet. If a lost packet has not been received after the receiver has sent k nacks, the receiver declares the link failed. This particular failure case

Hop: Sender protocol

```
1  switch (event)
2  lrecv(NACK) :
3
4       N[nack.n] ⇐ N[nack.n] + 1
5       send(receiver, Msg[nack.n])
6       if N[nack.n] > MaxNACKS then
7          linkfailed()
8  ACKtimeout :
9
10      ack.max ⇐ Seq
11      send(receiver, ack)
12      schedule_timeout(ACK)
13 lsend(PACKET) :
14
15      Seq ⇐ Seq + 1
16      Msg[Seq] ⇐ packet
17      send(receiver, packet)
18 lrecv(ACK) :
19
20      for i ⇐ smallest(Msg) to ack.aru
21         DISCARD(Msg[i])
```

Figure 4.1: Hop Link: Sender protocol

is necessary to eliminate the "failure to receive" problem that can occur either because of a networking fault that deletes certain packets or a malicious attacker who keeps removing one particular packet from the network. The second method of failure detection involves the inverse problem, where a sender keeps resending a packet, but the receiver also keeps sending nacks for it. This indicates that an asymmetric network situation exists where the sender can receive responses from the receiver, but the receiver cannot receive packets from the sender. After resending the packet $MaxNACKS$ times, the sender will fail the link.

The receiver has two methods to detect loss. First, when the receiver receives a packet which is sequenced beyond the next expected packet, it adds the sequence numbers between the highest previously received sequence number and the just arrived packet to a list of proposed lost packets. This method is shown in Figure 4.2. The receiver schedules a nack packet containing the newly lost sequence numbers to be sent in a short time if the packets

do not arrive. In real-life networks one expects some reordering of packets, so the delay can help avoid false positive loss indications. The delay does add a cost in both the latency to recover the truly missed packets and the memory requirements because the sender must store a larger number of messages. The delay is small and fixed. It should be possible to set the delay based on the amount of reordering experienced.

Second, to detect loss when no further packets are sent, or when there is a long time before the next packet, the sender sends a link acknowledgment to the receiver periodically, based on time and number of packets sent and received, which specifies the highest sequence value sent by the sender. This is shown in Figure 4.1. Sequence numbers that are equal to or below the specified value that the receiver has not yet received are added to the list of missing packets. After a short delay these sequence numbers are sent to the sender in a nack packet.

When the sender receives a nack packet it adds the packets represented by the requested sequence numbers to its outgoing retransmit queue as shown in Figure 4.1. The packets will be sent along with other data when flow control allows. Retransmissions are sent even when the limit on the number of outstanding packets has been reached. Therefore, the packet will cross the link in a bounded time or else the link will be declared failed.

The Hop protocol eliminates duplicates by checking every received packet against the list of known missing packets. If the received packet's sequence number is less than the highest previously received, either the missing packet list contains the packet or the packet has already been received. Therefore, if the missing packet list does not contain the packet, the packet is a duplicate and the receiver discards it.

To enable the sender to release buffered copies of packets, the receiver sends link acknowledgments either periodically or after some number of packets, whichever is sooner. The generation of these acks can be seen in Figure 4.2. These acknowledgments contain the sequence number of the packet for which all previous packets were received. This combination of acks and nacks provides a fully reliable channel with bounded buffers. However, it

Hop: Receiver protocol

```
1  switch (event)
2  lrecv(PACKET) :
3
4      Packets_since_ACK ⇐ Packets_since_ACK + 1
5      if Packets_since_ACK > MaxUnAckedPackets then
6          ack.aru ⇐ H_ARU
7          send(sender, ack)
8          Packets_since_ACK ⇐ 0
9      if packet.seq > Next_Packet then
10         for i ⇐ Next_Packet + 1 to packet.seq
11             Missing ⇐ Missing ∪ i
12         Next_Packet ⇐ packet.seq + 1
13         schedule_timeout(SendNACK)
14         deliver(packet)
15         return
16     if packet.seq = Next_Packet then
17         deliver(packet)
18         if H_ARU = Next_Packet then
19             H_ARU ⇐ H_ARU + 1
20         Next_Packet ⇐ packet.seq + 1
21         return
22     if packet.seq < Next_Packet then
23         if packet.seq ∈ Missing then
24             Missing ⇐ Missing − packet.seq
25             deliver(packet)
26             if packet.seq = H_ARU + 1 then
27                 H_ARU ⇐ smallest(Missing) − 1
28         else DISCARD(packet)
29 lrecv(ACK) :
30
31     if ack.max > Next_Packet then
32         for i ⇐ Next_Packet + 1 to ack.max
33             Missing ⇐ Missing ∪ i
34         Next_Packet ⇐ ack.max + 1
35         schedule_timeout(SendNACK)
36 SendNACKtimeout :
37
38     for each m in Missing
39         if NS[m] > MaxNACKS then
40             linkfailed()
41         NS[m] ⇐ NS[m] + 1
42     send(sender, nack)
```

Figure 4.2: Hop Link: Receiver protocol

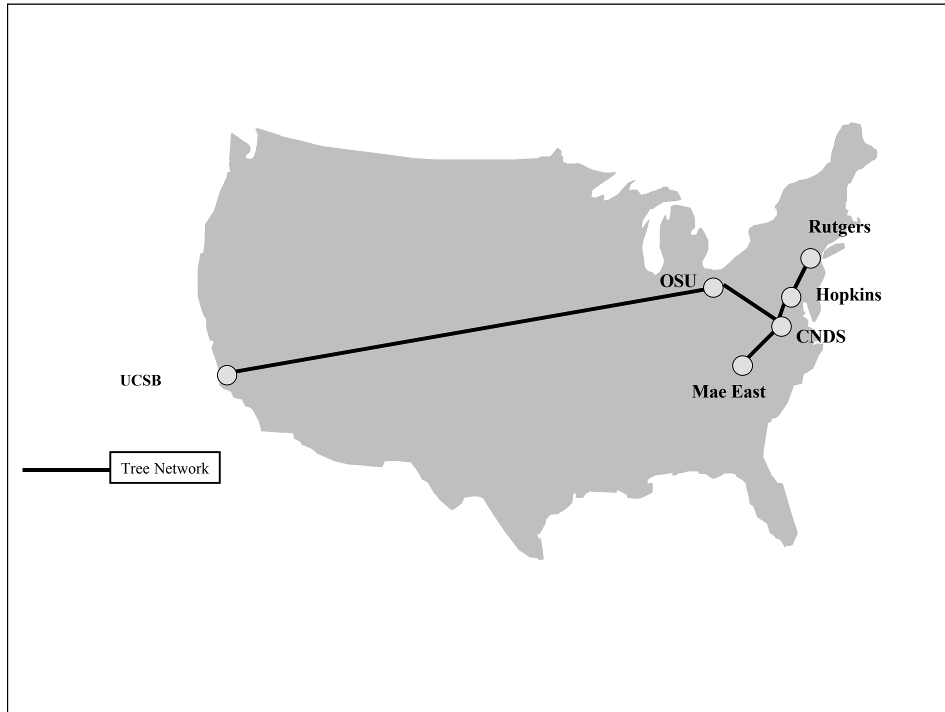

Figure 4.3: Experimental Network for Tree Network

does not create any restrictions on the ordering of messages so that each packet is delivered as soon as it is received by the Hop protocol.

The Hop protocol uses rate-based flow control to limit the rate packets are sent, and a maximum window of outstanding packets to provide termination guarantees for the reliability protocol as described above. The rate regulator is a leaky bucket with both a maximum burst size and an average rate limitation. All of these parameters are set per link so that they can be tuned separately for each link in the overlay network.

4.4 Evaluation

To evaluate the utility of construting distinct multicast trees for each sender the achieved throughput with two different routing matrices is compared.

For this experiment two different overlay networks were created between the six sites

shown in Figure 4.3 by adjusting the weights of the links in the configuration of Spread. A Fanout network contains a direct link between each two sites so that every source sends directly to every other site. This was created by assigning equal weights to every link. A shared-tree multicast network was created as labeled in Figure 4.3. This tree was constructed based on measurements of network latency. The experiment shown in Table 4.1 was conducted using TCP as the link protocol in Spread.

Table 4.1: Throughput using TCP and several network configurations.

Sending Site	Fanout	Tree
Mae East (Kbits/sec)	487.32	559.79
CNDS (Kbits/sec)	666.94	560.71
Rutgers (Kbits/sec)	184.99	568.34
UCSB (Kbits/sec)	328.81	578.84

In tests run on each of the two networks (Fanout and Tree), for every test, one of the four sites (Mae East, CNDS, Rutgers, UCSB) was a source of a stream of 10,000 reliable messages of 1024 bytes. The sending application on that site always made messages available to Spread. The remaining five sites were running a receiving application that computed the running time of the test at that site. The numbers in Table 4.1 represent the throughput of the slowest receiving site measured in kilobits per second. The difference between the fastest and slowest receiver in most of the tests was negligible. As in any reliable multicast system, the maximum sustained throughput is limited to the throughput of the slowest link.

Table 4.1 shows how both Fanout and multicast Tree overlay networks are each better than the other for different source sites. When the CNDS site is the source the Fanout network provides better throughput. This is probably because CNDS has extremely high throughput connectivity to the Internet and thus the first few hops do not form a bottleneck. However, when Rutgers or UCSB are multicasting, the multicast Tree network yields much better throughput. Even though the Mae East site is located very close to a major Internet backbone peering point, providing better connectivity then almost any typical server, the

multicast tree network was still 15 percent better then the fanout network.

This experiment validates the usefulness of source based routing using the overlay networks. For example, while messages generated by CNDS can be sent through a fanout configuration, messages sent by UCSB will be sent using a tree configuration.

4.4.1 General Comparison

Both the TCP based multicast and Hop based multicast protocols have several advantages over emulated multicast, where end-to-end links are used between all the application instances. Not only can they utilize multicast trees to avoid sending N copies of the data across the network, but they can also achieve localized recovery of lost packets without requiring the original sender to re-send the data. Localized recovery is crucial for high latency multicast networks, not only for large[Hof96], but also for small groups where the members are widely dispersed in the network. Each unicast link in the Spread multicast tree has buffers on the sending side to store data until it has successfully reached the other end of the link.

Most current reliable multicast protocols have some form of localized recovery, such as creating virtual local subgroups along the tree[Hof96], or using nack avoidance algorithms and expanding ring nacks[FJL+97]. All of these techniques approximate recovery of the missing data from the closest node. Spread actually does recover the data from the closest node on the overlay network.

4.4.2 Performance Comparison

In this section the behavior of Hop and TCP is compared on a high latency network with a background load to simulate active working conditions. The background load also triggers additional loss on the network that would not occur in a lightly loaded network.

All latency tests were done by an application level program which multicasts a reliable Spread message to a group and then listens for a response message. A second application

runs on the other site and acts as an echo-response server, sending anything it receives immediately back to the sender through Spread. The sender application calculates round-trip latency times by taking the difference between the time it received the echo-response and the time it sent the original message. These latency tests are repeated 30 times back to back and the minimum, average, and maximum are reported. All results are reported as round-trip times, which include time transferring the message from the client to the Spread daemon, processing time in the daemon, network transfer time, the receiving daemon's processing time and the transfer to the receiving application, and a similar reverse path back to the sender. For the tables and figures reporting 'ping' results, the standard 'ping' program was run from between the daemons using 1024 byte packets. The ping latencies provide an effective lower bound.

Table 4.2: Link Latency (Mae East to UCSB).

	ping	tcp	hop
min (ms)	103.3	108.946	107.798
average (ms)	104.1	136.277	108.493
max (ms)	106.8	311.649	110.944

Table 4.2 shows the single link latency for a link between Mae East and UCSB for 1024 byte messages. Clearly the ping latency is the best, however, both the TCP link protocol and the Hop link protocol have minimum times very close to ping. The Hop protocol also is very stable across all the tests, with a variance of only 3 milliseconds, the same as ping, while TCP produced a large variance of over 200 milliseconds between the minimum and maximum latency.

To more realistically evaluate latency over a wide-area network an overlay network was constructed of six sites in a chain. This chain is shown in Figure 4.4, as running from Mae East to UCSB to OSU to Rutgers to CNDS to Hopkins. Note, clearly this is not a practical setup, or even an efficient chain. However, using this chain demonstrates how the protocols interact when packets must be forwarded many times, and how the performance of the

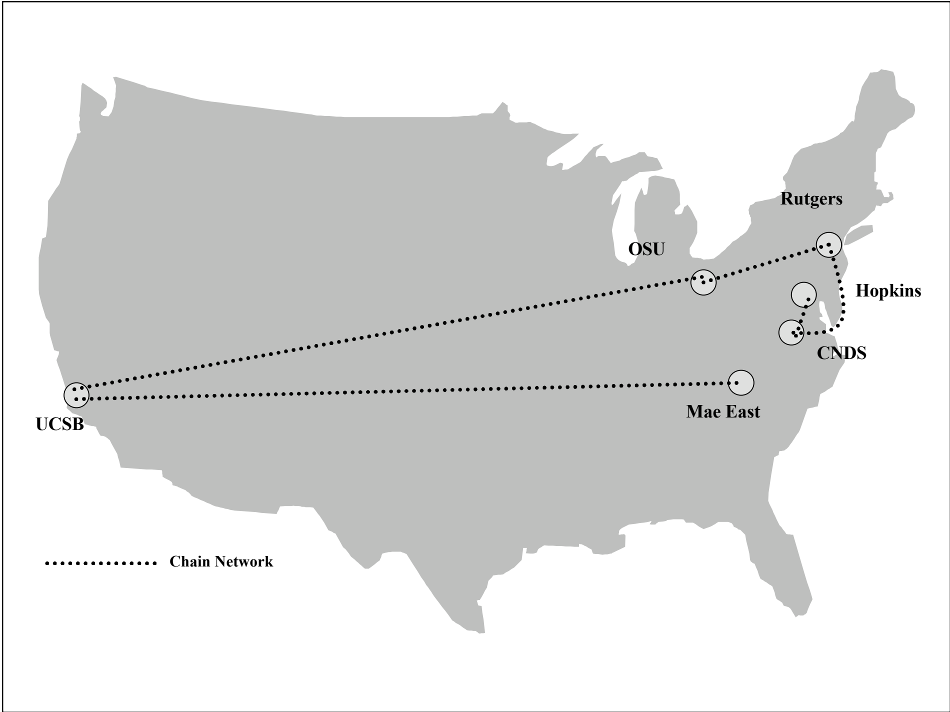

Figure 4.4: Map of Chain Network

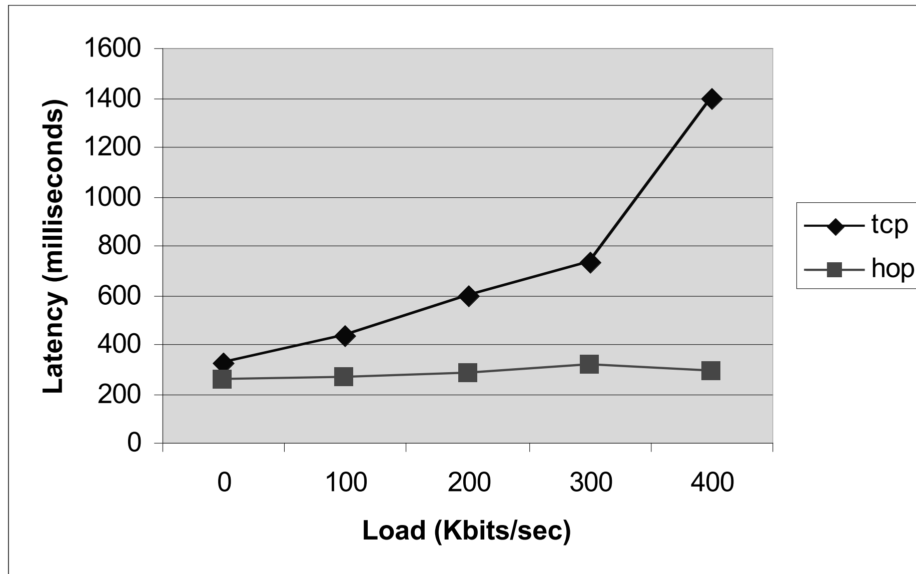

Figure 4.5: Latency of Messages under Load (Comparing TCP and Hop)

protocols scales with the diameter of the multicast network.

The latency of the two protocols under load was used to evaluate them under the expected conditions on networks with variable load and higher loss rates then local area networks. The results presented in Figure 4.5 use the chain network. In this test a load application using Spread was flooding the network from Mae East with a controlled level of messages per second. Concurrently, the latency test application measured reliable, 1024 byte message latency between Mae East and Hopkins. The Hop protocol has almost constant latency as the background load increases from 0 to 400 kilobits per second. The stability under load is attributed to the Hop protocol's forwarding policy, which does not delay packets even when there is loss or other application traffic. TCP latency shows a steady increase as the background load increases with a jump between 300 and 400 kilobits per second where the latency grows to almost a second and a half.

The latency of Hop will start to increase as the buffers in the Spread routers fill up and it takes longer to forward each packet. This behavior is unavoidable and can only be minimized by limiting the sending throughput of the applications to below the bottleneck link's available bandwidth so the overload condition never occurs. This graph does not show the increase in latency only because it does not include to a high enough load level to trigger it on the test network.

Chapter 5

Global Ordering and Safety

In the absence of failures or recoveries, the core services provided by a group messaging system are reliable and ordered delivery of messages. A system spends most of its execution time disseminating, ordering, and delivering messages. The types of message ordering and reliability provided by a system, as well as the speed at which it can perform those services, is the prime determinant of how useful the system is for a broad class of applications.

In the previous chapter the methods by which messages are reliably disseminated were presented. This chapter discusses how those messages may be ordered efficiently, and how message stability can be determined.

Three types of message orderings are provided by Spread: Unordered, FIFO, and Agreed. Unordered is useful for applications where each message can be handled independently or for applications who need specialized orderings, because the application can provide it's own ordering on top of Spread. The most traditional ordering in Internet applications is FIFO order, where messages that originate at a particular source are delivered in the same order as they were sent in. This is very useful for two party communication, and is also useful for groups where the interleaving of messages between senders does not matter. FIFO is also very useful for collaborative applications or control channels, where

several programs are sending a stream of commands to a service, and want to make sure their commands are done in order, yet they do not care about the commands sent by other programs. Finally, Agreed order provides a global ordering across senders so that everyone who receives the messages receives them in the same order, no matter who sent them. The Agreed order is consistent with the FIFO order in that messages that are sent Agreed from the same sender will be delivered in FIFO order with regards to that sender. Agreed order simplifies the design of consistent distributed applications. In the case where no network partitions are possible, sending transactions in Agreed order to a set of replicas is sufficient to ensure that the replicas remain consistent, assuming that the transactions are deterministic.

One subtle ordering question in group communication systems is how the order of messages sent to different groups interact. If two senders each send a message to two different groups, the question is whether someone who joins both groups will see the messages in the same order, or whether one of the receivers might see the message to group A prior to seeing the message to group B while the other one sees the opposite order. Both receivers see the same order of messages within the group, so the traditional group ordering properties are provided, but the ordering of messages between the groups is different.

A system that provides "inter-group ordering" guarantees all messages are globally ordered across groups so they will be received in the same order no matter whether they are destined for one group or many different groups. This property strengthens the system by allowing groups to be used in a consistent way. For example, if each group represents a shared resource and an application wants to acquire locks on three resources, the global ordering of messages to all groups allows all replicas of the resources to know whether or not the attempt to acquire resources succeeded in all groups. The members will all try to acquire the resources in the same order.

One can view all of the groups managed by a Spread configuration as an "ordering domain" that provides agreed ordering. If a particular application has several disjoint sets of

groups that do not require any inter-group ordering, it might be beneficial to run separate Spread configurations for each set of groups. The performance benefit of separate configurations will only occur, however, if the sets of groups involve different sets of machines participating in the groups. If all of the machines participate in all, or almost all, of the groups there is no performance cost to the inter-group ordering, as all the machines have to participate in the ordering anyway. Using separate configurations, in that case, will be less efficient as each configuration will have to exchange its own control messages and so the message overhead will be greater.

Message stability knowledge is required for two distinct purposes:

1. *Memory Buffer release.* When a message has successfully been disseminated to all the daemons who need it, the memory associated with that message can be freed as soon as the local daemon is finished.

2. SAFE *message semantics.* The properties of SAFE messages includes a guarantee that the message will only be delivered to an application after it has reached all the daemons. A network partition will not cause the message to be only delivered to some of the recipients.

Memory buffer release is typical of any reliable message passing system, because it is always necessary to know when messages have been successfully sent so the resources associated with them can be released. In order to accomplish this goal without delaying the system, it is only necessary to learn that resources can be released at least as fast as the average rate of new messages being inserted. Thus the resource release could be accomplished by a background task or some occasional acknowledgment or cleanup messages. Since the required performance is tied to the average load on the system, no absolute bounds on the time to recover resources are needed.

When SAFE messages are added to the specification the performance requirements change significantly. Because SAFE messages can be used as a key tool for building con-

sistent replicated transaction-oriented or database systems, the absolute time required to deliver a SAFE message is a critical performance characteristic of the system. For this purpose, knowledge of message stability must not only keep up with the average throughput of the system, but must also have an absolute time bound from when a message is sent to when all daemons have knowledge of its stability so they can deliver it as a SAFE message.

5.1 Ordering Algorithm

The basic ordering algorithm uses the approach of assigning each message an identifier and partial ordering information when the message is created. The information assigned at creation is only that which can be determined locally by the daemon or by the set of daemons who make up a site and only requires at most a local area delay. When a message is received a determination is made as to whether sufficient information from the sending daemon as well as other daemons at other sites is present to correctly order the message. If it is, the message can be immediately delivered to the application. If not, then the message is queued until additional information arrives from other daemons which determines the order. This information is guaranteed to arrive within a bounded time as every daemon sends an ARU_Update message at least every X seconds, and usually will be available within a time no more then one diameter of the network after the message originally arrived.

One of the known limitations of ordering algorithms is that supporting partitionable operation of a consistent service requires the messages that are ordered are "born ordered." This means that the order in which messages will be delivered is determined at the time of message creation and at the source of the message and that the order will not change from what was determined at creation. For example, an ordering algorithm that sent the messages with only a unique identifier, and had an elected "sequencer" process assign an order to all of the messages identifiers, would be insufficient. The intuition is that if a network partition occurs and a new sequencer is elected it may order messages differently

then the previous sequencer, that did not crash, but is still executing in a different network partition.

In the case of Spread, the additional information required to order a message does not change or determine the order, but only reveals the static, source assigned order. The revealing of the global order requires receiving messages from other sites so all of the servers are aware of all of the messages that might be ordered earlier than the one they are trying to deliver.

For FIFO ordering, the information to deliver the message is always present as soon as the message arrives, because it is included on the message itself. The only reason FIFO messages may not be delivered immediately is because prior messages from the same initiating session have not yet arrived. This is possible since the network can reorder messages. To guarantee FIFO order the earlier messages must arrive before delivering the later messages.

For Agreed order, a daemon must receive a message from every other site, not every other daemon, indicating that no message with earlier sequence values will be sent and the receiving daemon must verify all messages with earlier sequence values from any site have already been delivered.

Every message is assigned an identifier, that within a membership, uniquely identifies the message among all of the messages in the system. The identifier is made up of the client session name, which includes which server the client initiating the message is connected to, and a sequence number that is incremented for every message that session initiates. Since the membership is known, mapping the sending server to the the site in which the server resides is trivial.

The message identifier for every message includes:

seq a sequential, per site counter that is increased for every message that originates at a site.

lts a Lamport timestamp.

session_seq a sequential, per session counter that is increased for every message sent by a client session.

Each server tracks several variables to compute the order and stability of messages. These variables are:

AgreedLine $= min(C[i] : i \in CurSites)$

ARU $= min(HA[i] : i \in CurSites)$

C vector of sites. Stores the highest sequence number of which all messages with sequence numbers less then this have been received.

HA vector of sites. Stores the highest ARU value received from each site.

Session_seq tracks the highest session_seq value of which all messages with lesser session_seq numbers have been received. Session_seq is used to calculate which FIFO messages are deliverable.

When a complete message is received, Spread inserts the message into an ordered list, sorted by lts value and the originating site. Then Spread attempts to deliver any messages now deliverable, possibly including this new message. Delivery is attempted by following the delivery algorithm defined in Figure 5.1. This algorithm walks through the ordered list of received, but not discarded, messages and attempts to deliver all undelivered messages permitted. Four rules define which messages are deliverable and which can be discarded.

1. The first rule states that any message with an lts value less then the current ARU can be both delivered and discarded.

2. The second rule states that any message, except those of type Safe, with an lts value less then the current $AgreedLine$ can be delivered.

3. The third rule states that any message of type FIFO can be delivered if the session_seq value of the message is equal to the last sequence number delivered from that session plus one.

```
Deliver_Mess
 1  while entry ≠ NIL and entry.lts < ARU
 2  do
 3      if entry.delivered = FALSE then
 4          deliver(entry.msg)
 5          entry.delivered ⇐ TRUE
 6          UPDATE_FIFO(entry.msg)
 7          DISCARD(entry.msg)
 8      entry ⇐ entry.next
 9  while entry ≠ NIL and entry.type ∈ {R, F, A} and entry.lts < AgreedLine
10  do
11      if entry.delivered = FALSE then
12          deliver(entry.msg)
13          entry.delivered ⇐ TRUE
14          UPDATE_FIFO(entry.msg)
15          entry ⇐ entry.next
16  while entry ≠ NIL
17  do
18      if entry.delivered = FALSE and (entry.type = R or
19          (entry.type = F and
20          entry.session_seq = NEXT_FIFO_DELIVERABLE(entry.session))) then
21          deliver(entry.msg)
22          entry.delivered ⇐ TRUE
23          UPDATE_FIFO(entry.msg)
24          entry ⇐ entry.next
```

Figure 5.1: Deliver_Mess function

4. The fourth rule states that Reliable and Unreliable messages may be delivered as soon as they have been completely received by the server.

5.1.1 Proof of Ordering Properties

The ordering algorithm must provide the message semantics defined in Chapter 2. This section provides a proof that messages are delivered in the correct order in all cases. Specifically, the properties associated with Properties 2.11, 2.12, 2.13. The message properties involving the interaction of membership changes and message deliveries are discussed in Chapter 6.

We first show a lemma that ties the lts sequence assigned to all messages to the abstract

```
Deliver_Safe_Mess
1  while entry ≠ NIL and entry.lts < ARU
2  do
3     if entry.delivered = FALSE then
4        deliver(entry.msg)
5        entry.delivered ⇐ TRUE
6        UPDATE_FIFO(entry.msg)
7        DISCARD(entry.msg)
8     entry ⇐ entry.next
9
```

Figure 5.2: Deliver_Safe_Mess function

ord function which represents the "correct" global ordering of all messages.

LEMMA 5.1 ()
The lts timestamp on messages is consistent with the ord function.

PROOF: The ord function by definition is consistent with the $\rightarrow$ operator which defines a causal relation. LTS timestamps are also consistent with the $\rightarrow$ operator because the timestamp is always increased when a message is sent, and the timestamp is always increased when a message is received from another sender and the received message has a higher timestamp. The lts timestamp is transitive because only a single local timestamp clock is maintained and it is updated by messages from all other servers. Thus, the lts timestamp is consistent with the ord function. □

THEOREM 5.1 (FIFO MESSAGES)
If a server sends a FIFO message after sending a previous message then all servers who deliver both messages deliver them in the order in which they were sent.

PROOF: When a message is sent, a session_seq value is attached to it. The delivery rules, as shown in Figure 5.1, only delivers FIFO messages when either they are totally ordered, lts $>$ AgreedLine and all messages with lessor lts have been delivered, or all messages with session_seq less then this message have been delivered. Thus, FIFO messages will always be delivered in order. □

THEOREM 5.2 (CAUSAL MESSAGES)
If a server sends a causal message m' such that the send of another message m causally precedes the send of m', then any process that delivers both messages delivers m before m'.

PROOF: Causal messages are treated as Agreed messages. Agreed messages are delivered in an order consistent with the causally precedes Definition 2.6 because they are delivered only in order of increasing Lamport Time Stamps lts values, which are consistent with the $\rightarrow$ relation. If two messages have the same lts value, then they also do not causally precede each other and so are not relevant to this theorem. □

THEOREM 5.3 (AGREED MESSAGES)

(a). Agreed messages are Causal messages.

(b). If a server s delivers an agreed message m, then after that event it will never deliver a message that has a lower ord value.

PROOF: Theorem 5.3(b) is provided by the normal delivery algorithm in Figure 5.1 that will only deliver an agreed message if all messages with lessor lts value have already been delivered. By Lemma 5.1 the lts timestamp is consistent with the ord values, so a message with a lower ord value will not be delivered, because that would require it to have a lower lts timestamp then an already delivered message which violates the algorithm. Since ties in lts timestamp are broken by the server identifier who originally sent the message and that provides a deterministic order, everyone who delivers two messages with the same lts value will break the tie the same way and deliver the two messsages in the same order. □

5.2 Performance

The evaluation of the ordering algorithms can be divided into the four types of messages: Unordered, FIFO ordered, Agreed ordered, and SAFE.

The Unordered messages are reliable messages and the latency of their delivery and the throughput possible is entirely determined by dissemination time and reliability costs.

These were discussed fully in the previous chapter. In summary, the latency of an Unordered message sent by a daemon at site s and it being delivered to the last group member is approximately $1/2 * T^s_{rotation} + max(D^s_i | i \in Sites) + L_i$ where $T^s_{rotation}$ is the time it takes for the token at site s to rotate among all of the local daemons, D^s_i is the network distance from site s to site i, and L_i is the time for a message to be sent from one daemon in site i to any other daemon in site i. This calculation is assuming no packets are dropped and have to be recovered. In case of packet loss, the time to detect the loss and recover the missing packets is added to the above latency.

FIFO ordered messages have the same basic latency as Unordered messages, but the latency can be larger in the case of packet loss because packets lost not only from this message will cause delay but also packets lost from any previous message will cause this one to be delayed.

Agreed ordered messages require receiving a message or ARU_Update from every other site with an lts value at least as high as the message that is being delivered. If all of the sites are initiating new messages at a similar rate, then this could, in theory, require as little time as one diameter of the network plus a small epsilon. This best case occurs because with all of the sites sending messages, shortly after receiving the message that needs to be ordered.

The daemon will also receive messages from all of the other sites and these messages will have lts values close to that of the message. This is favored because the information required to order this message is traveling the network in parallel with the message itself, minimizing the latency by avoiding any request-reply communication pattern. If only some of the sites are sending, however, then the sites who are not sending data messages must send ARU_Update messages that reach all of the other sites before the Agreed message can be delivered. The ARU_Update includes the current lts value that this site will use for its next message and so it informs all sites that the site sending the ARU_Update message will not initiate new messages with a lower lts.

The worst case latency can be minimized to only $2 * Diameter$ if every site sends

an ARU_Update message as soon as they receive an Agreed message. Every site is then guaranteed to get a message, the ARU_Update, from every other site no more then two diameters of the network after the message is initiated. The overhead cost is substantial. Every message now adds an additional N broadcast messages if there are N sites. This is unacceptable. The approach adopted here is to trade off bandwidth for latency in order to achieve bounded latency at a reasonable overhead. To do this, ARU_Updates are only sent immediately if one has not been sent in "delta" time. The definition of "delta" is not precise but would probably be slightly larger then the diameter of network. If one has been sent more recently, it is not sent again immediately, but rather is marked to be sent later when the current time is at least delta after the last time the ARU_Update was sent. The possible latencies are shown in Table 5.1.

Table 5.1: Latency of Agreed Messages.

	latency
Best Case	1*D
High Overhead	2*D
Balanced	2*D + delta
Zero Overhead	infinite

SAFE messages are also Agreed messages so their latency will be at least the Agreed latency. Gathering stability information from all of the sites can be done in a number of ways. In order to minimize the overhead on the wide-area links cumulative acknowledgements are used so each ARU_Update only takes up 4 ints (usually 32 bytes) regardless of the number of sites or daemons in the system. Because these are cumulative acknowledgements, they only acknowledge those messages that have been received in Agreed order. Therefore before an ARU_Update is sent indicating a message is stable, that message must be in Agreed order. Thus, after a message becomes Agreed, an ARU_Update will be sent within delta time and received by all of the other sites within delta + D time. The worst case latency for SAFE messages is $3 * D + 2 * delta$, since it takes $D + delta$ time after

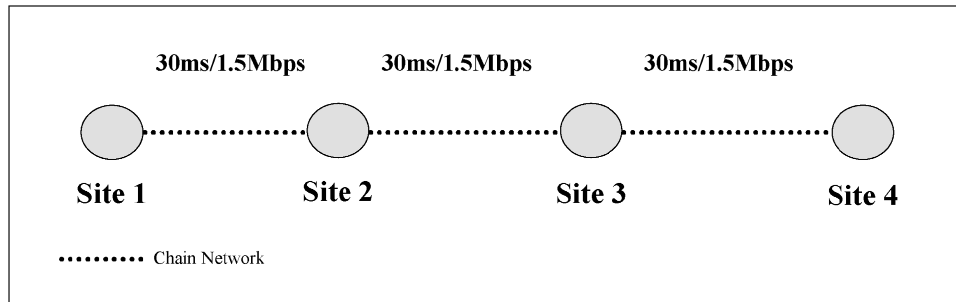

Figure 5.3: Wide Area Chain Network

the message becomes Agreed.

The predicted performance was validated with wide-area network experiments conducted on the Emulab facility at the University of Utah. For this experiment the configuration, as shown in Figure 5.3, consisted of four Pentium III 800 hosts running Linux with the ethernet links between them passed through a FreeBSD router acting as a traffic shaper. The router limited the bandwidth to the specified 1.5Mbps and added latency to make the total one-way latency 30ms between each of the hosts. Each host ran one Spread server and the two hosts at the end of the chain also ran two Spread client programs. The first program sent a controlled stream of messages through the Spread network to a receiver at the other end. This stream was regulated to provide a fixed background message rate. The second program sent a single 'ping' message from the first server to a group that an instance of the second program running on the last server joined. The program on the last server then sent a reply 'pong' message back to the original sender. The round-trip latency of these ping-pong messages was measured at the original sender.

The experiments shown in Figure 5.4 were conducted 90 times for each type of ordered message and the background traffic was of the same message type as the ping message. If the latency was being measured for Agreed messages, the background traffic was also Agreed messages. The test compares the overall costs of the different message types, how-

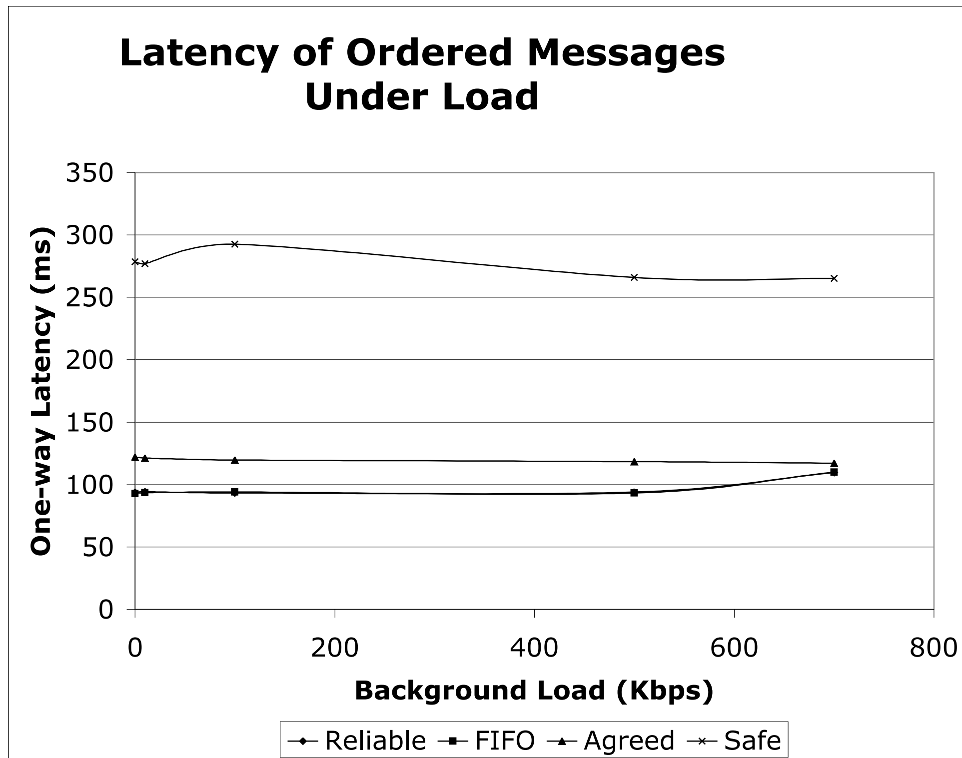

Figure 5.4: Wide Area Latency of Ordered Messages

ever since the background traffic of Agreed or Safe messages costs more in overhead messages then Reliable or FIFO, the effective background rate is slightly higher in the Agreed or Safe case. The results show that the experienced latency is very similar to that predicted. In a network with a diameter of 90ms, the Reliable and FIFO messages experienced a latency around 93ms, and since the delta factor in these tests was 100ms, the average latency of Safe messages of 270ms is right in the expected range. The slight decrease in latency as the load increased for Agreed and Safe messages shows how as more messages are active, the delay and overhead of ordering and gathering stability information decreases.

Chapter 6

Daemon and Group Membership

The most sophisticated service provided by group communication systems is a view-oriented membership service. This service provides the application with both knowledge of which other applications are currently running and connected as well as a strong set of guarantees about the inter-relationships between changes to the membership set and the messages that are delivered. The membership service makes fault-tolerant and consistent replication applications possible with group communication systems.

The membership algorithms presented here do rely to some degree on the other services provided in Spread but they also have a method of bootstrapping the system when no other services are running yet. This is required because most of the other services assume the knowledge of a currently stable set of servers. To best use this interlocking relationship between the membership algorithm and the other services, membership is divided into three distinct stages, made up of a number of states and algorithms.

The first stage assumes no services besides unreliable UDP datagrams are available and only some static information about the potential location of other services. This stage of the daemon membership algorithm gathers the currently running and reachable set of daemons and constructs the new server membership view that will attempt to be delivered. This stage ends by flooding a message about this new server membership to all of the servers.

The second stage begins when that message is received because based on it all of the other services such as the overlay network links, reliability, ordered messages, safe deliver, etc. can not restart with the new server configuration. Now that the services are running, the daemon membership algorithm exchanges state messages with all of the other servers who are also in the new membership. Finally, the messages the other servers are missing are exchanged and the daemon membership algorithm delivers the new view and completes.

The third and final stage begins once the daemon membership is complete. This stage is the group membership algorithm and translates the single group of servers into separate views for each light-weight application group that exists in the system. This algorithm not only can assume the basic services of the daemons, such as reliable and ordered messages, but also the daemon membership algorithm correctly provides strong EVS semantics itself.

This chapter presents these three stages divided into two algorithms, daemon membership in Section 6.1 containing stages one and two, and group membership in Section 6.4 containing stage three.

6.1 Daemon Membership Algorithm

The Membership component of Spread handles changes in the current set of reachable daemons. Spread provides detection of failures, partitions, and merges and will discover the currently reachable set of daemons. Spread forms a membership based on that set and reconfigures the overlay network. Spread transfers state from every daemon to every other daemon and, based on the transfered state information, recovers any missing messages so that all the daemons are consistent. Finally, Spread delivers any stored messages, creates a new application membership and delivers it to the Groups layer where the per-group membership algorithm executes.

Membership is complex because it requires both preserving a sophisticated set of semantic guarantees in addition to performing system level fault detection and network re-

configuration. The membership algorithm implemented in Spread uses several stages to simplify these operations. The membership algorithm can be expressed as a state-machine with 6 states representing 4 major steps: Gathering, Reconfiguration, State Transfer, and Cleanup.

6.1.1 States

The Membership system always runs in one of the following six states:

OP Active Operational state when no membership change is in progress.

SEG Initial state upon detection of membership change. During this state all members try to find others in their segment.

REP Another daemon has taken the role of representative of this segment. Daemons in this state wait for the representative to tell them the complete new membership.

GATHER A daemon in this state is the representative of a segment. The daemon will try to find all the reachable representatives of other segments and form a membership with them.

STATETRANS A membership has been formed. Each daemon multicasts their current state to the new set of daemons. Then, each daemon resends what they are required to.

EVS Each daemon receives resent data messages. When all missing messages have been received the membership protocol is completed by delivering all necessary messages and creating the transitional and final membership events. After doing this, each daemon shifts back to OP state.

6.1.2 Types of Messages

- ALIVE

- JOIN
- REFER
- NEWMEMB
- STATETRANS
- DATA

All messages begin with a type field which identifies what kind of message it is. In the following description and pseudo-code the message of a particular type will be represented by the name of the type in lowercase, typeset in italics. For example, ALIVE messages will be referenced as *alive*. And fields of a message will be referenced just as fields of a structure are in C, with a '.', for example, *alive.sender*.

A message of type ALIVE also has the following fields:

- *sender* – identifier of whichever server sent the message.

A message of type JOIN also has the fields:

- *seq* – sequence number of join message.
- *members* – set of proposed members.
- *reps* – set of representatives that are known. Each representative has an id, a type, and the segment in which the representative is located.

A message of type REFER also has the fields:

- *rep* – identifier of a representative.
- *type* – the type of representative being referred to.

A message of type NEWMEMB also has the fields:

- *stid* – membership id used for STATETRANS messages.
- *members* – set of new members.

A message of type STATETRANS also has the fields:

- *sender* – identifier of whichever server sent the message.
- *pviewid* – proposed view id.
- *lts* – highest lts seen by sender.
- *aru* – ARU of sender.
- *membid* – old membership id of sender.
- *H* – copy of senders H array.
- *committed* – set of members to whom the sender is committed.
- *messages* – set of messages the sender knows they missed from each site.

6.1.3 Possible Events

- Locally detect a link failed.
- Process Crash.
- Process Recovery.
- Receive a DATA message.
- Receive an ALIVE message.
- Receive a JOIN message.
- Receive a NEWMEMB message.
- Receive a STATETRANS message.
- Representative Timeout.
- Segment Timeout.
- Gather Timeout.
- Statetrans Timeout.
- Join Timeout — Just for sending.
- Alive Timeout — Just for sending.

In the pseudo-code that defines the protocols, the following conventions are followed to ease reading. All constants and function calls display in SMALL CAPS. All events, such as those listed above display in **`bold typewriter`** font. Variables are typeset in *italic*, message types in ALL CAPITALS and comments in normal text.

Each server maintains a number of state variables. All of the code is considered to have access to these variables. At the end of any code section, for example a function or the handler for an event, the state variables must be consistent and correct. However, they may be temporarily incorrect during execution of a code section.

The server state consists of:

State A value containing the current state of the protocol.

H A vector of sites storing the highest sequence and LTS values this server has seen from each site.

C A vector of sites storing the highest contiguous sequence number of messages this server has received that originated at each site.

ARU An integer value. All messages with LTS value less than this have been received by all servers.

AgreedLine An integer value. This server has received all messages with a LTS value less than this.

MesgList A list storing all messages received by the server. Messages are removed from the list and discarded when the message's LTS value is less then the value of ARU.

Memb A membership set. This consists of a set of servers organized into segments.

FMemb The future membership set that will become an actual membership if the membership protocol completes.

Memb_id The unique view identifier of the membership Memb.

FMemb_id	The unique view identifier of the membership FMemb.
ST_id	The temporary view id used to identify STATETRANS messages.
PReps	A set of potential representatives. The PReps vector only provides an optimization by keeping track of which other servers might currently be alive and good servers to contact to reconstruct a membership.
GReps	A set of representatives that represent all the members who will be part of the new membership.
GMemb	A set of members that represents each server's best guess as to who will be in the new membership.
Commit	Commit set.
CommitTrans	Commit Set transitional
FCommit	Future Commit set.
FCommitTrans	Future commit set transitional.
SMemb	Seg members.
StableNet	Stable network membership boolean
Active	A boolean value that is true of the membership is alive, false if a failure has occurred and the current membership is broken.
NumST	The number of Statetrans messages collected so far.
WaitLTS	A value indicating the target LTS value of the membership. When a server has recovered all messages it needs, it sets AgreedLine to WaitLTS. Then, when a server's ARU reaches WaitLTS it can install the new membership.
LH	A vector of sites storing the lowest of all of the H values sent in statetrans messages by members who were in the server's old membership This value is only used in the Statetrans state.

PH	A vector of sites storing the highest seq and LTS values seen by any of the members of the server's old membership. This value is only used in the Statetrans state.
PARU	A value storing the highest ARU of any of the statetrans messages received. This value is only used in the Statetrans state.
Cur_seg_rep	A value that tracks the identifier of the representative for the server's segment. This value is only used during Seg and Rep states.
MissingMessages	The set of messages that servers trying to form a new membership are missing. The received Statetrans messages establish this message set.

All sets return their size (number of elements) when the cardinality operator $|set|$ is applied to them.

A set of utility functions used throughout the code are given below:

- seg_rep() returns the smallest member of the segment, they will act as representative.
- seg()
- smallest() member of a membership
- smallest_rep() this returns the smallest rep of a set of reps. has lots of special cases.
- now() returns the current local time in seconds since Jan 1 midnight GMT 1970.

6.1.4 Start State

A process always begins execution in the start state. The only event possible in this state is *recover*. A recover event consists of initializing the membership set to contain only the local process and triggering a membership change by declaring a failure. The membership id of a recovery membership is the process id of the process and a timestamp of -1 which is considered older then all other possible membership ids.

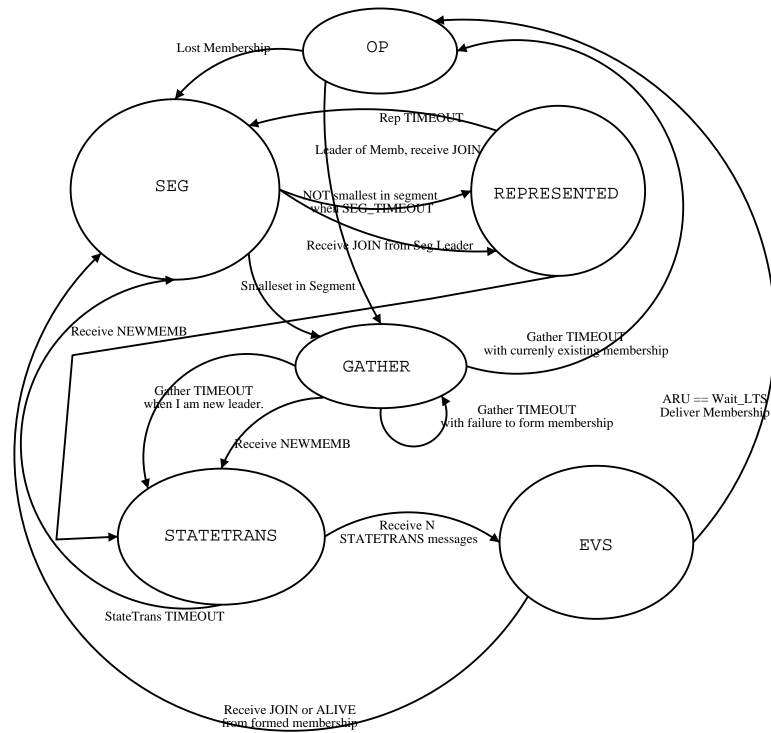

Figure 6.1: Membership State Machine

Start State

---

1 **switch** $(event)$
2 **`recover`** :
3 $\quad Memb \Leftarrow$ SELF
4 $\quad CommitTrans \Leftarrow$ SELF
5 $\quad SiteIDtoSeg[$SELF$] \Leftarrow 0$
6 $\quad Memb_id \Leftarrow \{$SELF$, -1\}$
7 $\quad$ DECLAREMEMBLOSS()

Figure 6.2: Daemon Membership – Start State

6.1.5 OP State

Spread has several ways to detect other servers that have either started up or previously were partitioned away. First, when a server starts up it announces itself to both the local network with broadcasts and to a subset of the potential other sites with unicast messages. If any existing server hears these announcements they will initiate a merge with this new server. Second, the representative of an existing set of servers will probe for other servers every 90 seconds or so (the time is configurable) by sending unicast messages to all sites at which the representative does not have existing members.

6.1.6 Segment State

While in Seg state an ALIVE message is sent every Alive_timeout seconds. The Seg state performs a search for any other servers that are alive and reachable in the local site. To start the search for other servers this daemon sends an ALIVE message every Alive_timeout seconds and sets a timeout of `Seg_timeout` after which it will switch to either Gather or Represented state.

The server will switch to Gather state in three cases. First, if the server is alone in the site it will not receive any ALIVE messages from any other servers and will switch to Gather. Second, even if there are other servers running on the site and they are reachable. They will ignore the ALIVE message sent by the sender if they already belong to an active membership. Therefore, the sender will switch to Gather. Third, if the server does receive ALIVE messages from other servers but the other servers have larger ids, then this server will become the representative of the set of alive servers and will switch to Gather state.

The reason servers do not join an existing segment of alive servers is because the membership protocol treats all the members of a segment as one set who came together. This does not require that all servers on a network segment always come from the same membership. That is clearly impossible. What will happen is that the sets of servers will merge during the Gather state through the exchange of Join messages. The Alive messages are

OP State

```
1  switch (event)
2  linkfailed :
3      Commit ⇐ SELF
4      for each seg ∈ Conf
5          if |Memb[seg]| = 0 then
6              PReps ⇐ PReps ∪ SEG_REP(Conf, seg)
7          else PReps ⇐ PReps ∪ SEG_REP(Memb, seg)
8      DECLAREMEMBLOSS()
9  ureceive(s, ALIVE) :
10     if s ∈ Memb then
11         DECLAREMEMBLOSS()
12         SM ⇐ SMemb ∪ s
13 ureceive(s, JOIN) :
14     if s ∈ Memb then
15         DECLAREMEMBLOSS()
16     else if SMALLEST(Memb) = SELF then
17             PReps ⇐ NIL
18             PReps ⇐ s
19             SMemb ⇐ Memb
20             GReps ⇐ SELF
21             GReps.type = NET_REP
22             GMemb ⇐ SMemb
23             State ⇐ GATHER
24             SENDJOIN()
25         else if (s ∉ SEG(SELF) or (SEG_REP(Memb, SEG(SELF)) = SELF) then
26                 mcast(s, refer2)
27 ureceive(s, NEWMEMB) :
28     PROCESSNEWMEMB()
29     SENDSTATETRANS()
30 nreceive(s, DATA) :
31     Normal Data message handling
32 ureceive(s, REFER), nreceive(s, STATETRANS),
33 Rep Timeout, Segment Timeout, Gather Timeout, Statetrans Timeout,
34 Join Timeout, Alive Timeout :
35     impossible
```

Figure 6.3: Daemon Membership – OP State

ProcessNewMemb()

1 **if** $newmemb.stid <= FMemb_id$ **then**
2 **ignore**
3 **if** $State =$ GATHER **then**
4 **for each** m **in** $SMemb$
5 **mcast**$(m, newmemb)$
6 $ST_id \Leftarrow newmemb.stid$
7 $StableNet \Leftarrow$ FALSE
8 $FMemb \Leftarrow newmemb.members$
9 $PrevSitesToSiteID \Leftarrow SitesToSiteID$
10 INITIALIZE$(SitesToSiteID, SiteIDToSeg, FMemb)$
11 $Active \Leftarrow$ TRUE
12 $NumST \Leftarrow 0$
13 $WaitLTS \Leftarrow 0$
14 $PARU \Leftarrow 0$
15 $PH \Leftarrow \{0\}$
16 $LH \Leftarrow \{MAXINT\}$
17 $FCommit \Leftarrow$ NIL
18 $FCommitTrans \Leftarrow$ NIL
19 $State \Leftarrow$ STATETRANS
20 **ninit**$(FMemb)$
21 Force newmemb as first message on all links

Figure 6.4: Daemon Membership – ProcessNewMemb function

DeclareMembLoss()

1 $SMemb \Leftarrow$ SELF
2 **mcast**(SEG(SELF), $alive$)
3 $StableNet \Leftarrow$ FALSE
4 $Active \Leftarrow$ FALSE
5 $Memb \Leftarrow$ SELF
6 $SiteIDtoSeg$[SELF] $\Leftarrow 0$
7 $Cur_seg_rep \Leftarrow$ NIL
8 $State \Leftarrow$ SEG

Figure 6.5: Daemon Membership – DeclareMembLoss function

SendJoin()

```
1  join.members ⇐ SMemb
2  join.reps ⇐ GReps
3  if ¬Active then
4    mcast(SEG(SELF), join)
5  for each r in PReps
6    mcast(r, join)
```

Figure 6.6: Daemon Membership – SendJoin function

Seg State

```
 1  switch (event)
 2  linkfailed :
 3      impossible
 4  ureceive(s, ALIVE) :
 5      SMemb ⇐ SMemb ∪ s
 6  ureceive(s, JOIN) :
 7      if SEG(s) = SEG(SELF) and join.reps[0].type = SEG_REP then
 8         Cur_seg_rep ⇐ s
 9         State ⇐ REP
10         if SELF ∉ join.members then
11            mcast(SEG(SELF), alive)
12      else mcast(s, refer)
13  Segment Timeout :
14      Cur_seg_rep ⇐ NIL
15      if smallest(SMemb) = SELF then
16         GReps ⇐ SELF
17         GReps.type ⇐ SEG_REP
18         GMemb ⇐ SMemb
19         State ⇐ GATHER
20         SENDJOIN()
21      else State ⇐ REP
22  Alive Timeout :
23      mcast({SMemb}, alive)
24  ureceive(s, REFER), ureceive(s, NEWMEMB),
25  nreceive(s, STATETRANS), nreceive(s, DATA) :
26      ignore
27  Gather Timeout, Statetrans Timeout,
28  Join Timeout, Rep Timeout :
29      impossible
```

Figure 6.7: Daemon Membership – Seg State

meant to locally gather alive servers so to make the wide-area exchange of Join messages more efficient.

The server transitions to Represented state if it has heard an Alive message from some other server who has a smaller id.

6.1.7 Represented State

Servers in the Represented state do not send any messages. The purpose of the Represented state is to increase the scalability of the membership algorithm by suppressing all servers in a site except one from sending wide-area messages between sites. A server in Represented state will receive JOIN messages from their representative which indicate that the representative is still alive and running. If they fail to receive these JOIN messages for more then Rep_timeout seconds, they shift to Seg state and resend ALIVE messages to refind who is alive. If they receive a NEWMEMB message then they process it normally and try to construct that membership. They then shift to Statetrans state.

6.1.8 Gather State

While in Gather state a JOIN message is sent every Join_timeout seconds. The JOIN message is sent to the local site and to all the servers listed in the PReps set.

If a daemon in the active membership exists on the same segment as the new daemon, the any existing daemon will hear the new daemon's join message and will respond if it is the segment representative or the network representative. If the daemon is the network representative it will initiate their own membership change by calling `Shift_to_Gather`, and if they are a segment representative they will send the new daemon a REFER message identifying the network representative. Only the network representative will actually start a membership change in the current membership. Even if a daemon answers by sending a referral, the daemon will continue to work in the old membership, i.e. forget that it knows someone new has tried to join, until the network representative of the current membership

Rep State

```
 1  switch (event)
 2  linkfailed :
 3      impossible
 4  ureceive(s, ALIVE) :
 5      ignore
 6  ureceive(s, JOIN) :
 7      if SEG(s) = SEG(SELF) and join.reps[0].type = SEG_REP then
 8         if s ≠ Cur_seg_rep then
 9            Cur_seg_rep ⇐ s
10            reset Rep Timeout
11         if SELF ∉ join.members then
12            mcast(SEG(SELF), alive)
13      else mcast(s, refer)
14  ureceive(s, NEWMEMB) :
15      PROCESSNEWMEMB()
16      SENDSTATETRANS()
17  Rep Timeout :
18      SMemb ⇐ SELF
19      mcastSEG(SELF), alive
20      Cur_seg_rep ⇐ NIL
21      State ⇐ SEG
22  ureceive(s, REFER), nreceive(s, STATETRANS), nreceive(s, DATA) :
23      ignore
24  Segment Timeout, Gather Timeout, Statetrans Timeout,
25  Join Timeout, Alive Timeout :
26      impossible
```

Figure 6.8: Daemon Membership – Rep State

is contacted by the new daemon.

If any of these JOIN messages reach any of the currently running daemons, the new daemon will be informed of who the current network representative is [1]. Since that network representative will also enter Gather state and send JOIN messages, the new daemon will receive a JOIN and call `Memb_handle_join`. This function adds the representatives listed in the join message to the `Potential_Reps` list and the members listed in the join to the `Form_Members` list. The actual sender of the Join is added to the `Form_Reps` list. If the daemon who sent the join is the network representative then the `Form_or_fail` call is reset to `Gather_timeout` time in the future.

A server will leave this state after `Gather_timeout` seconds. When the timeout expires the server will decide which of four next steps will be taken.

- If no new servers have been found and a current active membership exists, the server shifts to OP state and continues normal execution. This is the case when the network representative was probing for new servers and did not find any.
- If this server is the network representative and the set of GReps is larger then 1, new servers have been found and a new membership should be formed. This server sends a NEWMEMB message and shifts to Statetrans state.
- If this server is not the network representative, but its current membership is still marked active. It then shifts to OP and continues normal operation.
- If this server is not the network representative, and no membership is active, then the attempt to form a membership has failed, possibly because after sending a JOIN message the smallest id server crashed, so restart the Gather state with a new round of JOIN messages.

[1] It is possible because of message loss for the new member to not discover the existing daemons. In this case the new daemon will form a membership by themselves and try to discover the others in its usual way by occasional calls to `Lookup_new_members`.

Gather State

```
1  switch (event)
2  linkfailed :
3      impossible
4  ureceive(s, ALIVE) :
5      if ¬Token_alive then
6         GMemb ⇐ GMemb ∪ s
7      else if s ∈ Memb then
8              DECLAREMEMBLOSS()
9              GMemb ⇐ GMemb ∪ s
10 ureceive(s, JOIN) :
11     GReps ⇐ GReps ∪ join.reps[0]
12     for each r in join.reps
13        PReps ⇐ PReps ∪ r
14     for each m in join.members
15        GMemb ⇐ GMemb ∪ m
16     if join.reps[0] = SMALLESTREP(GReps) then
17         reset gather timeout
18 ureceive(s, REFER) :
19     PReps ⇐ PReps ∪ refer.rep
20 ureceive(s, NEWMEMB) :
21     PROCESSNEWMEMB()
22     SENDSTATETRANS()
23 Gather Timeout :
24     if SMALLESTREP(GReps) = SELF then
25        if Active and |GReps| = 1 then
26           State ⇐ OP
27           DELIVER_MESS()
28        else CREATE_SEND_NEWMEMB()
29     else if Active then
30             State ⇐ OP
31             DELIVER_MESS()
32          else GMemb ⇐ SELF
33                PReps ⇐ GReps
34                State ⇐ GATHER
35                GReps ⇐ SELF
36                GReps.type = SEG_REP
37                GMemb ⇐ SMemb
38                SENDJOIN()
39 nreceive(s, STATETRANS), nreceive(s, DATA) :
40     ignore
```

Figure 6.9: Daemon Membership – Gather State

Gather State (Part 2)

```
1  switch (event)
2  Join Timeout :
3       SENDJOIN()
4  Rep Timeout, Segment Timeout, Statetrans Timeout, Alive Timeout :
5       impossible
```

Figure 6.10: Daemon Membership – Gather State (Part 2)

Create_Send_NewMemb()

```
1  PReps ⇐ GReps
2  SORT(GMemb)
3  newmemb.stid ⇐ {Self, NOW()}
4  newmemb.members ⇐ GMemb
5  for each r in GReps
6    mcast(r, newmemb)
7  for each m in SMemb
8    mcast(m, newmemb)
9  PROCESSNEWMEMB()
```

Figure 6.11: Daemon Membership – Create_Send_NewMemb function

6.1.9 Statetrans State

The Statetrans state begins when a process receives a NEWMEMB message and creates a new network level configuration. The Statetrans state begins by creating a new STATETRANS message and sending it to all of the other servers who are in the new membership through the new overlay network that was just established. The STATETRANS messages are sent in the new "proposed" membership defined by FMemb. Although Spread usually rejects messages which arrive with a membership id that is not equal to the current membership, Spread will accept STATETRANS messages that arrive with a memb_id equal to the FMemb_id.

The Statetrans state is the first state in the membership when the overlay network operates. All of the necessary point-to-point and broadcast link protocols are active. Thus, if a ring exists then the token is circulating. If a hop exists then it is sending acks. If a TCP link exists then it has connected. Also, Spread sends ARU_Update messages. ARU_Update

SendStateTrans()

1 $statetrans.lts \Leftarrow H[Self]$
2 $statetrans.aru \Leftarrow ARU$
3 $statetrans.pviewid \Leftarrow \{Self, \text{NOW}()\}$
4 $statetrans.membid \Leftarrow Memb_id$
5 $statetrans.H \Leftarrow H$
6 $statetrans.committed \Leftarrow Self \cup Commit$
7 $statetrans.messages \Leftarrow missingmessages$
8 **rmcast**$(statetrans)$

Figure 6.12: Daemon Membership – SendStateTrans function

messages play a critical role in completing the EVS state but do not play any role in the actual transmission of the STATETRANS messages.

When a STATETRANS message arrives at a server the server verifies that the message was sent by a member of this server's prior membership. This can be easily checked as each STATETRANS message has the membership id of its sender's previous membership stored in the message. If the message does belong to a different prior membership then the message's contents are ignored and only the fact that it was received is recorded by the NumST counter. When all the STATETRANS messages expected by a server have arrived (one per daemon in the new membership) the server uses the stored state of those messages to calculate what messages were missed at the end of the previous membership and need to be resent. The server also sets up the protocol counters WaitLTS, ARU, and H, and shifts to EVS state.

6.1.10 EVS state

The EVS state consists to two distinct communication phases.

First, all the messages that need to be resent are sent by one of the servers who has them. The resent messages use the dissemination network set up just prior to the Statetrans state. Each server compiles a list of which messages it is responsible for resending and sends them. During this state the servers will also receive messages resent by other servers.

Statetrans State

```
1  switch (event)
2  linkfailed :
3      DeclareMembLoss()
4  ureceive(s, ALIVE) :
5      if s ∈ FMemb then
6          DeclareMembLoss()
7          GMemb ⇐ GMemb ∪ s
8  ureceive(s, JOIN) :
9      if s ∈ FMemb then
10         mcast(s, newmemb)
11 nreceive(s, STATETRANS) :
12     if statetrans.stid ≠ ST_id then
13         ignore
14     if statetrans.sender = SMALLEST(FMemb) then
15         FMemb_id ⇐ statetrans.pviewid
16     if statetrans.lts > WaitLTS then
17         WaitLTS ⇐ statetrans.lts;
18     if statetrans.membid = Memb_id then
19         FCommitTrans ⇐ FCommitTrans ∪ s
20         if statetrans.aru > PARU then
21             PARU ⇐ statetrans.aru
22         for each l in Sites
23             PH[l] ⇐ MAX(PH[l], statetrans.H[l])
24         for each l in Sites
25             LH[l] ⇐ MIN(LH[l], statetrans.H[l])
26         FCommit ⇐ FCommit ∪ statetrans.committed
27         MissingMessages ⇐ MissingMessages ∪ statetrans.messages
28         NumST ⇐ NumST + 1
29     if NumST = |FMemb| then
30         for each l in Sites
31             if LH[l] < PH[l] and PH[l] = H[l] then
32                 MissingMessages ⇐ MissingMessages ∪ {LH[l] + 1, ..., PH[l]}
33         WaitLTS ⇐ WaitLTS + 1000
34         ARU ⇐ PARU
35         Deliver_Safe_Mess()
36         H ⇐ PH
37         for each m in MissingMessages
38             if m.num_missed = |FCommitTrans| then
39                 INSERT_DUMMY_MESSAGE(m)
40         ResendMissingMessages()
41         State ⇐ EVS
42         if C[l].seq = H[l].seq∀ l ∈ PrevSites then
43             AgreedLine ⇐ WaitLTS
44             Commit ⇐ FCommit
45             CommitTrans ⇐ FCommitTrans
```

Figure 6.13: Daemon Membership – Statetrans State

Statetrans State (Part 2)

```
1  switch (event)
2  ureceive(s, REFER), ureceive(s, NEWMEMB), nreceive(s, DATA) :
3      ignore
4  Statetrans Timeout :
5      DeclareMembLoss()
6  Rep Timeout, Segment Timeout, Gather Timeout,
7  Join Timeout, Alive Timeout :
8      impossible
```

Figure 6.14: Daemon Membership – Statetrans State (Part 2)

EVS State

```
 1  switch (event)
 2  linkfailed :
 3      DeclareMembLoss()
 4  ureceive(s, ALIVE) :
 5      if s ∈ FMemb then
 6          DeclareMembLoss()
 7          GMemb ⇐ GMemb ∪ s
 8  ureceive(s, JOIN) :
 9      if s ∈ FMemb then
10          DeclareMembLoss()
11          GMemb ⇐ GMemb ∪ s
12  ureceive(s, REFER), ureceive(s, NEWMEMB), nreceive(s, STATETRANS) :
13      ignore
14  nreceive(s, DATA) :
15       Normal Data message handling
16       Except – allow ARU_Update messages from unknown memberships
17      if data.membid == FMemb_id
18          and data.type ∈ {Frag, Meta, ARU_Update} then
19          DeliverMembership()
20      if C[l].seq = H[l].seq∀l ∈ PrevSites then
21          AgreedLine ⇐ WaitLTS
22          Commit ⇐ FCommit
23          CommitTrans ⇐ FCommitTrans
24      if ARU ≥ WaitLTS then
25          DeliverMembership()
26  Rep Timeout, Segment Timeout, Gather Timeout,
27  Statetrans Timeout, Join Timeout, Alive Timeout :
28      impossible
```

Figure 6.15: Daemon Membership – EVS State

DeliverMembership()

1 $TransMembership \Leftarrow CommitTrans$
2 $CommitMembership \Leftarrow Commit$
3 **stranssig**$(Self, TransMembership)$
4 $entry \Leftarrow MesgList$
5 **while** $entry \neq$ NIL **and** $entry.lts < ARU$
6 **do**
7   **if** $entry.type =$ DUMMY **then**
8         $hole_found \Leftarrow$ TRUE
9   **if** $hole_found =$ FALSE **or** $entry.sender \in CommitMembership$ **then**
10   **if** $entry.delivered =$ FALSE **then**
11   **sdeliver**$(entry.msg)$
12             $entry.delivered \Leftarrow$ TRUE
13   UPDATE_FIFO$(entry.msg)$
14   DISCARD$(entry.msg)$
15     $entry \Leftarrow entry.next$
16 $Memb \Leftarrow FMemb$
17 $Memb_id \Leftarrow FMemb_id$
18 $StableNet \Leftarrow$ TRUE
19 $H \Leftarrow \{0\}$
20 $C \Leftarrow \{0\}$
21 $Highest_ARU \Leftarrow 0$
22 $AgreedLine \Leftarrow 0$
23 $ARU \Leftarrow 0$
24 $Session_Seq \Leftarrow 0$
25 CLEAR_FIFO_DELIVERABLE()
26 $State \Leftarrow$ OP
27 **sview**$(Self, Memb_id, Memb, TransMembership)$
28 DELIVER_MESS()

Figure 6.16: Daemon Membership – DeliverMembership function

The server resends its messages with the memb_id of the membership they were originally sent in. Thus, the only servers that will process the data messages when they arrive are those servers who came from the same prior membership as the sender. These messages are essentially the cleanup of ‘what should have happened’ in the old membership. Servers who were never in that old membership need have no knowledge of them.

Second, when the server receives all the messages it requires, which it will know because the C vector for all sites in the old membership will reach the same values as the H vector, the server will update AgreedLine to be equal to WaitLTS. The server will commit to the current set of members by setting Commit equal to FCommit and CommitTrans equal to FCommitTrans. The update of AgreedLine will allow the site wide ARU entry in HA to be updated once all members of the site have received everything they are waiting for. When the site wide ARU is updated the site representative will send updated ARU_Update messages to inform all of the sites of the new ARU for this site. When a server receives ARU_Updates from all other sites indicating that they have finished receiving messages the ARU (at this server) will reach the WaitLTS value. At that point the server knows EVS state is completed and it can deliver the messages and membership view.

It is possible for the server to receive data messages sent in the new membership before installing the membership itself. This is possible if one server receives the ARU_Updates required, installs the new membership, and begins sending before another server has received all the ARU_Updates. This is legal because the connections between servers are NOT FIFO and are only “eventually reliable” so a server might get messages out of order from the links. If a server receives such a message, they can treat the message as an indication that the ARU has reached WaitLTS and can install the new membership. They can do this because all servers have completed EVS. The servers just do not yet know that they are all finished. Therefore, if one server tells another that they have finished, the second server knows that all the servers have finished.

The actual delivery of a membership has several distinct steps.

1. Deliver a transitional membership.
2. Deliver all messages the server has which meet the appropriate guarantees.
3. Deliver a regular membership.
4. Reset all protocol level counters and variables for a new membership.
5. Deliver any new messages that are ready.

As a result of the delivery of a transitional and regular membership the upper layers of Spread, specifically the groups layer, will initiate protocols to deliver correct process level membership events. The group membership protocol will send messages in the new server membership to synchronize the process-level group state and then deliver actual membership events to the applications. The server membership event is not directly delivered to any process except the server itself.

During recovery from a membership change it is possible for a server to know of messages which they do not have and thus cannot insert into the main Message Lists. These messages are dependent on other messages and can only be inserted when the messages upon which they depend arrive. When the server receive those other messages then it can immediately insert the dependent messages.

One of the key ideas in this membership algorithm is the Commit_Set (Commit). This set always increases during a membership process. It is only reset to the local server when it first enters a membership change from the OP state. The set grows when Commit is set equal to Future_Commit_Set(FCommit) during the EVS state. The server sets the Commit set to equal FCommit when we have received all the messages sent by members of the FCommit. The idea is after that point the server is committed to delivering those members messages in whatever membership ends up forming, even if according to the server some of the other servers fail before the membership completes (i.e. after we commit, but before we know everyone else is done and deliver the membership).

If a cascading membership occurs, each server adds anyone in their Commit to their Statetrans message. Every server who receives the statetrans message will add the members to their FCommit.

6.2 Proof of Daemon Membership

LEMMA 6.1 ()
View ids are unique up to the granularity of the timestamp.

The granularity of timestamps in Spread is seconds. Therefore, if more then one membership change occurs within a second, both membership views can have identical view ids even though they are actually different views. This engineering limitation can be avoided by using a finer granularity timestamp clock.

View identifiers can be compared by using the timestamp portion as a primary key and the IP address as a secondary key.

LEMMA 6.2 ()
Every membership execution results in a transsig event prior to the corresponding view event.

PROOF: The only place a **`view`** event is delivered is in Figure 6.16 and it always delivers a **`stranssig`** event with the same view id first. □

LEMMA 6.3 ()
SMemb always contains Self.

PROOF: SMemb is either initialized to Self (in several places) or initialized to Memb (which must include Self by induction on the sequence of views. The initial case being the initial view which always includes Self by definition, and each later view) □

LEMMA 6.4 ()
The sender of a Join message is always listed in the member list of the Join.

PROOF: The member list is always identical to SMemb, as shown in Figure 6.6. By Lemma 6.3 SMemb always contains Self, so the sender of the join is listed as a member. □

LEMMA 6.5 ()
Every server that is in GReps is also in GMemb.

PROOF: New members are added to GReps in two ways. First, when GReps is initialized to Self, at which point GMemb is initialized to SMemb. By Lemma 6.3, SMemb always contains Self. Second, when GReps is updated by a join message, the sender of the join is added to GReps. At that point all of the members listed in the join message are also added to GMemb. By Lemma 6.4, the sender of a join message is always listed in the members list of the message. Thus, every member in GReps will also be in GMemb. □

LEMMA 6.6 ()
For every installed view v, containing vid_v and membership M_v, the server received a newmemb message with newmemb.members = M_v and a statetrans message from `smallest(`M_v`)` *with statetrans.pviewid = vid_v.*

Essentially, every installed view corresponds exactly with the members list of a newmemb message received previously by the server and the viewid proposed by the smallest member of that set.

PROOF: The only place in the algorithm where the FMemb set is assigned is when a newmemb message is received. The FMemb set is never changed again until either a new newmemb message arrives or the membership is declared and FMemb is assigned to Memb. Therefore, the M_v set is always the set to the last newmemb members list when the membership is declared.

The FMemb_id which will become the vid_v of the final view is set in the algorithm, as shown in Figure 6.13, when a statetrans message is received from the smallest member of the FMemb set. The statetrans message will only be handled if it was sent in response

to the same newmemb message that initiated the FMemb set because of the check at the beginning of the hander for statetrans messages in Figure 6.13. □

LEMMA 6.7 ()
If a server installs a view v, the transitional set of v is a subset of M_v.

PROOF: We prove that every member of the transitional set of v is also in M_v. By Lemma 6.6 every view corresponds to the most recent preceeding NEWMEMB message.

The transitional set of v is equal to the FCommitTrans set when all messages that need to be recovered in EVS state have been received ($C[i] == H[i] \forall i \in PrevSites$). FCommitTrans is initialized to empty when a NEWMEMB message is received and is only increased when the STATETRANS messages are received. When a process p receives a STATETRANS messages from another process q, if prev_membq == prev_membp then q is added to FCommitTrans. Because STATETRANS messages are rejected unless they are from someone who is in FMemb, q is also in FMemb. So q will be in M_v when v is delivered. □

LEMMA 6.8 ()
If a server installs a view v' in a view v, the transitional set of v' is a subset of M_v.

PROOF: We prove that every member of the transitional set of v' is also in M_v. By Lemma 6.6 every view corresponds to the most recent preceeding NEWMEMB message.

The transitional set of v is equal to the FCommitTrans set when all messages that need to be recovered in EVS state have been received ($C[i] == H[i] \forall i \in PrevSites$). FCommitTrans is initialized to empty when a NEWMEMB message is received and is only increased when the STATETRANS messages are received. When a process p receives a STATETRANS messages from another process q, if prev_membq == prev_membp then q is added to FCommitTrans. Therefore, the only processes that are added to FCommitTrans are ones who have the same previous membership and so were in M_v. □

THEOREM 6.1 (INITIAL VIEW EVENT)
In all traces of the EVS-S automaton, every send, deliver, and transsig event occurs in some view.

PROOF: The Server Execution Structure assumption requires that a recover event is the first event that occurs in a trace. The recover event, shown in Figure 6.2, causes an initial view to be established containing only the process itself. All events occur after that initial view.

After a crash event, no messages are delivered. □

THEOREM 6.2 (SELF INCLUSION)
All view events include Self in the set of members.

PROOF: By Lemma 6.3 SMemb always includes Self. GMemb is always initialized to SMemb and members are only added to it, never removed. Thus, GMemb always contains Self.

From the algorithm, FMemb is determined by the set of members in the newmemb message. The members set in the newmemb message is always a copy of the GMemb set held by whoever sends the newmemb message. The thing to prove is that the GMemb of that member always includes everyone who will act on the newmemb. This might require a check upon receiving a newmemb to verify that we are in it. The installed Memb is always a copy of the current FMemb. Nowhere in the algorithm is anyone removed from the membership. So Self is always included in the membership. □

THEOREM 6.3 (MEMBERSHIP AGREEMENT)
If server s installs a view with identifier id and server t installs a view with the same identifier, then the membership sets of the views are identical.

PROOF: The identifier id of a view is set when the statetrans message from the representative of the new membership is created. The id is composed of the IP address of the first member and the current time (in seconds) at that member and is stored in the pviewid field of the statetrans message. A process only sends a statetrans message in response to receiving a newmemb message, and the statetrans message has a stid identical to the stid received

from the newmemb message. So a unique view id is generated in response to a particular membership set specified in the newmemb message. Thus, any two servers who install a view with id received the same statetrans message from server id.address and that statetrans message stid field matched the stid field of the newmemb message with membership set $M_{newmemb}$. FMemb is set to $M_{newmemb}$ by receiving a newmemb message and is never changed. The installed membership set is identical to FMemb at the time it is installed. So both servers will have the same membership set in view with identifier id.

Lemma 6.6 provides that every view corresponds to a newmemb message and a statetrans message. □

THEOREM 6.4 (LOCAL MONOTONICITY)
If server s installs a view with identifier id' after installing a view with identifier id, then id' is greater then id.

PROOF: Under an assumption of loosely synchronized clocks, the view ids proposed by each server in its statetrans messages will all have a greater timestamp then any previous proposed view ids. Thus, since every installed view id is identical to one of the proposed view ids in a statetrans message, the installed view ids will increase monotonically.

If one does not want to assume synchronized clocks, the existing algorithm can be made monotonic by treating the timestamp field of the view id as a Lamport time stamp instead of a real clock time. The Lamport time stamp can be initialized by the real clock time to preserve monotonicity over restarts of the entire system (asuming the real clocks did not run backwards). By using the Lamport virtual time stamps each server will never propose a view id that is less then any view id it has seen before. The final view id will use the timestamp that is greatest of all of the proposed time stamps in the statetrans messages. □

THEOREM 6.5 (TRANSITIONAL SET)

(a). The transitional set for the first view installed at a server following a recover event consists of just the server itself.

(b). *If a server s installs a view v' in a previous view v, then the transitional set for v' at s is a subset of the intersection of M_v and $M_{v'}$.*

(c). *If servers s and t install the same view, then t is included in s's transitional set for that view if and only if s's previous view was identical to t's previous view.*

(d). *If servers s and t install the same view v' in the same previous view v, then they have the same transitional sets in v'.*

PROOF:

Property a holds because the transitional set is initialized to Self when the daemon recovers as shown in Figure 6.2. This simplifies the semantics because groups can rely on the fact that itself is in trans-memb.

Property b can be proved by showing that all of the members of the transitional set for v' are in both M_v and $M_{v'}$, because by the definition of intersection, the transitional set is a subset of the intersection of M_v and $M_{v'}$.

All members of the transitional set for v' are in $M_{v'}$ is a direct result of Lemma 6.7. All members of the transitional set for v' are in M_v is a result of Lemma 6.8.

Property c This can be proven by establishing the following two claims, the "if" case and "only if" case.

If s's previous view was identical to t's previous view and s and t install the same view v then t is included in s's transitional set for v.

We are given that $v' = vid(v_s) = vid(v_t), \wedge v_s.viewid = v_t.viewid \wedge M_{v_s} = M_{v_t}$ and by Self Inclusion $t \in M_{v_s}$ **and** $t \in M_{v'_s}$. By the algorithm CommitTrans at s contains all servers whose previous view was the same as s's previous view and whom s received a STATETRANS message from. Since t is in M_{v_s} then s must have received a STATETRANS message from it and since $vid(v_s) = vid(v_t)$ s will have added t to it's FCommitTrans set. Thus $t \in T_{v_s}$.

If s's previous view was not identical to t's previous view and s and t install the same view, then t is not included in s's transitional set.

The only way for a server t to be in the CommitTrans set of another server s is for s to have received a STATETRANS message from t, and for the previous view id of t to equal the previous view id of s. But because of Lemma 6.1 and the condition that s's previous view was not identical to t's previous view, the previous view of s will not be the same view id as the previous view seen by t. So s will not include t in FCommitTrans and since CommitTrans is equal to FCommitTrans when the server leaves EVS state. The transitional set of s for view v cannot include t.

Property d As described above, the transitional set at any server s is exactly equal to the set of servers who sent a STATETRANS message and whose previous view was the same as s's previous view. Since s and t both installed v', they both must have received STATETRANS messages from all of the members of v'. Since every server only sends one STATETRANS message for each FMemb_id they must have received the same set of STATETRANS messages. Since s and t have the same previous view, they will have added the same subset of STATETRANS messages senders to their FCommitTrans set. And thus they will have the same transitional sets in v' when it is delivered. □

THEOREM 6.6 (NO DUPLICATION)
A server never delivers a message more then once.

PROOF: Figure 5.1 shows the process of delivering all messages. In every case a message is marked delivered and if a message is so marked, it is not delivered again. When messages are retransmitted during the EVS state, they maintain the unique identifiers of site_seq, session_seq and session_id and no message with a lts value less then ARU is retransmitted. A message is not discarded and removed from the list of active messages until the global ARU has increased higher then the lts of the message. Thus, a message is never delivered more then once. □

THEOREM 6.7 (DELIVERY INTEGRITY)
A deliver event in a view is the result of a preceding send event.

PROOF: Every deliver event requires the receipt of a DATA message either from the network, or from a local **`send`** event. Every DATA message received from the network originated at some process since we assume the network does not spontaneously create messages. The only way a DATA message is created is by a **`ssend`** input event. □

THEOREM 6.8 (SELF DELIVERY)
If a server s sends a message m, then s delivers m unless it crashes.

PROOF: When a server sends a message it also stores the message in the list of ordered messages. The only situation when a server will not deliver a message that it has stored in the message list is in the `DeliverMembership` function where messages that were sent by servers not in the Commit set and which are ordered after a hole in the message sequence are discarded without being delivered. Since the Commit set of a server always includes the server itself, the server will never discard messages that it sent. The messages will be eventually delivered. The only way delivery can be delayed is if necessary information from other daemons has not been received and if the lack of messages from some other server continues then a membership failure will be declared. When the membership is reformed the message will be deliverable because the membership will only include those members who are able to communicate with this server. □

THEOREM 6.9 (SAME VIEW DELIVERY)
If servers s and t both deliver a message m, then they both deliver m in the same view.

PROOF: For all reliable messages, if the message was sent by a server and delivered in view v, then if any other server received that message it will also deliver it in view v and not a later view, because in the Deliver_Membership function shown in Figure 6.16 all messages that are known about are either delivered or discarded. If the message was not received by some server then either the message will be recovered during the EVS state because at

least one server has the message or it will not be delivered because no one has the message. Because the only messages that are recovered and delivered during EVS are those from other servers who were in the same previous view, the membid of the statetrans message and recovered data messages is checked against our previous view, it is not possible to receive a recovered message that the sending server also provided to a different server in a different view. □

THEOREM 6.10 (VIRTUAL SYNCHRONY)
If servers s and t are virtually synchronous in a view then any message delivered by s in that view is also delivered by t.

PROOF: By the Transitional Set property 6.5 both s and t will have the same transitional set since they are virtually synchronous.

Take message m that is delivered by s. Since s delivered m, it must have the message and message m has some location in the global order of messages (even if it is only a reliable or FIFO message it still has a global order as well, it just can be delivered earlier then that order would require). At some point in the stream of events, a fault or join is detected and the membership algorithm begins. At that point if t has also delivered m then we are done. So let t not have delivered m yet. Then when t sends it's STATETRANS message it will either have an H[] field lower then the value for m (meaning it has not received m or any message from that site with a higher lts) or m will be listed in the set of missing messages. Because s and t deliver the same view, they must have received a STATETRANS message from each other, as otherwise they would not have completed the StateTrans state since the test for NumST would have failed. Based on the STATETRANS message s (or someone else) will have resent m because either it was in the list of requested messages or someone else knew of a higher LTS valued message and so resends those that it only knows about. Since the WaitLTS will be set to the highest known by anyone, the termination case that ARU = WaitLTS can only be reached once all of the servers in the new view have received all of the messages. Since t will be one of the servers in the new

view, it must also have received m. If s already delivered m prior to any failures, then t will deliver m in the Deliver_Membership function because no holes will exist in the set of ordered messages since s was able to deliver it. If s has not delivered it before, so a hole may exist, then either both s and t will deliver it or neither will, since they will have the same set of recovered messages and will have the same Commit set. So in any case if s delivered the message, then t will also deliver it. □

THEOREM 6.11 (SANE VIEW DELIVERY)

(a). A message is not delivered in a view earlier than the one in which it was sent.

(b). If a process p sends a message m, crashes and later recovers in a view v and a process q delivers m, then m is delivered in a view before v.

PROOF: The first part is a direct consequence of Delivery Integrity, Theorem 6.7, Self Delivery, Theorem 6.8, Same View Delivery, and Theorem 6.9, because all servers who deliver a message deliver it in the same view as the server who originally sent it and they must deliver it after the preceding send event.

The second part is a consequence of the message recovery that is performed in the EVS state. When p recovers, even if it recovers prior to the others noticing that it did crash it will know that it crashed and has zero state. So when the server participates in a membership instance, the other servers will exchange statetrans messages about the messages they have, including p's. Those messages are guaranteed to be delivered in the SAFE delivery of all message that occurs in the Deliver_Membership function in Figure 6.16. Therefore, m will be delivered before the view v that includes the newly recovered p. □

Proofs of the FIFO and Causal properties are provided in Chapter 5. These properties hold across membership changes because nothing in the membership algorithm modify the session_seq, lts, or seq values of the messages or the delivery rules that apply to these messages. Once the membership change begins all messages that are currently undelivered or incomplete and are recovered during the state exchange will actually be delivered as SAFE messages, providing the strongest possible service, including all of the properties of

the lessor FIFO and Causal services.

THEOREM 6.12 (AGREED MESSAGES)

(a). Agreed messages are Causal messages.

(b). If a server s delivers an agreed message m, then after that event it will never deliver a message that has a lower ord value.

(c). If a server s delivers an agreed message m' before a transsig event in its current view, then s delivers every message with a lower ord value than m' delivered in that view by any process.

(d). If a server s delivers an agreed message m' after a transsig event in its current view, then s delivers every message with a lower ord value then m' sent by any process in s's next transitional set that were delivered in the same view as m'.

PROOF: The first two properties are provided in the proofs of Chapter 5 because they do not involve membership changes. Since an agreed message cannot be delivered until all previous messages have been delivered by the standard delivery algorithm, it can never deliver an older, lower ord, message. Even when messages are skipped in the Deliver_Membership function (see Figure 6.16) because earlier messages are missing, once they are skipped, the messages will never be delivered later because they will have an earlier lts value.

The messages delivered prior to a **`stranssig`** event consist of all those messages delivered before the current instance of membership protocol began as well as those that became deliverable during the algorithm. Since the normal delivery rule for Agreed messages requires all messages ordered prior to the current message be delivered before the current message Theorem 6.12(c) is enforced. The messages that arrive during the membership are only delivered prior to the **`stranssig`** event by the Deliver_Safe_Mess call in Figure 6.13 which delivers all messages that qualify as Safe (their lts $<$ ARU) based on the ARU values provided in the Statetrans messages. Because of Theorem 6.12(b), messages with a lower ord value must be delivered prior to m'. Therefore, all messages with a lower ord value must be delivered in the Deliver_Safe_Mess call or have been already delivered in OP state.

Theorem 6.12(d) requires that the only holes in the agreed order of messages be holes representing messages that originated at servers who are not in the transitional set of the next view. By the delivery rule the only messages that will be skipped are those whose source server is not in the Commit set and that occur after an unrecovered hole in the agreed (LTS) message order. Since the transitional set is a subset of the Commit set, by the algorithm the commit set consists of those members in the transitional set of the current protocol instance as well as any members who this server completed message recovery with in a prior instance that failed to actually deliver a view. □

THEOREM 6.13 (SAFE MESSAGES)

(a). Safe messages are agreed messages.

(b). If a server s delivers a safe message m before a transsig event in its current view v, then every member of that view delivers m, unless that member crashes in v.

(c). If a server s delivers a safe message m after a transsig event in its current view v, then every member of s's transitional set from s's next view delivers m, unless a member crashes in v.

PROOF: Theorem 6.13(b) is a consequence of the delivery rule for Safe messages. If anyone delivers the safe message, then it follows that all of the servers have the message, otherwise they could not have acknowledged it. Finally, the fact that every safe message that was every acknowledged by the server will be delivered. This is true because if a message was acknowledged then the server must have the message and all message ordered earlier in the lts order, since the acknowledgements are cumulative. Thus, when messages are delivered in the Deliver_Membership function, even if a safe message was sent by a server not in the current servers commit set, it will still be delivered because there can not be any holes in the lts order prior to the message.

Theorem 6.13(c) is also a consequence of the same reasoning, but with the caveat that since the safe message was delivered after a `stranssig` event, the delivering server did not get acknowledgements from all other servers in the previous view prior to delivering it; only from the current transitional set during the EVS state. Therefore, some servers who

partitioned away may not have recevied the safe message and so may not deliver it. All servers in the transitional set of s, however, must have received the safe message at the latest during the EVS message exchange since the servers only proceed to deliver the view if all of the exchanged messages during EVS become stable at all of the servers in the new view. Thus, the members of the transitional set of s when v' the successor view of v is delivered will all have the safe message m and all previous messages and will thus deliver it. □

THEOREM 6.14 (TRANSITIONAL SIGNALS)

(a). At most one transsig event occurs at a server during a view.

(b). If two servers s and t are virtually synchronous in a view v, and s has a transsig event occur in v, then t also has a transsig event occur in v, and they both deliver the same sets of agreed messages before and after their transsig events in v.

PROOF: Theorem 6.14(a) results from the algorithm shown in Figure 6.16 as that is the only place where **`stranssig`** events are generated and they are only generated immediately prior to delivering messages and a new **`sview`** event.

The first part of Theorem 6.14(b) is trivial, as a **`stranssig`** event is always delivered in Figure 6.16. The messages delivered prior to the **`stranssig`** are those that were delivered in the Deliver_Safe_Mess() function call made in Figure 6.13 which were safe based on the ARU values exchanged during the STATETRANS messsages. Since by assumption s and t were virtually synchronous and so delivered the same new view, they must have both received the same set of StateTrans messages during the exchange. So they will have computed the same highest ARU upto which they can deliver messages. For the messages delivered after the **`stranssig`**, since they were virtually synchronous they will have received the same sets of recovered messages as they would have been in each other's Commit_Set as they installed the same new view. □

6.3 Evaluation of Daemon Membership

The daemon membership algorithm does have the nice property of guaranteeing to finish running the algorithm within a bounded time. The bound is provided by the part of the algorithm that does not add any new process to the membership once it has passed the commit stage. After that point daemons can continue to fail, forcing the algorithm to recalculate the live membership. Once they fail they will not be allowed back into the membership until this instance of the algorithm has completed.

The actual time to complete a membership is highly dependent on the pattern of failures and the speed at which failures are detected. Once a failure has been detected, the algorithm requires N_a Alive message probes to detect local members, N_j Join messages to find the global membership, 1 Newmemb message to set the new membership, N_s state-trans messages to exchange state amoung all of the sites, N_m messages to retransmit any missing mesages where N_m is the number of missing messages, and finally, requires the usual ARU_Update exchange to discover if all recovered messages are safe and can now be delivered as part of the installing the membership.

The membership algorithm is very affected by tuning parameters, as failure detection is a difficult problem that is highly dependant on the underlying stability and latency of the network on which the process is running. The algorithm can be tuned by changing the timeouts for all of the basic message types and changing timeouts on individual point-to-point wide-area network links if special knowledge about their failure and loss probabilities are known.

6.4 Process Group Membership Algorithm

The group membership algorithm delivers a set of views and data messages to each application connected to a particular Spread daemon. The group algorithm solves two problems. First, it maps the single membership view of servers onto N group specific views where many members may be connected through a single daemon. Second, it provides light-weight group joins and leaves after a client connects to a daemon. These light-weight group changes only ever add or remove one member of a group at a time and are interleved with the heavy-weight group view changes that are triggered by changes in the underlying set of available servers.

Some groups may not even see a group view change when a server view changes as groups which have no members on the daemons who failed, partitioned, or merged will not need to see that any membership change occured at all.

The semantics provided by the group membership algorithm to applications who use the system is presented in Chapter 2 as the general EVS specification as well as a few special cases are presented in Section 2.2.4.

6.4.1 States

The Process Group membership algorithm uses the following states:

GOP Active Operational state when no membership change is in progress.

GTRANS A transitional membership has been received. Awaiting regular membership.

GGATHER A regular membership has been received. During this state the active groups and members state is synchronized by GROUPS messages.

GGT Additional membership changes occurred before the groups synchronization was complete.

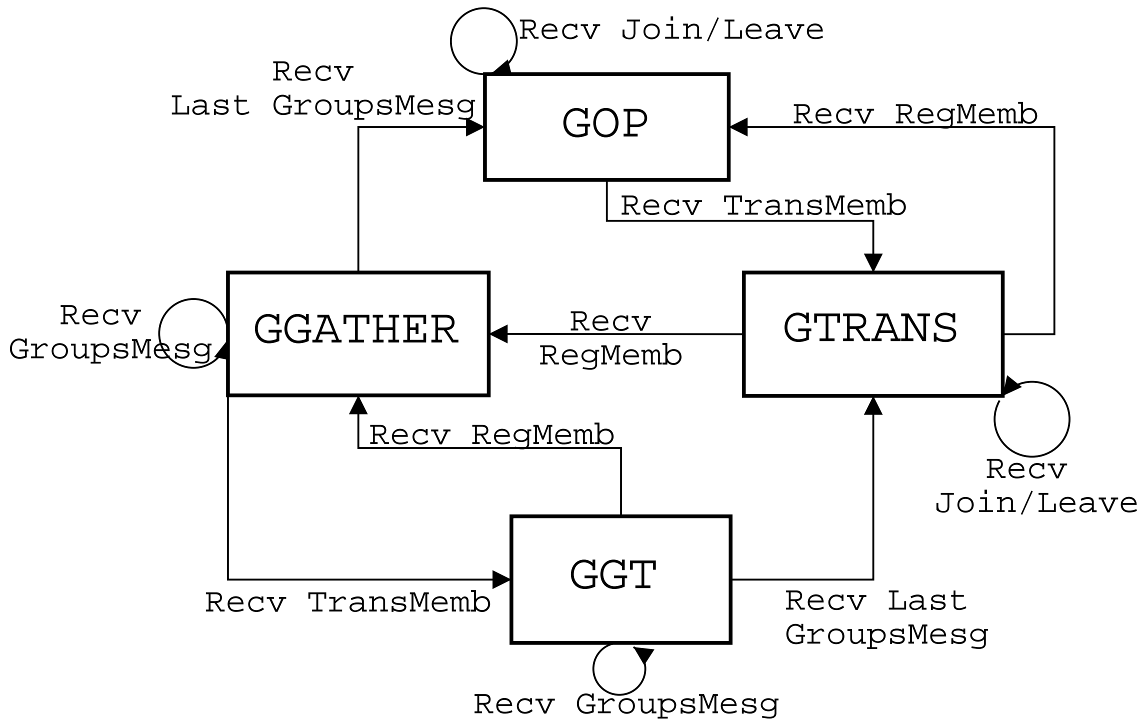

Figure 6.17: Group State Machine

6.4.2 Types of Messages

The group protocol handles the JOIN, LEAVE, KILL, and GROUPS messages as well as responding to view and transsig events. The group messages (JOIN, LEAVE, KILL, GROUPS) are treated as normal data messages by the rest of the Spread daemon. The message type specific to the group protocol is the GROUPS message which contains the state of all of the current groups to which the sending server has locally connected members. The GROUPS message is sent to all of the servers during the group protocol.

- GROUPS: contains a list of known groups and all the members of those groups who are connected locally to the sending server. All members received in a GROUPS message are in ESTABLISHED state.

6.4.3 Possible Events

During the group membership algorithm the following events can occur:

- Receive a transitional signal.
- Receive a view.
- Receive a GROUPS message.
- Receive a JOIN message.
- Receive a LEAVE message.
- Receive a KILL message.

- input $connect(c, s), c \in C, s \in S$
- input $disconnect(c, s), c \in C, s \in S$
- input $send(c, s, m), c \in C, s \in S, m \in M$
- input $join(c, s, g), c \in C, g \in G$
- input $leave(c, s, g), c \in C, g \in G$
- output $deliver(c, s, g, m), c \in C, s \in S, g \in 2^G, m \in M$
- output $view(c, s, g, id, D, T), c \in C, s \in S, g \in G, id \in Vid, D \in 2^P, T \in 2^P$
- output $transsig(c, s, g, id), c \in C, s \in S, g \in G, id \in Vid$

The complete signature also includes the actions:

- input $sdeliver(s, m)$
- input $sview(s, id, D, T)$
- input $stranssig(s, T)$
- output $ssend(s, m)$

The state of the group membership algorithm consists of:

Gstate The current state the algorithm is in.

TMemb The current transitional membership set.

RMemb The current regular membership set.

Gathered The set of GROUPS messages received.

LC The set of local connections to this server.

G The set of active groups the server knows about.

6.4.4 GOP State

The GOP State acts as both the normal operational state and the start state for the Group Membership state machine. In this state, single member join, leave or kill events can occur as well as normal message delivery. If a server-level membership change occurs the first notice that the group membership receives about it is a **`stranssig`** event.

The **`join`**, **`leave`**, and **`kill`** events are light-weight events that are implemented as regular data messages sent in AGREED order. When one of these events is received a corresponding data message is sent through the **`ssend`** event. When the message is delivered to every process the member is acutally added or removed from the appropriate group. Each group consists of a name, a set of members, a set of local members, and a group identifier. The set of local members is the subset of the members who are directly connected to this server. A group is implicitly created when the first member joins the group and destroyed when the last member leaves the group.

Each member of a group has a name, a server it is connected to, and a status. The status field is used to deliver the right type of view events to the client. Normally, a member is in the ESTABLISHED state, which means it is a member of group and is connected to a server who is still present, the server has not crashed or partitioned away. The other possible states are PARTITIONED, which means the member's server has partitioned away so the member will be removed from the group, and NEW, which means the member just joined and should be given a first view event with the transitional set T set to only itself.

In the **`join`** case, shown in Figure 6.18, after the **`sdeliver`** event for the JOIN message occurs, the group the member is joining is created, if it does not already exist, and the new member m is created with the joining member's identity. Then m is added to the set

of members of the group and the group view index is incremented. This will cause a new view id which is ordered after any previous view id. The member m will be in the NEW state until after the `view` event is generated showing it has joined the group.

Two different `view` events are generated in response to the JOIN message. First, a view is generated for any local members who were already in the group before the new member joined. Their view will show the complete new membership and a transitional membership of the new membership set minus the joining member. The second view is generate only for the joining member and it shows the complete new membership and a transitional set of only the joining member because the new member was not previously with any other group members.

When the `leave` and `disconnect` events take effect upon the delivery of the LEAVE and KILL messages respectively, they are handled in a very similar way. The details are shown in Figure 6.19. In both cases the member is removed from the group membership set and from the local set if they are connected to the server, and if the group still exists (i.e. it has at least one member). The index field is incremented to produce the next view id and a new view is delivered to the remaining members with the transitional set being equal to the membership set.

The only differences are that in the KILL case the member is removed from all of the groups, not just one, and in the LEAVE case a Self Leave view is generated for the departing member to notify them about where in the stream of messages their leave took effect. A Self Leave notification is not delivered in the case of KILL messages because the entire connection to the client has been destroyed, so it is no longer possible to deliver any views to the client any more.

6.4.5 GTRANS State

The GTRANS state handles those messages that occur subsequent to a `stranssig` event from the EVS-S algorithm. These messages may be data messages; join, leave, or kill

GOP State

```
1  switch (event)
2  sdeliver(s, JOIN) :
3      if join.group ∉ G then
4         g.name ⇐ join.group
5         g.members ⇐ NIL
6         g.local ⇐ NIL
7         g.grpid.membid ⇐ Memb_id
8         g.grpid.index ⇐ 0
9         G ⇐ G ∪ g
10     else
11           g ∈ G | g.name = join.group
12     m.name ⇐ join.sender
13     m.server ⇐ CLIENT2SERVER(join.sender)
14     m.status ⇐ NEW
15     g.members ⇐ g.members ∪ m
16     if join.sender ∈ LC then
17        g.local ⇐ g.local ∪ m
18     g.grpid.index ⇐ g.grpid.index + 1
19     if |g.local| > 0 then
20        view(g.local − m, Self, g.name, g.grpid, g.members, g.members − m)
21        if join.sender ∈ LC then
22           view(join.sender, Self, g.name, g.grpid, g.members, join.sender)
23        m.status ⇐ ESTABLISHED
24 sdeliver(s,m —m.type = DATA) :
25     for each c in LC | c ∈ m.group
26       deliver(c, Self, m.group, m)
27 join(c, g) :
28     ssend(c, s, JOIN)
29 leave(c, g) :
30     ssend(c, s, LEAVE)
31 disconnect(c, s) :
32     ssend(s, KILL)
33 send(c, s, m) :
34     ssend(s, DATA)
35 connect(c, s) :
36     LC ⇐ LC ∪ c
```

Figure 6.18: Group Membership – GOP State

GOP State (Part 2)

```
1  switch (event)
2  sdeliver(s, LEAVE) :
3        g = leave.group
4        if leave.sender ∈ LC then
5           view(leave.sender, Self, g.name, NIL, NIL, NIL)
6           g.local ⇐ g.local − leave.sender
7        g.members ⇐ g.members − leave.sender
8        if |g.members| = 0 then
9           G ⇐ G − g
10       g.grpid.index ⇐ g.grpid.index + 1
11       if |g.local| > 0 then
12          view(g.local, Self, g.name, g.grpid, g.members, g.members)
13 sdeliver(s, KILL) :
14       for each g in G | kill.sender ∈ g
15          if kill.sender ∈ LC then
16             g.local ⇐ g.local − kill.sender
17          g.members ⇐ g.members − kill.sender
18          if |g.members| = 0 then
19             G ⇐ G − g
20          g.grpid.index ⇐ g.grpid.index + 1
21          if |g.local| > 0 then
22             view(g.local, Self, g.name, g.grpid, g.members, g.members)
23 stranssig(s, T) :
24       TMemb ⇐ T
25       for each g in G
26          group_changed ⇐ FALSE
27          for each m in g.members
28             if m.server ∉ TMemb then
29                m.status ⇐ PARTITIONED
30                group_changed ⇐ TRUE
31             if group_changed then
32                if |g.local| ≥ 0 then
33                   transsig(g.local, Self, g.name, g.grpid)
34                g.grpid.index ⇐ −1
35          Gstate ⇐ GTRANS
36 sview(s, id, D, T) :
37       impossible
38 sdeliver(s, GROUPS) :
39       impossible
```

Figure 6.19: Group Membership – GOP State (Part 2)

messages; or the new **sview** event. Most of these events result in changes to the state of some group. The essential difference from the GOP state is that some of the servers whose events are received may not be synchronized with the executing server and so their events can not be directly applied to the state, but must rather be 'tentatively' applied and will only take full effect after the state synchronization occurs in GGATHER state.

The receipt of a **sdeliver** event for a JOIN message in GTRANS state causes the joining member to be added to the group, just like in GOP state. However, the view id and status of the joining member differs depending on which server the join originated from and the current view id index field of the group. The details of handling a JOIN message are shown in Figure 6.20. Since a transitional signal event has occured the join may originate from a server who is in the TMemb set of this server or it may originate at a server who is not in the TMemb set.

If the originating server is not in the TMemb set, then the new member is considered to be a PARTITIONED member because the two servers are not necessarily synchronized. Since the group now has some partitioned members the index field is made negative, if it is not already, and the delivered view has a transitional set consisting of only those members who are in ESTABLISHED state. This will clearly not include the new joining member as it was joined in PARTITIONED state. A **transsig** event is immediately generated as even though a new view was delivered with the joining member. More membership changes will immediately follow because the **sview** event has not occured yet. The **sview** will require the state to be synchronized and a new view delivered in the GGATHER state.

If the originating server is in the TMemb set, and if no previous view change has marked the group as inconsistent (the index is greater then 0), the index can be incremented and a view delivered to the group just like in GOP state. However, if the index is negative, this group has some PARTITIONED members and so the view with the joining member can be delivered only with the transitional set consisting of only ESTABLISHED members of the group. A **transsig** event must be immediately delivered to indicate that messages

delivered after this are only guaranteed with respect to the transitional set and not the compete membership set. A later view will be delivered when the **sview** event occurs just like when the originating server is not in the TMemb set.

Figure 6.21 shows how the LEAVE and KILL messages are handled similarly to GOP state with one exception. When the member is removed from the group a **view** event is sent to the group and the view id is increased only if the index field is positive. If the index field is negative, this group has PARTITIONED members and it's current membership set will be constructed by the GROUPS message exchange in GGATHER. In this case the delivery of the view showing the departure of the leaving member is unnecessary as the subsequent view delivered in GGATHER will show the member has left and any events subsequent to the leave and prior to the new **view** will not include the member who left. This differs from the join case because the member who joins legitimately expects to receive events that occured subsequent to their join, while a leaving member expects nothing subsequent to their leave.

DATA messages can occur in GTRANS state and will be delivered as normal to the current set of group members. These messagaes may be subsequent to a **transsig** event or not, depending on whether or not the group they are sent to has any PARTITIONED members. In either case, all data messages received from the EVS-S algorithm must be delivered to all members of the group.

In the GTRANS state, all client initiated events are disabled. This includes the **join**, **leave**, **disconnect** and **send** events. These events are disabled because until the EVS-S membership algorithm has completed no new messages will be initiated by any of the servers.

Finally, the **sview** event handling is shown in Figure 6.22. The set of servers in the new **sview** is stored in the RMemb field and the view id of the **sview** is stored in the RMemb_id field. This will be used to construct the new view id delivered in the **view** event.

GTRANS State

```
1   switch (event)
2   sdeliver(s, JOIN) :
3       if join.group ∉ G then
4           g.name ⇐ join.group
5           g.members ⇐ NIL
6           g.local ⇐ NIL
7           g.grpid.membid ⇐ Memb_id
8           g.grpid.index ⇐ −1
9           G ⇐ G ∪ g
10      else
11              g ∈ G | g.name = join.group
12      m.name ⇐ join.sender
13      m.server ⇐ CLIENT2SERVER(join.sender)
14      m.status ⇐ NEW
15      g.members ⇐ g.members ∪ m
16      if join.sender ∈ LC then
17          g.local ⇐ g.local ∪ m
18      if m.server ∈ TMemb then
19          if g.grpid.index ≥ 0 then
20              g.grpid.index ⇐ g.grpid.index + 1
21              if |g.local| > 0 then
22                  view(g.local − m, Self, g.name, g.grpid, g.members, g.members − m)
23                  if join.sender ∈ LC then
24                      view(join.sender, Self, g.name, g.grpid, g.members, join.sender)
25          else  g.grpid.index ⇐ g.grpid.index − 1
26                  if |g.local| > 0 then
27                      view(g.local − m, Self, g.name, g.grpid, g.members,
28                      {n | n ∈ g.members ∧ n.status = ESTABLISHED})
29                      if join.sender ∈ LC then
30                          view(join.sender, Self, g.name, g.grpid, g.members, join.sender)
31                      transsig(g.local, Self, g.name, g.grpid)
32          m.status ⇐ ESTABLISHED
33      else  m.status ⇐ PARTITIONED
34              if g.grpid.index > 0 then
35                  g.grpid.index ⇐ −1
36              else  g.grpid.index ⇐ g.grpid.index − 1
37              if |g.local| > 0 then
38                  view(g.local, Self, g.name, g.grpid, g.members,
39                  {n | n ∈ g.members ∧ n.status = ESTABLISHED})
40                  transsig(g.local, Self, g.name, g.grpid)
```

Figure 6.20: Group Membership – GTRANS State

GTRANS State (Part 2)

```
1  switch (event)
2  sdeliver(s, GROUPS) :
3      impossible
4  sdeliver(s, LEAVE) :
5      g = leave.group
6      if leave.sender ∈ LC then
7         view(leave.sender, Self, g.name, NIL, NIL, NIL)
8         g.local ⇐ g.local − leave.sender
9      g.members ⇐ g.members − leave.sender
10     if |g.members| = 0 then
11        G ⇐ G − g
12     if g.grpid.index ≥ 0 then
13        g.grpid.index ⇐ g.grpid.index + 1
14        if |g.local| > 0 then
15           view(g.local, Self, g.name, g.grpid, g.members, g.members)
16 sdeliver(s, KILL) :
17     for each g in G | kill.sender ∈ g
18        if kill.sender ∈ LC then
19           g.local ⇐ g.local − kill.sender
20        g.members ⇐ g.members − kill.sender
21        if |g.members| = 0 then
22           G ⇐ G − g
23        if g.grpid.index ≥ 0 then
24           g.grpid.index ⇐ g.grpid.index + 1
25           if |g.local| > 0 then
26              view(g.local, Self, g.name, g.grpid, g.members, g.members)
27 sdeliver(s,m — m.type = DATA) :
28     for each c in LC | c ∈ m.group
29        deliver(c, Self, m.group, m)
30 join(c, g) :
31     disabled
32 leave(c, g) :
33     disabled
34 disconnect(c, s) :
35     disabled
36 send(c, s, m) :
37     disabled
38 connect(c, s) :
39     disabled
```

Figure 6.21: Group Membership – GTRANS State (Part 2)

The straightforward method of handling all **sview** events would be to always send a GROUPS message and switch to GGATHER state to synchronize. However, a more efficient approach is possible in which synchronization only occurs when required because two servers have different state. That approach uses the transitional set of the **stranssig** and the membership set of the **sview** events, and the EVS-S Transitional Set properties that guarantee those members in a server's transitional set installed the same view in the same previous view. By the Virtual Synchrony property, they will have seen the same set of messages in the previous view. Thus, they will have exactly the same state since the group state is a deterministic function of the received messages. This knowledge of transitional sets allows a **view** event to be immediately generanted without any synchronization when the TMemb set equals the RMemb set.

Specifically, the first half of the **sview** handler provides the algorithm when this optimization is possible. For each group all PARTITIONED members are removed from the group, any groups that are now empty are removed, and a new view id is constructed with the RMemb_id and delivered in a **view** event whose membership set and transitional set are equal. In this case the entire group membership algorithm required no additional communication and local computation equal to the costs of searching and removing members from a local datastructure.

When the transitional set is less than the member set, it is necessary to synchronize the group state of all of the servers. To initiate this process for all of the groups who have a negative index, the members who are in the PARTITIONED state are removed from that group and the view id of the group is changed to be the new RMemb_id and an index field of -1. The negative index field indicates that this group had members in PARTITIONED state and so is changed from the last **view** event. The server then sends a GROUPS message as a SAFE message. The GROUPS message contains all of the groups the server knows about and all of the current members of those groups. This list of current members is not the same as the members who were in the group at the time of the last **view** event, however. The

current member list may have changed because of leave or disconnect events or because members of the group were connected to servers who are not in the TMemb set and so were marked PARTITIONED. It is true that the current member set will be a subset of the previous view, because no members will have been added without delivering a new **view**.

6.4.6 GGATHER State

The GGATHER and GGT states are used to complete the transfer of all of the group state amoung the new set of servers. These two states are only needed when servers who were not previously reachable from the executing server become reachable. If servers only fail then the transitional set TMemb will equal the final membership RMemb and the GTRANS **sview** event handler will complete the new view and switch to GOP state. The purpose of the GGATHER state is to collect all of the GROUPS messages and create the subsequent **view** event.

During the GGATHER state the only possible events are receiving a GROUPS message from another server and experiencing an additional server level membership change where a new **stranssig** is delivered. No other messages can occur in this state for two reasons. First, because the EVS-S algorithm guarantees that it will not accept any client messages between detecting the failure and delivering the **sview** event. No data messages will be initiated until after the EVS-C algorithm gets a chance to handle the **sview** event. Secondly, when the **sview** event is handled, the server stops accepting new client requests by blocking all connected clients and switching the queue. Messages will be pulled from to a special queue only used by the EVS-C algorithm.

Because of the atomic nature of the delivery of the **stranssig** event, the following data messages, and the new **sview** event, the server will not have accepted any client events once the first **stranssig** event occurs, Therefore, the blocking of new client requests can actually be viewed as beginning when the first **stranssig** event occurs.

When a GROUPS message is received, as shown in Figure 6.23, the message is checked

GTRANS State (Part 3)

```
1   switch (event)
2   stranssig(s, T) :
3         impossible
4   sview(s, id, D, T) :
5         RMemb ⇐ D
6         RMemb_id ⇐ id
7         if |TMemb| = |RMemb| then
8            for each g in G
9               if g.grpid.index < 0 then
10                 for each m in g.members
11                    if m.status = PARTITIONED then
12                       g.members ⇐ g.members − m
13                 if |g.members| = 0 then
14                    G ⇐ G − g
15                 else g.grpid.membid ⇐ RMemb_id
16                       g.grpid.index ⇐ 1
17                       if |g.local| > 0 then
18                          view(g.local, Self, g.name, g.grpid, g.members, g.members)
19            Gstate ⇐ GOP
20        else
21              for each g in G
22                 if g.grpid.index < 0 then
23                    for each m in g.members
24                       if m.status = PARTITIONED then
25                          g.members ⇐ g.members − m
26                    if |g.members| = 0 then
27                       G ⇐ G − g
28                    else g.grpid.membid ⇐ RMemb_id
29                          g.grpid.index ⇐ −1
30              Blocknewclientrequests!
31              UseGroupsdownqueue!
32              ssend(c, s, GROUPS)
33              Gstate ⇐ GGATHER
```

Figure 6.22: Group Membership – GTRANS State (Part 3)

GGATHER State

```
 1  switch (event)
 2  sdeliver(s, JOIN) :
 3      impossible
 4  sdeliver(s, LEAVE) :
 5      impossible
 6  sdeliver(s, KILL) :
 7      impossible
 8  sdeliver(s, DATA) :
 9      impossible
10  sdeliver(s, GROUPS) :
11      if groups.membid ≠ RMemb_id then
12         ignore
13      Gathered ⇐ Gathered ∪ groups
14      if |Gathered| = |RMemb| then
15          Set normal downqueue!
16          Enable client events!
17         COMPUTENOTIFY()
18         Gstate ⇐ GOP
19
```

Figure 6.23: Group Membership – GGATHER State

to make sure it was generated by the current algorithm instance, because cascaded EVS-S view changes could cause several versions of GROUPS messages to be delivered to each server. If the GROUPS messsage is valid, it is added to the Gathered set. When all of the required GROUPS messages have been received, one from every server in the new **sview**, the message queues are switched back because all of the special EVS-C algorithm messages are complete. The ComputeNotify function is called which merges all of the GROUPS messages state information and constructs the new membership state for every group in the system. In this function the **transsig** and **view** events are generated.

The details of the ComputeNotify function can be seen in Figure 6.24. Essentially, during ComputeNotify each group, that at least one GROUPS message has any information about, is created, if it does not already exist, and the membership sets reported by all of the servers are merged. The transitional set of the new view is set to the membership sets of those servers who had the same previous view id for the particular group as the executing

ComputeNotify()

```
1  for each g[] in Gathered | ∀i, g[i].name = SMALLEST_NAME(Gathered)
2    ts ⇐ NIL
3    if g[0].name ∉ G then
4      og.name ⇐ g[0].name
5      og.grpid ⇐ g[0].grpid
6      og.members ⇐ NIL
7      og.local ⇐ NIL
8    else og ⇐ {og ∈ G | og.name = g[0].name}
9        og.members ⇐ NIL
10   for i = 0; i < |g[]|; i ⇐ i + 1
11     if og.grpid = g[i].grpid then
12       ts ⇐ ts ∪ g[i].members
13     else changed ⇐ 1
14     og.members ⇐ og.members ∪ g[i].members
15   if og.grpid.index = −1 then
16     changed ⇐ 1
17   if changed then
18     og.grpid.membid ⇐ RMemb_id
19     og.grpid.index ⇐ 1
20     if |og.local| > 0 then
21       view(og.local, Self, og.name, og.grpid, og.members, ts)
22 Gathered ⇐ NIL
```

Figure 6.24: Group Membership – ComputeNotify function

server. The view id of the new group view is set to the view id reported by the **sview** event together with an index value of 1. The initial index of 1 for the view id establishes the new series of views within an EVS-S view, which changes the view id but does not change the index field of the complete view id.

When cascaded membership changes occur a **stranssig** event will be received in the GGATHER state. This is shown in Figure 6.25. Here any group member's connected server that is not in the transitional set of the **stranssig** event will be removed from the local group state. The group they were removed from will have its index set to -1. This marks the group as having changed because of servers who left the current configuration. Because this is a cascaded change, not the first **stranssig** event, the state then switches to GGT instead of GTRANS because none of the special cases handled in GTRANS apply.

GGATHER State (Part 2)

```
1   switch (event)
2   stranssig(s, T) :
3       TMemb ⇐ T
4       for each g in G
5         group_changed ⇐ FALSE
6         for each m in g.members
7           if m.server ∉ TMemb then
8             g.members ⇐ g.members − m
9             group_changed ⇐ TRUE
10        if |g.members| = 0 then
11          G ⇐ G − g
12        else if group_changed then
13              if index ≥ 0 then
14                transsig(g.local, Self, g.name, g.grpid)g.grpid.index ⇐ −1
15      Gstate ⇐ GGT
16  sview(s, id, D, T) :
17      impossible
18  join(c, g) :
19      disabled
20  leave(c, g) :
21      disabled
22  disconnect(c, s) :
23      disabled
24  send(c, s, m) :
25      disabled
26  connect(c, s) :
27      disabled
```

Figure 6.25: Group Membership – GGATHER State (Part 2)

6.4.7 GGT State

The GGT state is the most straightforward of the states because only two events can occur: the **`sdeliver`** of a GROUPS message, or receipt of an **`sview`** event.

As presented in Figure 6.27, if an **`sview`** event occurs then the current set of received GROUPS messages is flushed. They are no longer useful because the current set of servers will have to resynchronize not some past set. Then a new GROUPS message is sent and the state switches to GGATHER. This essentially restarts the synchronization process at all of the servers who are in the new membership. Since the necessary GROUPS messages were not received prior to the new **`sview`** event, by the Same View Delivery property of the EVS-S algorithm if the server receives them later, no other server will have received those missing GROUPS messages prior to the **`sview`** event. The missing messages would have been delivered in different views. If a server does not receive the GROUPS message but some other server does, then they must be in a different network component and will not be part of the new view, so it is acceptable that they installed the other view.

The interesting case is shown in Figure 6.26 when the required GROUPS messages are received in the GGT state, then just as if they had been received in GGATHER. A new view is computed and delivered by the ComputeNotify function. The difference is that in addition to switching to GOP state, a new **`stranssig`** event is generated, just as if a new membership change was occuring from the EVS-S algorithm. The generated **`stranssig`** event includes the TMemb set that was delivered in the most recent 'real' **`stranssig`** received. This handles the case where all of the required GROUPS messages were received, but some were delivered after the transitional signal. Therefore, this server must deliver the view because some other server may have received the GROUPS messages prior to the transitional signal and also delivered the view.

GGT State

```
1  switch (event)
2  sdeliver(s, JOIN) :
3      impossible
4  sdeliver(s, LEAVE) :
5      impossible
6  sdeliver(s, KILL) :
7      impossible
8  sdeliver(s, DATA) :
9      impossible
10 sdeliver(s, GROUPS) :
11     if groups.membid ≠ RMemb_id then
12        ignore
13     Gathered ⇐ Gathered ∪ groups
14     if |Gathered| = |RMemb| then
15         Set normal downqueue!
16         Enable client events!
17        ComputeNotify()
18        Gstate ⇐ GOP
19        stranssig(s, TMemb)
```

Figure 6.26: Group Membership – GGT State

GGT State (Part 2)

```
1  switch (event)
2  stranssig(s, T) :
3      impossible
4  sview(s, id, D, T) :
5      RMemb ⇐ D
6      RMemb_id ⇐ id
7      Gathered ⇐ NIL
8      ssend(c, s, GROUPS)
9      Gstate ⇐ GGATHER
10 join(c, g) :
11     disabled
12 leave(c, g) :
13     disabled
14 disconnect(c, s) :
15     disabled
16 send(c, s, m) :
17     disabled
18 connect(c, s) :
19     disabled
```

Figure 6.27: Group Membership – GGT State (Part 2)

6.5 Proof of Group Membership

The group membership algorithm runs on top of the server-based membership algorithm which provides EVS semantics. Thus, the entire group membership algorithm can assume that all of the EVS properties specified in the EVS-S automaton are provided.

The structure of a view id must be defined. This structure is not necessarily visable to applications, as they are given a function which compares view id's for the application. The structure and the rule for comparing ids is important for the Local Monotonicity property because the view ids are not a simple number and the way they are compared is non-trivial.

DEFINITION 6.1 (VIEW ID COMPARISON RULE)
The following rule is used to compare id's gid1 and gid2:
If the $gid1.membid > gid2.membid$ *then* $gid1 > gid2$*. If* $gid1.membid = gid2.membid$*, then two cases exist: If* $gid1.index > 0$ ***and*** $gid2.index < 0$ *(i.e. one is positive and one is negative) then* $gid2 > gid1$ *(i.e. negative index values are greater then positive index values). Finally if gid1.index and gid2.index have the same sign (both positive or both negative), and if* $|gid1.index| > |gid2.index|$ *then* $gid1 > gid2$ *(i.e. whichever one has the larger absolute value is the greater gid).*

One aspect of the group membership algorithm that simplifies it is that the algorithm never changes the order of messages. All messages are delivered to the application in the same order they were provided by the EVS-S layer. The Group membership algorithm does have to block new messages from being sent for a period of time but it always delivers actual user data messages immediately and does not buffer them.

LEMMA 6.9 (IDENTICAL ORDER)
All messages are delivered by the group membership algorithm in the same order they are were delivered by the server membership algorithm.

PROOF: In the algorithm, the only two places where **`deliver`** events occur for data messages are on GOP and GTRANS. At those times, data messages are delivered immediately to all of the members of the groups. Nowhere does the algorithm buffer data messages or reorder them. □

The index field of the view id follows a fairly complex set of rules. Therefore several properties of the index are isolated in the following lemmas.

LEMMA 6.10 (POSITIVE INDEX IN GOP)
The index field of a grpid is always ≥ 0 *when* $GState = GOP$.

PROOF: Two transitions to GOP state exist: The first, at the end of handling a GROUPS message in GGATHER or GGT state and, the second, when a **`sview`** event is received that does not include any new members. In the first case, the ComputeNotify function is called. ComputeNotify always sets the index to 1 if it was negative prior to delivering the new **`view`** event, so all groups will have $index \geq 0$ when shifted to GOP state. In the second case, every group that has a negative index value is either removed entirely if all of its members have partitioned away, or has its index value set to 1. So again in GOP state the index will always be positive. □

LEMMA 6.11 (CHANGING NEGATIVE TO POSITIVE INDEX)
The only time a negative index value on a group view id can be changed to a postive index is immediately prior to switching to GOP state.

PROOF: The only locations where the index is assigned a postive value are in ComputeNotify and in the GTRANS handler for **`sview`** events. In all other locations an index is incremented only if it is already positive. Therefore, these are the only locations where an index can change from negative to positive. In both places the ComputeNotify function is called and after it completes the state is switched to GOP. In the GTRANS state **`sview`** handler, after delivering all of the views the state is switched to GOP. □

LEMMA 6.12 (GROUP INDEX FIELD IS EITHER -1 OR POSITIVE IN GROUPS)
Every group in a GROUPS message has an index field of either -1 or a positive value if the group is synchronized.

PROOF: GROUPS messages are generated in two places. In GTRANS when a **`sview`** event is received and in GGT when a **`sview`** event is received. In GTRANS, the algorithm

shows that every group that has a negative index value is assigned an index of -1 before the GROUPS message is sent. In the other case the index value is not changed when a GROUPS is generated in GGT. However, for several reasons the index field will still be -1 or a positive value when a GROUPS is sent in GGT. First, the only way the algorithm can enter GGT state is after entering GGATHER and the only way it can enter GGATHER is through the **`sview`** handler of GTRANS which sets the index to -1. Second, no part of the algorithm in GGATHER or GGT sets the index to anything besides -1 or a positive number . □

The state of a member of a group can be one of three states: NEW, ESTABLISHED, or PARTITIONED. The NEW state is only used during the execution of a JOIN message handler and separates the joining member from all of the existing members. The ESTABLISHED state represents the normal state of a member that is currently in the group. The only way to leave the ESTABLISHED state is to leave the group or become PARTITIONED. The PARTITIONED state represents a group member who is connected to a server who is not synchronized with the local server. Therefore, PARTITIONED is a relative state from the point of view of the server's local datastructures. Since servers may be in different components, or may have different transitional sets, it is expected that a group member will be in ESTABLISHED state at one server and in PARTITIONED state at another server. The following lemmas establish some of the basic invarients of the member state field.

LEMMA 6.13 (ESTABLISHED REQUIRES PREVIOUS VIEW MEMBERSHIP)
Every member of a group whose status is ESTABLISHED, was a member of the previous view for that group.

PROOF: The only places where a group member is set to ESTABLISHED status is when they join the group in the **`sdeliver`**(s, JOIN) event of either the GOP or GTRANS state. When the member is set to ESTABLISHED state they have already been part of the group membership of the view that showed them joining. Thus, they are in the previous view for

the group. □

LEMMA 6.14 (STATUS IS ESTABLISHED OR PARTITIONED)
Every group member has a status of ESTABLISHED or PARTITIONED both before starting an event handler, and after completing an event hander.

PROOF: The only time any member has a status besides ESTABLISHED or PARTITIONED is when the member first joins the group during which they have a status of NEW. The only two places this occurs is the **`sdeliver`**(s, JOIN) event handler for GOP or GTRANS. At the end of the event handler the member is set to ESTABLISHED state. □

LEMMA 6.15 (GROUPS MEMBERS ESTABLISHED)
All members sent in a GROUPS message will be marked ESTABLISHED.

PROOF: Immediately before sending a GROUPS message, the algorithm loops over every group and removes all members whose status is PARTITIONED. By Lemma 6.14 the only state members can be in prior to starting an event is PARTITIONED and ESTABLISHED. □

LEMMA 6.16 (TRANSSIG IMPLIES VIEW EVENT PRIOR TO GOP)
If a transsig event was delivered for group g, then a subsequent view event will be delivered for group g prior to the server reentering GOP state.

PROOF: The server can reenter GOP state in only two places: First, when an **`sview`** event is received in GTRANS state and TMemb equals RMemb. Second, in GGATHER or GGT state after ComputeNotify is called. In the first case, as seen in Figure 6.22, for every group with negative index if the group has any members a **`view`** is delivered to all the local members. When a **`transsig`** event is delivered, either in GOP or GTRANS, the index field is always made negative, either by assignment to -1 or by decrementing a value that is already negative. Therefore, as seen in Lemma 6.11 the index will still be negative because the only places it can become positive are the two cases we discus in this proof. If a **`transsig`** event occurs, the index of the group will become negative and will stay

negative until the `sview` handler in GTRANS both deliveres a `view` event and makes the index positive.

In the second case, the ComputeNotify function shown in Figure 6.24, a view will be delivered for every group that either has an index field of -1, or that is the result of merging two separate previous views of the same group (if $g[i].grpid$ is different from $og.grpid$). By Lemma 6.12 the index field received in a GROUPS message will be -1 or positive, and by Lemma 6.11 if it was negative previously as a result of a `transsig` event, it must still be negative (-1 actually) because the code that can set it positive has not been executed yet since the server has not reentered GOP state. □

THEOREM 6.15 (INITIAL VIEW EVENT)
In all traces of the EVS-C automaton, every deliver and transsig event occurs in some view.

PROOF: A deliver event is only generated when the process who receives it is a member of the group at the server it is connected to. The only ways a server adds a process to a group is upon receiving a `sdeliver` event for a JOIN message. When handling a JOIN message the member is added to the group and a view message is sent to the process prior to any other event being handled.

When merging a group upon receiving a `sview` event, the set of group members is made up of all previous group members in any of the components. These members were already group members and so had already received a view event for that group.

Before a member joins a group no `transsig` event is generated for that user because they are not in the local member list for the group. □

THEOREM 6.16 (SELF INCLUSION)
All view events include Self in the set of members.

PROOF: A `view` event is only delivered to a process who is a member of the group because the algorithm verifies that membership before delivering the event. □

THEOREM 6.17 (LOCAL MONOTONICITY)
If process p installs a view with identifier id' after installing a view with identifier id, then id' is greater then id.

PROOF: The identifier is created by combining the **`sview`** identifier, called grpip.membid, with an index field represented by grpid.index in the algorithm. The index field is a signed integer which varies between $-\infty$ and $+\infty$.

Definition 6.1 specifies how viewids are ordered. One must prove that the algorithm always delivers views with viewid's that increase according to that order.

The algorithm provides the monotonicity property by guaranteeing that whenever the grpid is changed the change always proceeds in the following steps (possibly skiping steps): Set membid = RMemb_id provided by EVS-S, set index to 1; Increase index by one; Set membid = RMemb_id provided by EVS-S, set index to -1; Set index to -1; Decrease index by one. After reaching a negative index the only way the index becomes positive again is when the membid field is reset to a new and higher value, because EVS-S provides Local Monotonicity. Within a series of views during which no sview events are received, the only change to grpid is to the index value and that value can only increase or be set to -1 if positive and can only decrease if negative.

At every view, either the membid field or the index field or both are changed. In the algorithm, every **`view`** event is preceeded by one of these types of changes. □

THEOREM 6.18 (TRANSITIONAL SET)

(a). The transitional set for the first view installed at a process following a join event consists of just the process itself.

(b). If a process p installs a view v' in a previous view v, then the transitional set for v' at p is a subset of the intersection of M_v and $M_{v'}$.

(c). If processes p and q install the same view, then q is included in p's transitional set for that view if and only if p's previous view was identical to q's previous view.

(d). If processes p and q install the same view v' in the same previous view v, then they have the same transitional sets in v'.

PROOF:

Property a When a view is created because of a process executing a **`join`** event, the algorithm always delivers a special view to only the process who joined. That special view always sets the transitional set to be the joining process.

Property b With respect to the transitional set, all of the **`view`** events in the algorithm can be classified as one of five types:

1. Self-Leave: The self leave view delivered to a member who leaves a group is a NILview which is a subset of any set.

2. Leave or Kill: The transitional set delivered to the group in both of these cases is the same as the current membership. The current membership always consists of the previous membership minus the leaving member. Therefore, the current membership is a subset of the intersection of the previous view and current view.

3. Single Joiner: When a member joins a group, the first view they receive contains a transitional set of only themselves. Since this is the first view, no previous view exists. Thus, the precondition is false.

4. Members Joined to: When a single member joins a group the other members receive a view with a transitional set of all of the members of the current group who are in the ESTABLISHED state. By Lemma 6.13 every member who is in ESTABLISHED state was a member of the previous view. As a result the set of current members who are also in ESTABLISHED state must also be members of the previous view, and so are in the intersection of the previous and current view.

5. View generated in ComputeNotify: This view message represents the results of the group state exchange protocol. The transitional set is constructed of all group memberships sent by daemons who had the same previous view id. This is clearly a subset of the current membership. It is also a subset of the previous membership because the

members sent in the groups message will only be those members who had ESTABLISHED state when the groups message was sent (from the algorithm all members with PARTITIONED status will be removed from the group prior to sending the groups message, and from Lemma 6.14 every member is either in ESTABLISHED or PARTITIONED state when the **sview** event is handled, and the **sview** event handler does not set any member to any state).

So in all cases the transitional set is in the intersection of the previous and current views.

Property c In the case of a join, leave, or disconnect event the only way p and q can install the same view in the same previous view is if neither is the joining or leaving member. In that case the transitional set delivered at both of them will be the one of

1. the current member set in the case of a leave or disconnect, which will include all members except the one who left, so it will certainly include q.

2. the current member set except for the joining member in the case of a join. This set clearly contains all members of the group including q as q is not the joining member by assumption.

3. the current members who are of ESTABLISHED state. In this case by Lemma 6.13 all of the ESTABLISHED members were in the previous view

When an **sview** event occurs there are two possible ways the new view is generated. First, if TMemb = RMemb, the only change to the EVS-S membership was servers leaving, no new servers merged. Second, servers both merged and partitioned so a GROUPS message will be sent and the GGATHER state will be entered. In either case, if a **join** event occurs prior to the **sview** event the new view and transitional set will be determined as discussed above. If a leave or disconnect event occurs the member is removed but no new view is delivered. Since the member is removed it will not be sent as part of the GROUPS

message as discussed below and so will not be in the new membership or new transitional set.

If TMemb = RMemb, then the new set of servers will be a subset of the previous set, so once the partitioned members are removed from the members set the transitional set will equal the member set and so any q will be in the transitional set.

In the second case, where servers merged as well as partitioned, the transitional set is computed by each server as the union of all of the member sets sent in a GROUPS message with the same previous view id. In this case, by Lemma 6.15 all of the members sent in a GROUP message were of the ESTABLISHED state at the sender. Therefore, the transitional set made up of the member lists whose previous view has the same view as our previous view will be the set of established members of our previous view. Since the only servers who will have a member set with the same previous view id is one who was actually with us previously, because if they did not then that would violate Theorem 6.3, this set of members will include everyone who is being installed in the current view and was also in the previous one.

Property d If processes p and q install the same view v' in the same previous view v, then they have the same transitional sets in v'

This consists of showing that no 'extra' processes are included in the transitional sets. The potential 'extra' processes are those who are not required by Property 3. These extra processes are not possible because in the algorithm only those processes who were in ESTABLISHED state and so were members of a previous view, are included. Servers who install the same view in the same previous view by the Virtual Synchrony property of EVS-S will see the same set of messages so the joins and leaves will be the same. Thus, the transitional sets will be the same. □

THEOREM 6.19 (NO DUPLICATION)
A process never delivers a message more then once.

PROOF: The group membership algorithm never creates new data messages, only JOIN, LEAVE, and KILL messages which are never delivered to an application. The data messages are delivered as soon as the **sdeliver** event occurs in any state. Therefore, no duplicates are delivered because of the EVS-S No Duplication property. □

THEOREM 6.20 (DELIVERY INTEGRITY)
*A **deliver** event in a view is the result of a preceding send event.*

PROOF: The only **deliver** events are generated for messages for which an **sdeliver** event occured with the message tagged DATA. In no other cases will messages be delivered to the application. The only messages tagged as DATA messages are those **ssend** in response to a **send** event. Since EVS-S also guarantees that every **sdeliver** event is the result of a preceeding **ssend** event, every **deliver** event is the result of a preceeding **send** event. □

THEOREM 6.21 (SELF DELIVERY)
If a process p sends a message m, and p is a member of g and does not leave group g, then m is delivered to p unless p crashes.

PROOF: Since m will generate a **ssend** event, we are guaranteed by the EVS-S Self Delivery property that the server will receive a **sdeliver** event for a DATA message with contents m unless the server crashes. When the **sdeliver** event occurs, the message m will be delivered to every local client who is a member of the group g. Since p is a member of group g prior to sending and does not leave the group, p will still be a member of the group as the algorithm executing at server s only removes someone connected to itself from a group in response to a **leave** or **disconnect** event, never as a result of a **sview** or **stransig** event. In this case no **leave** or **disconnect** event occurs, so the message will be delivered to p at the time it is **sdelivered**. □

THEOREM 6.22 (SANE VIEW DELIVERY)

(a). A message m with $m.type \in \{A, S\}$ is not delivered in a view earlier than the one in which it was sent.

(b). If a process p sends a message m, crashes and later recovers in a view v and a process q delivers m, then m is delivered in a view before v.

PROOF:

Property a If a message m is sent in view v with id, **sdeliver**(s, m) occurs with vid(m) = id', for a proof by contradiction assume that id' ¡ id.

Then the view with identifier id' can either be the result of a single **join**, **leave** or **disconnect** event, or it can be the result of an **sview** event.

In the first case, since id' ¡ id, the agreed message m' that is sent in response to the **join**, **leave**, or **disconnect** event must have been **sdelivered** prior to whatever triggered view v (an agreed message m" or **sview** event). So $ord(m) > ord(m'')$ because ord is consistent with causality and m was sent in v, which was triggered by m"; $ord(m'') > ord(m')$ because id ¿ id'. Since the vid(m) = id' and the view id' was generated by message m', no other message that generates a view could have been ordered between m' and m, because then the vid(m) != id'. So, specifically, the view v generated by message m" must either have been prior to m' which contradicts the assumption that id' ¡ id, or v must be ordered after m is delivered which implies $ord(m'') > ord(m)$ which contradicts the above statement that $ord(m) > ord(m'')$. The only other case is where view v was generated by an **sview** event, but that contradicts the Sane View Deliver for EVS-S.

In the second case where an **sview** event was the cause, id' ¡ id contradicts Sane View Delivery for EVS-S as the message m would be delivered in an **sview** prior to the **sview** in which it was sent. So $id' \geq id$.

Property b Assume m is the last message the server receives from p before detecting the crash. Since the goal is to prove that the message is delivered prior to an event, any earlier message will also be delivereed before v if m is delivered. Upon receiving a **disconnect**

event, the server will generate a KILL message and send it to all of the servers as an Agreed order message. By Theorem 6.12 after an Agreed message, such as the KILL is deliverd, no message with a lower ord value (such as m) will be delivered. Therefore, if m is delivered at some process q, then m is delivered prior to the KILL message. When the server **`sdelivers`** the KILL message a new view v' will be generated

The **`disconnect`** event is guaranteed to be prior to any **`join`** event p initiates subsequent to recovering from its crash because prior to the **`disconnect`** being processed at s (the server p was connected to previously and to whom it is reconnecting) the **`connect`** event of p will be rejected as a duplicate client. So s must have processed the **`disconnect`** event and the view in which p will reconnect will be higher then the view generated by the **`disconnect`** of p. □

THEOREM 6.23 (FIFO MESSAGES)
If a process sends a FIFO message after sending a previous message then all processes who deliver both messages deliver them in the order in which they were sent.

PROOF: This is a direct result of Lemma 6.9. □

THEOREM 6.24 (CAUSAL MESSAGES)
If a process sends a causal message m' such that the send of another message m causally precedes the send of m', then any process that delivers both messages delivers m before m'.

PROOF: This is a direct result of Lemma 6.9. □

THEOREM 6.25 (AGREED MESSAGES)

(a). Agreed messages are Causal messages.

(b). If a process p delivers an agreed message m, then after that event it will never deliver a message that has a lower ord value.

(c). If a process p delivers an agreed message m' before a transsig event in its current view, then p delivers every message with a lower ord value than m' delivered in that view by any process.

(d). *If a process p delivers an agreed message m' after a transsig event in its current view, then p delivers every message with a lower ord value then m' sent by any process in p's next transitional set that were delivered in the same view as m'.*

PROOF: Theorem 6.25(b) results directly from Lemma 6.9.

Part b of Theorem 6.25 is a consequence of the same Agreed Message property of the EVS-S specification and the fact that a **`transsig`** event is delivered immediately to any group who is effected by a **`stranssig`** event. Therefore, any message delivered prior to the **`stranssig`** event will also be delivered to this algorithm and this algorithm delivers every data message it receives immediately.

To prove part c of Theorem 6.25 one observes that the transistional set of of a group never contains any member who was not connected to a server in the TMemb delivered in the **`stranssig`** event. By the Agreed Messages property of EVS-S, the server will receive all of the messages sent by someone in the TMemb, and thus, someone in the transitional set of the group. □

THEOREM 6.26 (SAFE MESSAGES)

(a). *Safe messages are agreed messages.*

(b). *If a process p delivers a safe message m before a transsig event in its current view v, then every member of that view delivers m, unless that member crashes in v.*

(c). *If a process p delivers a safe message m after a transsig event in its current view v, then every member of p's transitional set from p's next view delivers m, unless a member crashes in v.*

PROOF: Both of these properties are a consequence of the Safe Message property of EVS-S and the fact that once a **`stranssig`** event has occured, this algorithm never delivers any messages that are not subsequent to a **`transsig`** event for any group that experiences changes. As shown in the handling a JOIN message, when a **`view`** event is delivered, a **`transsig`** event is immediately generated following it. Therefore, any subsequent messages will be delivered after a **`transsig`** event.

The algorithm never drops messages, fails to deliver them or reorders them. As a result the safe message will be delivered to everyone in the group at the time it arrives, including the same set of join, leave, or disconnect group changes because they are agreed messages. □

THEOREM 6.27 (TRANSITIONAL SIGNALS)

(a). At most one transsig event occurs at a process during a view.

(b). If two processes p and q are virtually synchronous in a view v, and p has a transsig event occur in v, then q also has a transsig event occur in v, and they both deliver the same sets of agreed messages before and after their transsig events in v.

PROOF: The proof of Theorem 6.27(a) requires that once a **`transsig`** event is delivered, no other **`transsig`** event will be delivered until a **`view`** event is delivered to that process. A **`transsig`** event can be generated in two places in the algorithm. First, in GOP state when a **`stranssig`** event is received. Second, in GTRANS state when a JOIN message is received. In the join case, a **`view`** is delivered for the group immediately before generating the **`transsig`** event so there is no possibility that the delivered **`transsig`** is the second one in a view. Therefore, the only way a second **`transsig`** event is delivered to a group prior to a **`view`** is if the server twice receives a **`stranssig`** event in GOP state, or receives a **`stranssig`** event in GOP subsequent to a JOIN message in GTRANS without an intervening **`view`** event.

Both cases are impossible because of Lemma 6.16 because the server leaves GOP state immediately after delivering the **`transsig`** event or because the server was not in GOP state in the case of the JOIN, so it must deliver a **`view`** event for the group prior to enter GOP state.

The second part of Theorem 6.27 requires that every two processes that are virtually synchronous both deliver the transitional signal in the same view and deliver the same sets of agreed messages prior and subsequent to the transitional signal.

By the Virtual Synchrony property of groups, both processes will deliver the same set

of messages in the previous view.

The transitional signal can be delivered because of two different causes: first, as a result of a **stranssig** event, and second, as a result of a JOIN message in GTRANS state. In either case, since both processes install the same, next view by the EVS-S properties they will both deliver a **transsig** event and they will have seen the same set of messages prior to receiving the **stransig** or JOIN message. Therefore, they will deliver the same set of messages prior to the **transsig**. After the **transsig** they will be in each other's transitional set as shown in Theorem 6.18. They must have been in each others TMemb set since otherwise they would have been removed each other during the **stranssig** event handling. The **view** event would not have them in each others transitional set. □

Chapter 7

Conclusion

This thesis presented an architecture, algorithms, and a complete implementation of a high-performance, wide-area group communication system. The goal of this work was to develop the necessary network protocols and distributed algorithms to provide practical and usable group services on wide-area networks.

Presented in this thesis is an innovative architecture for building scalable group-oriented services including a hierarchical network model that minimizes costs on the wide-area networks. It also includes the developement of an overlay network approach to constructing a group-communication system and efficient, customized network protocols to create the overlay network and disseminate messages efficiently.

These services are useful to a wide range of applications, not only traditional fault-tolerant applications, but also collaboration, cluster management, message-oriented middleware, and distributed application servers. The set of services provided include light-weight messaging such as unordered reliable and FIFO reliable messages as well as strongly ordered messages such as agreed order or safe delivery that simplify the development of fault-tolerant applications.

The predominant characteristic of modern distributed systems is change. Whether the change is the evolution of the software infrastructure, or changing scalability requirements,

or a changing network environment, a distributed system must successfully adapt to change. The Spread wide-area group communication toolkit helps distributed applications adapt to changes both in the network and in the available resources.

References

[ABCD96] Yair Amir, D. Breitgand, Gregory Chockler, and Danny Dolev. Group communication as an infrastructure for distributed system management. In *Proceedings of 3rd International Workshop on Services in Distributed and Networked Environment (SDNE)*, pages 84–91, June 1996.

[ABKM01] David Andersen, Hari Balakrishnan, M. Frans Kaashoek, and Robert Morris. Resilient overlay networks. In *Proceedings of the 18th ACM Symposium on Operating System Principles (SOSP '01)*, pages xx–xx, Chateau Lake Louise, Banff, Canada, October 2001.

[ACBMT95] Emmanuelle Anceaume, Bernadette Charron-Bost, Pascale Minet, and Sam Toueg. On the formal specification of group membership services. Technical Report 95–1534, Department of Computer Science, Cornell University, August 1995.

[ACDK97] Tal Anker, Gregory Chockler, Danny Dolev, and Idit Keidar. The Caelum toolkit for CSCW: the sky is the limit. In *Proceedings of the Third International Workshop on Next Generation Information Technologies and Systems (NGITS 97)*, pages 69–76, June 1997.

[ACDK98] Tal Anker, Gregory V. Chockler, Danny Dolev, and Idit Keidar. Scalable group membership services for novel applications. In Marios Mavronicolas, Michael Merritt, and Nir Shavit, editors, *Proceedings of the workshop on Networks in Distributed Computing*, DIMACS Series in Discrete Mathematics and Theoretical Computer Science, 1998.

[ADK99] Tal Anker, Danny Dolev, and Idit Keidar. Fault tolerant video-on-demand services. In *Proceedings of 19th International Conference on Distributed Computing Systems*, pages 244–252, June 1999.

[ADKM92] Yair Amir, Danny Dolev, S. Kramer, and Dalia Malki. Transis: A communication subsystem for high-availability. In *Digest of Papers, The 22nd International Symposium on Fault-Tolerant Computing Systems*, pages 76–84. IEEE Computer Society Press, Los Alamitos, CA, 1992.

[ADMSM94] Yair Amir, Danny Dolev, P.M. Melliar-Smith, and L.E. Moser. Robust and efficient replication using group communication. Technical Report CS94–

20, Institute of Computer Science, The Hebrew University of Jerusalem, Jerusalem, Israel, 1994.

[ADS00] Yair Amir, Claudiu Danilov, and Jonathan Stanton. A low latency, loss tolerant architecture and protocol for wide area group communication. In *Proceedings of the International Conference on Dependable Systems and Networks*, pages 327–336. IEEE Computer Society Press, Los Alamitos, CA, June 2000. FTCS 30.

[Aga94] Deborah A. Agarwal. *Totem: A Reliable Ordered Delivery Protocol for Interconnected Local-Area Networks*. PhD thesis, University of California, Santa Barbara, 1994.

[Ami95] Yair Amir. *Replication using Group Communication over a Partitioned Network*. PhD thesis, Institute of Computer Science, The Hebrew University of Jerusalem, Jerusalem, Israel, 1995.

[AMMS+95] Yair Amir, L. E. Moser, P. M. Melliar-Smith, D.A. Agarwal, and P. Ciarfella. The totem single-ring ordering and membership protocol. *ACM Transactions on Computer Systems*, 13(4):311–342, November 1995.

[AMMSB98] D.A. Agarwal, L. E. Moser, P. M. Melliar-Smith, and R. K. Budhia. The totem multiple-ring ordering and topology maintenance protocol. *ACM Transactions on Computer Systems*, 16(2):93–132, May 1998.

[AS98] Yair Amir and Jonathan Stanton. The Spread wide area group communication system. Technical Report 98–4, Johns Hopkins University, Center for Networking and Distributed Systems, 1998.

[ASAWM99] Ehab S Al-Shaer, Hussein Abdel-Wahab, and Kurt Maly. HiFi: A new monitoring architecture for distributed system management. In *Proceedings of 19th International Conference on Distributed Computing Systems*, pages 171–178, June 1999.

[ASSS01] Yair Amir, George Schlossnagle, Theo Schlossnagle, and Jonathan Stanton. Distributed logging using group communications. Technical Report CNDS2001-xx, Center for Networking and Distributed Systems, The Johns Hopkins Univeristy, October 2001. More information available at http://www.backhand.org/.

[AT01] Yair Amir and Ciprian Tutu. From total order to database replication. Technical Report 2001-6, Center for Networking and Distributed Systems, Johns Hopkins University, November 2001.

[BDGB94] Özalp Babaoğlu, Renzo Davoli, Luigi-Alberto Giachini, and Mary Gray Baker. RELACS: A communication infrastructure for constructing reliable applications in large-scale distributed systems. Technical Report UBLCS94-15, Laboratory of Computer Science, University of Bologna, 1994.

[BDM99] Özalp Babaoğlu, Renzo Davoli, and Alberto Montresor. Group communication in partitionable systems: Specifications and algorithms. Technical Report UBLCS-98-01, Department of Computer Science, University of Bologna, Mura Anteo Zamboni 7
40127 Bologna, Italy, October 1999. Originally published April 1998.

[BDMS98] Özalp Babaoğlu, Renzo Davoli, Alberto Montresor, and R. Segala. System support for partition aware network applications. In *Proceedings of 18th International Conference on Distributed Computing Systems*, pages 184–191, Amsterdam, The Netherlands, May 1998.

[BFHR98] Ken Birman, Roy Friedman, Mark Hayden, and Injong Rhee. Middleware support for distributed multimedia and collaborative computing. In *Multimedia Computing and Networking (MMCN98)*, 1998.

[Bir86] Kenneth Birman. ISIS: A system for fault-tolerant distributed computing. Technical Report TR86-744, Department of Computer Science, Cornell University, April 1986.

[Bir93a] Kenneth Birman. The process group approach to reliable distributed computing. *Communications of the ACM*, 36(12):36–53, December 1993.

[Bir93b] Kenneth Birman. A response to Cheriton and Skeen's criticism of causal and totally ordered communication. Technical Report 93-1390, Department of Computer Science, Cornell University, 1993.

[Bir96] Keneth P. Birman. *Building Secure and Reliable Network Applications*. Manning, 1996.

[BJ87] K. P. Birman and T. Joseph. Exploiting virtual synchrony in distributed systems. In *11th Annual Symposium on Operating Systems Principles*, pages 123–138, November 1987.

[BR94] K. P. Birman and Robbert Van Renesse. *Reliable Distributed Computing with the Isis Toolkit*. IEEE Computer Society Press, March 1994.

[CHRC97] Sarah Chodrow, Michael Hircsh, Injong Rhee, and Shun Yan Cheung. Design and implementation of a multicast audio conferencing tool for a collaborative computing framework. *JCIS*, March 1997.

[CHTCB96] Tushar Deepak Chandra, VAssos Hadzilacos, Sam Toueg, and Bernadette Charon-Bost. On the impossibility of group membership. In *Proceedings of the 15th ACM Symposium on Principles of Distributed Computing*, pages 322–330, May 1996.

[CS93] D.R. Cheriton and D. Skeen. Understanding the limitations of causally and totally ordered communications. In *Proceedings of the 11th ACM Symposium on Operating System Principles*, December 1993.

[CT96] Tushar Deepak Chandra and Sam Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, 43(2):225–267, March 1996.

[DMS96] Danny Dolev, Dalia Malki, and R. Strong. A framework for partitionable membership service. In *Proceedings of the ACM Symposium on the Principles of Distributed Computing*, May 1996.

[EMS95] Paul Ezhilchelvan, Raimundo Macêdo, and Santosh Shrivastava. Newtop: a fault-tolerant group communication protocol. In *Proceedings of the 15th International Conference on Distributed Computing Systems*, June 1995.

[FJL+97] Sally Floyd, Van Jacobson, C. Liu, Steve McCanne, and L. Zhang. A reliable multicast framework for light-weight sessions and application level framing. *IEEE/ACM Transactions on Networking*, 5(6):784–803, December 1997.

[Flo62] Robert W. Floyd. Algorithm 97 (shortest path). *Communications of the ACM*, 5(6):345, 1962.

[FLP85] M.J. Fischer, Nancy Lynch, and M.S. Patterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32(2):374–382, April 1985.

[FLS97] A. Fekete, N. Lynch, and A. Shvartsman. Specifying and using a partionable group communication service. In *Proceedings of the 16th annual ACM Symposium on Principles of Distributed Computing*, pages 53–62, Santa Barbara, CA, August 1997.

[FLS01] Alan Fekete, Nancy Lynch, and Alex Shvartsman. Specifying and using a partitionable group communication service. *ACM Transactions on Computer Systems*, 19(2):171–216, May 2001.

[Fri95] Roy Friedman. Using virtual synchrony to develop efficient fault tolerant distributed shared memories. Technical Report TR95-1506, Department of Computer Science, Cornell University, March 1995.

[FV97a] Roy Friedman and Alexy Vaysburg. Fast replicated state machines over partitionable networks. In *16th IEEE International Symposium on Reliable Distributed Systems(SRDS)*. IEEE, October 1997.

[FV97b] Roy Friedman and Alexy Vaysburg. High-performance replicated distributed objects in partitionable environments. Technical Report TR97-1639, Dept. of Computer Science, Cornell University, Ithaca, NY 14850, USA, July 1997.

[FvR95] R. Friedman and R. van Renesse. Strong and weak Virtual Synchrony in Horus. Technical Report 95–1537, Cornell University, Computer Science, August 1995.

[FvR97] Roy Friedman and Robbert van Renesse. Packing messages as a tool for boosting the performance of total ordering protocols. In *6th IEEE International Symposium on High Performance Distributed Computing*, pages xx–xx, 1997. Also available as Technical Report 95-1527, Department of Computer Science, Cornell University.

[GGLA97] L. Gu and J.J. Garcia-Luna-Aceves. New error recovery structures for reliable networking. In *Proceedings of the Sixth International Conference on Computer Communications and Networking*, September 1997.

[GS95] R. Guerraoui and André Schiper. Transaction model vs virtual synchrony model: bridging the gap. In *Theory and Practice in Distributed Systems*, volume LNCS938, pages 121–132. Springer-Verlag, September 1995.

[GVvR96] Katherine Guo, Werner Vogels, and Robbert van Renesse. Structured virtual synchrony: Exploring the bounds of virtual synchronous group communication. In *7th ACM SIGOPS European Workshop*, September 1996.

[Hay98] Mark Hayden. *The Ensemble System*. PhD thesis, Cornell University, 1998.

[hCRSZ01] Yang hua Chu, Sanjay G. Rao, Srinivasan Seshan, and Hui Zhang. Enabling conferencing applications on the internet using an overlay multicast architecture. In *Proceedings of ACM SIGCOMM 2001*, pages 55–67, San Diego, CA, August 2001.

[HLvR99] Jason Hickey, Nancy Lynch, and Robbert van Renesse. Specifications adn proofs for Ensemble layers. In *Fifth International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 99)*, Lecture Notes in Computer Science, Amsterdam, the Netherlands, March 1999. Springer-Verlag.

[Hof96] Markus Hofmann. A generic concept for large-scale multicast. In B. Plattner, editor, *International Zurich Seminar on Digital Communications*, number 1044 in Lecture Notes in Computer Science, pages 95–106. Springer-Verlag, Februrary 1996.

[Hul96] N. Huleihel. Efficient ordering of messages in wide area networks. Master's thesis, Institute of Computer Science, The Hebrew University of Jerusalem, Jerusalem, Israel, 1996.

[HvR95] Takako M. Hickey and Robbert van Renesse. Incorporating system resource information into flow control. Technical Report TR 95-1489, Department of Computer Science, Cornell University, Ithaca, NY, 1995.

[JFR93] Farnam Jahanian, S. Fakhouri, and R. Rajkumar. Processor group membership protocols: Specification, design and implementation. In *Proceedings of 12th IEEE International Symposium on Reliable Distributed Systems (SRDS)*, pages 2–11. IEEE, October 1993.

[JJS99] Scott Johnson, Farnam Jahanian, and Jigney Shah. The inter-group router approach to scalable group composition. In *Proceedings of 19th International Conference on Distributed Computing Systems*, pages 4–14, Austin, Texas, June 1999.

[KA98] Bettina Kemme and Gustavo Alonso. A suite of database replication protocols based on group communication primitives. In *Proceedings of 18th International Conference on Distributed Computing Systems*, Amsterdam, The Netherlands, May 1998.

[KA00] Bettina Kemme and Gustavo Alonso. A new approach to developing and implementing eager database replication protocols. *ACM Transactions on Database Systems*, 25(3):333–379, September 2000.

[KCH98] Alan Krantz, Sarah Chodrow, and Michael Hircsh. Design and implementation of a distributed X multiplexor. In *Proceedings of 18th International Conference on Distributed Computing Systems*, pages xx–xx, Amsterdam, The Netherlands, May 1998.

[KD96] Idit Keidar and Danny Dolev. Efficient message ordering in dynamic networks. In *Proceedings of 15th ACM Symposium on Principles of Distributed Computing (PODC)*, pages 68–76, May 1996.

[Kei94] Idit Keidar. A highly available paradigm for consistent object replication. Master's thesis, Institute of Computer Science, The Hebrew University of Jerusalem, Jerusalem, Israel, 1994. Also HUJI Technical Report CS95-5.

[KFL98] Roger Khazan, Alan Fekete, and Nancy Lynch. Multicast group communication as a base for a load-balancing replicated data service. In *Proceedings of 12th International Symposium on DIStributed Computing(DISC)*, pages 258–272, Andros, Greece, September 1998.

[KK00] Idit Keidar and Roger Khazan. A client-server approach to virtually synchronous group multicast: Specifications and algorithms. In *Proceedings of the 20th IEEE International Conference on Distributed Computing Systems*, pages 344–355, Taipei, Taiwan, April 2000. IEEE Computer Society Press, Los Alamitos, CA.

[KSMD99] Idit Keidar, Jeremy Sussman, Keith Marzullo, and Danny Dolev. Moshe: A group membership service for WANs. Technical Report MIT-LCS-TM-593a, Laboratory for Computer Science, MIT, 1999. Also U. California San Diego TR CS99–623a; revised September 2000.

[KSMD00] Idit Keidar, Jeremy Sussman, Keith Marzullo, and Danny Dolev. A client-server oriented algorithm for virtually synchronous group membership in WANs. In *Proceedings of the 20th IEEE International Conference on Distributed Computing Systems*, pages 356–365, Taipei, Taiwan, April 2000. IEEE Computer Society Press, Los Alamitos, CA.

[KT91] Franz Kaashoek and Andy Tannenbaum. Group communications in the Amoeba distributed operating system. In *Proceedings of the IEEE Symposium on Reliable Distributed Systems*, pages 222–230, Arlington, Texas, May 1991.

[Lam78] Leslie Lamport. Time, clocks and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, 1978.

[LP96] J.C. Lin and S. Paul. RMTP: A reliable multicast transport protocol. In *Proceedings of IEEE Infocom*, pages 1414–1424, March 1996.

[LT87] Nancy A. Lynch and Mark R. Tuttle. Hierarchical correctness proofs for distributed algorithms. Master's thesis, Massachusetts Institute of Technology, April 1987.

[Lyn96] Nancy A. Lynch. *Distributed Algorithms*. Morgan Kaufmann, 1996.

[Maf95] S. Maffeis. Adding group communication and fault-tolerance to CORBA. In *Proceedings of the 1st USENIX Conference on Object-Oriented Technologies adn Systems (COOTS)*, pages xx–xx, Monterey, CA, June 1995.

[MAMSA94] L. E. Moser, Yair Amir, P. M. Melliar-Smith, and D. A. Agarwal. Extended virtual synchrony. In *Proceedings of the IEEE 14th International Conference on Distributed Computing Systems*, pages 56–65. IEEE Computer Society Press, Los Alamitos, CA, June 1994.

[MDB00] Alberto Montresor, Renzo Davoli, and Özalp Babaoğlu. Enhancing Jini with group communication. Technical Report UBLCS-2000-16, Dept. of Computer Science, University of Bologna, December 2000. revised January 2001.

[MFSW95] C. P. Malloth, P. Felber, André Schiper, and U. Wilhelm. Phoenix: A toolkit for building fault-tolerant distributed applications in large scale. In *Workshop on Parallel and Distributed Platforms in Industrial Products*, San Antonio, Texas, October 1995. Workshop held during the 7th IEEE Symposium on Parallel and Distributed Processing. (SPDP-7).

[MMSN98] L.E. Moser, P.M. Melliar-Smith, and P. Narasimhan. Consistent object replication in the Eternal system. *Distributed Systems Engineering*, 4(2):81–92, January 1998.

[Mon00] Alberto Montresor. *System Support for Programming Object-Oriented Dependable Applications in Partitionable Systems*. PhD thesis, Dept. of Computer Science, University of Bologna, Februrary 2000.

[MP99] Shivakant Mishra and G. Pang. Design and implementation of an availability management service. In *Proceedings of 19th International Conference on Distributed Computing Systems Workshop on Middleware*, pages 128–133, June 1999.

[MPS93] Shivakant Mishra, Larry L. Peterson, and Richard D. Schlichting. Consul: A communication substrate for fault-tolerant distributed programs. *Distributed System Engineering*, 1:87–103, December 1993.

[NB97] Jorg Nonnenmacher and Ernst W. Biersack. Performance modelling of reliable multicast transmission. In *Proceedings of INFOCOM 97*. IEEE Computer Society, April 1997.

[Now98] Ariel Nowersztern. MOSHE Membership Object-oriented Service for Heterogeneous Environments. Lab project, The Hebrew University of Jerusalem, Israel, Februrary 1998. http://www.cs.huji.ac.il/ reln.

[Pow91] D. Powell. *Delta-4: A Generic Architecture for Dependable Distributed Computing*. Springer-Verlag, 1991.

[RBM96] Robbert Van Renesse, Kenneth Birman, and S. Maffeis. Horus: A flexible group communication system. *Communications of the ACM*, 39(4):76–83, April 1996.

[RCHS97] Injong Rhee, Shun Yan Cheung, P. Hutto, and V. Sunderam. Group communication support for distributed multimedia and CSCW systems. In *Proceedings of 17th International Conference on Distributed Computing Systems*, pages xxx–xxx, xxx 1997.

[RFV96] Luis E.T. Rodrigues, H. Fonseca, and Paulo. Verissimo. A dynamic hybrid protocol for total order in large-scale systems. In *Proceedings of the 16th International Conference on Distributed Computing Systems*, May 1996. Selected portions published in.

[RGS+96] Luis Rodrigues, Katherine Guo, A. Sargento, Robbert van Renesse, B. Glade, Paulo Verissimo, and Ken Birman. A dynamic light-weight group service. In *15th IEEE International Symposium on Reliable Distributed Systems (SRDS)*, pages 23–25, October 1996. Also Cornell University Technical Report TR96-1611, August 1996.

[RV92] Luis Rodrigues and Paulo Verissimo. xAMp: a multi-primitive group communications service. In *Proceedings of the 11th Symposium on Reliable Distributed Systems*, Houston, Texas, October 1992.

[Sch90] Fred B. Schneider. Implementing fault tolerant services using the state machine approach: A tutorial. *Computing Surveys*, 22(4):299–319, December 1990.

[SCH+99] Stefan Savage, Andy Collins, Eric Hoffman, John Snell, and Thomas Anderson. The end-to-end effects of internet path selection. In *Proceedings of ACM SIGCOMM 1999*, August 1999.

[Sch01] John Schultz. Partitionable virtual synchrony using extended virtual synchrony. Master's thesis, Department of Computer Science, Johns Hopkins University, January 2001.

[SM98] Jeremy Sussman and Keith Marzullo. The *bancomat* problem: An example of resource allocation in a partitionable asynchronous system. In *Proceedings of 12th International Symposium on Distributed Computing (DISC)*, September 1998. Full version available as Univ. Calfornia, San Diego Dept. of Computer Science Tech Report 98-570.

[SR96] André Schiper and Michel Raynal. From group communication to transactions in distributed systems. *Communications of the ACM*, 39(4):84–87, April 1996.

[TMMS97] E. Thomopoulos, L. E. Moser, and P. M. Melliar-Smith. Analyzing the latency of the Totem multicast protcols. In *Proceedings of the Sixth International Conference on Computer Communications and Networks*, pages 42–50. IEEE Computer Society Los Alamitos, CA, September 1997.

[WLF00] David Watson, Yan Luo, and Brett D. Fleisch. The oasis+ dependable distributed storage system. In *Proceedings of the 2000 Pacific Rim International Symposium on Dependable Computing*, pages xx–xx, Los Angeles, CA, December 2000.

[WLF01] David Watson, Yan Luo, and Brett D. Fleisch. Experiences with oasis+: A fault tolerant storage system. In *Proceedings of the IEEE International Conference on Cluster Computing*, pages xx–xx, Newport Beach, CA, October 2001.

[WMK94] B. Whetten, T. Montgomery, and S. Kaplan. A high performance totally ordered multicast protocol. In *Theory and Practice in Distributed Systems, International Workshop*, Lecture Notes in Computer Science, page 938, September 1994.

[WS95] U. Wilhelm and André Schiper. A hierarchy of totally ordered multicasts. In *Proceedings of Fourteenth IEEE International Symposium on Reliable Distributed Systems (SRDS)*, September 1995.

Curriculum Vita

Department of Computer Science
The Johns Hopkins University
3400 North Charles Street
Baltimore, MD 21218

jonathan@cs.jhu.edu
http://www.cnds.jhu.edu/~jonathan/
Phone: (410) 516-0454 (Office)
Fax: (410) 516-6134

RESEARCH INTERESTS

Distributed systems, network protocols, distributed security.

EDUCATION

2002	Ph.D. in Computer Science	The Johns Hopkins University, *Practical Wide Area Group Communication.* Adviser: Prof. Yair Amir.
1998	M.S.E. in Computer Science	The Johns Hopkins University.
1995	B.A. in Mathematics	Cornell University.

WORK EXPERIENCE

Academic

Sept. 2001 – present	Assistant Research Professor. Johns Hopkins University.
Sept. 1996 – Aug. 2001	Research Assistant. Johns Hopkins University.
July 1995 – Aug. 1996	Teaching Assistant. Johns Hopkins University.
Sept. 1994 – May 1995	Teaching Assistant. Cornell University.
May 1994 – Sept. 1994	Instructor. Cornell University.

Non-Academic

Sept. 2000 – present	Co-Founder and Officer. Spread Concepts LLC.

GRANTS

- Co-PI, "A Cost-Benefit Approach to Fault Tolerant Communication and Information Access," DARPA BAA 00-01, $944,015, May 2000 – April 2003 (with Yair Amir (Co-PI) and Baruch Awerbuch (Co-PI)).

- Co-PI, "High Performance, Robust and Secure Group Communications," DARPA BAA 99-33, $1,350,824 of which $450,000 is co-funded by the NSA, May 2000 – April 2003 (with Yair Amir (Co-PI), Baruch Awerbuch (Co-PI), and Gene Tsudik as a sub-contractor from UC Irvine).

RESEARCH AND PUBLICATIONS

My research focuses on designing, exploring, and building scalable, high-performance, reliable distributed systems that take advantage of wide-area networks. I am one of the creators of the Spread system that is described below. I help lead or participate in a number of distributed systems developed in the Center for Networking and Distributed Systems at Johns Hopkins University. These include a secure group communication system supporting privacy, integrity, and authentication; a scalable, efficient replication tool for databases, and high availability tools for clusters. I have also worked on custom network protocols and sophisticated flow-control models for application-level overlay networks.

Released Software

- Spread – I am the chief architect of the Spread wide-area group messaging toolkit (http://www.spread.org/). Currently, Spread is believed to have thousands of users in commercial, research, and teaching environments. Several popular applications use Spread, such as the Apache-SSL secure web server, a distributed logging service for the Apache and thttpd web servers, the native database replication in Postgres, and the Zope application server. Spread is part of FreeBSD and is included in some Linux distributions. We have recorded over 2500 distinct downloads from our web site. Spread has an active developer and user community from around the world.

Refereed Conference Proceedings

"Flow Control for Many-to-Many Multicast: A Cost-Benefit Approach," Yair Amir, Baruch Awerbuch, Claudiu Danilov, and Jonathan Stanton, *To appear in Proceedings of Fifth IEEE Conference on Open Architectures and Network Programming, New York, New York*, June 28-29, 2002.

"Framework for Authentication and Access Control of Client-Server Group Communication Systems," Yair Amir, Cristina Nita-Rotaru, and Jonathan Stanton, *Third International Networked Group Communications Workshop, London, UK.* November 7-9, 2001, Published in LNCS 2233, 120-128.

"Exploring Robustness in Group Key Agreement," Yair Amir, Yongdae Kim, Cristina Nita-Rotaru, John Schultz, Jonathan Stanton, and Gene Tsudik, *21th IEEE International Conference on Distributed Computing Systems (ICDCS), Phoenix, Arizona,* April 16-19, 2001, 399-408.

"A Low Latency, Loss Tolerant Architecture and Protocol for Wide Area Group Communication," Yair Amir, Claudiu Danilov, and Jonathan Stanton, *International Conference on Dependable Systems and Networks (ICDSN) (previously FTCS-30), New York, New York*, June 25-28, 2000, 327-336.

“Secure Group Communication in Asynchronous Networks with Failures: Integration and Experiments,” Yair Amir, Giuseppe Ateniese, Damian Hasse, Yongdae Kim, Cristina Nita-Rotaru, Theo Schlossnagle, John Schultz, Jonathan Stanton, and Gene Tsudik, *20th IEEE International Conference on Distributed Computing Systems (ICDCS), Taipei, Taiwan,* April 10-13, 2000, 330-343.

Technical Reports

“Practical Wide-Area Database Replication,” Yair Amir, Claudiu Danilov, Michal Miskin-Amir, Jonathan Stanton, and Ciprian Tutu, *Technical Report CNDS-2002-1, submitted to a conference.*

“Robust Contributory Key Agreement in Secure Spread,” Yair Amir, Yongdae Kim, Cristina Nita-Rotaru, John Schultz, Jonathan Stanton, and Gene Tsudik, *submitted to a journal.*

“The Spread Wide Area Group Communication System,” Yair Amir and Jonathan Stanton, *Technical Report CNDS-98-4.*

Professional Service

- Member of the Program Committee for the *IEEE International Conference on Distributed Computing Systems*, Vienna, Austria July 2002.
- Reviewer for DISC 1998, FTCS 1999, ICDCS 2000, 2001, IPDPS 2002, Journal of Parallel and Distributed Computing.
- Served on Graduate Admissions Committee for Johns Hopkins Computer Science Department in 1998.

Presentations and Lectures

Workshops

- Invited Speaker at the Spread Workshop June 2001.
- Invited Panelist at IEEE ICDCS April 2001 Workshop on Applied Reliable Group Communication.

Conference Lectures

- Networked Group Communication Workshop, London, November 2001.
- International Conference on Dependable Systems and Networks, New York City, July 2000.
- International Conference on Distributed Computing Systems, Taipei, March 2000.

Other Presentations

- Fault Tolerant Networks PI meeting, San Diego, January 2002.
- Johns Hopkins Information Security Institute Open House, Baltimore, October 2001.
- Dynamic Coalition PI meeting, Colorado Springs, July 2001.

Teaching

Johns Hopkins University.

- Instructor for Distributed Systems, Fall 2001.
- Co-instructor for Distributed Systems, Fall 1999, Fall 2000.

  Distributed Systems is a senior undergraduate and graduate course with 40 students. It covers the theory of distributed algorithms, how distributed systems are built and used today, and includes programming assignments and a final project.

- Teaching Assistant for Parallel Algorithms, Spring 1996.
- Teaching Assistant for Computer Architecture, Fall 1995.
- Volunteer intercollegiate debate coach for Cornell University and Towson University from Fall 1995 until Spring 1997.

Cornell University.

- Teaching Assistant for Introduction to Programming, Fall 1994, Spring 1995.
- Instructor in basic computer and web use as well as author of instructional materials, Summer 1994.
- Assistant Instructor for Introduction to Debate in the Communications Department, Fall 1993.