The InterGroup Protocols: Scalable Group Communication for the Internet

Karlo Berket[1]

December 4, 2000

[1]This work was supported by the Director, Office of Science. Office of Advanced Scientific Computing Research. Mathematical, Information, and Computational Sciences Division, U.S. Department of Energy under Contract No. DE-AC03-76SF00098.

UNIVERSITY OF CALIFORNIA
Santa Barbara

The InterGroup Protocols: Scalable Group Communication for the Internet

A Dissertation Submitted in Partial Satisfaction
of the Requirements for the Degree of

Doctor of Philosophy
in
Electrical and Computer Engineering
by

Karlo Berket

December 2000

The Dissertation of Karlo Berket
is approved:

Committee Chairman

December 2000

30 September, 2000

ACKNOWLEDGMENTS

The route to the completion of my dissertation has been long and winding. There are many people to be thanked for helping me along the way.

Firstly, I would like to thank my advisors Professor Michael Melliar-Smith and Professor Louise Moser for their guidance and support. I would also like to thank my committee members, Deb Agarwal, Professor Steven Butner and Professor Klaus Schauser, for their advice and time.

The stay in the lab was enjoyable and memorable due to the companionship of Elizabeth, Kim, Lauren, Mike, Nitya, Priya, Raj, Ravi, Ruppert, Stratis, Vana, and Wenbing. Thanks to one and all.

I would like to thank all my friends, you know who you are, for all their help and support throughout.

Finally, I would like to thank my family and my parents, for being with me at each step of this arduous, though memorable and enjoyable, journey.

This work was supported by the Collaboratory Interoperability Framework project, a part of the DOE2000 program.

VITA

1995	B.E. Department of Electrical Engineering The Cooper Union for the Advancement of Science and Art
1996	M.S. Department of Electrical and Computer Engineering University of California, Santa Barbara
1996-1997	Researcher Department of Electrical and Computer Engineering University of California, Santa Barbara
1997-2000	Researcher Distributed Systems Department Ernest Orlando Lawrence Berkeley National Laboratory

PUBLICATIONS

- "A group communication protocol for CORBA," with L. E. Moser, P. M. Melliar-Smith, and R. Koch, *Proceedings of the International Workshop on Group Communication*, Aizu-Wakamatsu, Japan, September 1999, pp. 30–36.

- "Multicast group communication for CORBA," with P. M. Melliar-Smith, L. E. Moser, P. Narasimhan and R. Koch, *Proceedings of the International Symposium on Distributed Objects and Applications*, Edinburgh, Scotland, September 1999, pp. 98–107.

- "Timestamp acknowledgments for determining message stability," with R. Koch, L. E. Moser and P. M. Melliar-Smith, *Proceedings of the 2nd International Conference on Parallel and Distributed Computing and Networking*, Brisbane, Australia, December 1998, pp. 1–8.

- "The InterGroup Protocols: Scalable Group Communication for the Internet" with L. E. Moser and P. M. Melliar-Smith, *Proceedings of the Third Global Internet Mini-Conference* (in conjunction with Globecom '98), Sydney, Australia, November 1998.

- "Self-stabilizing multiple-sender/single-receiver protocol," with R. Koch, *Proceedings of the 3rd Workshop of Self-Stabilizing Systems*, Santa Barbara, CA, August 1997, pp. 170–184.
- "On technologies in computer networks and distributed systems," with R. K. Budhia, K. P. Kihlstrom, R. Koch, N. Narasimhan, P. Narasimhan, E. M. Royer, M. D. Santos, A. Shum, E. Thomopoulos, P. M. Melliar-Smith and L. E. Moser, *looking.forward, The IEEE Computer Society's Student Newsletter*, vol. 5, no. 3, Fall 1997, pp. 2–6.

FIELDS OF STUDY

Major Field: Computer Engineering

Studies in Group Communication Systems
Professors P. M. Melliar-Smith and L. E. Moser

ABSTRACT

The InterGroup Protocols:
Scalable Group Communication
for the Internet

by
Karlo Berket

Reliable group ordered delivery of multicast messages in a distributed system is a useful service that simplifies the programming of distributed applications. Such a service helps to maintain the consistency of replicated information and to coordinate the activities of the various processes. With the increasing popularity of the Internet, there is an increasing interest in scaling the protocols that provide this service to the environment of the Internet. The InterGroup protocol suite, described in this dissertation, provides such a service, and is intended for the environment of the Internet with scalability to large numbers of nodes and high latency links.

The InterGroup protocols approach the scalability problem from various directions. They redefine the meaning of group membership, allow voluntary membership changes, add a receiver-oriented selection of delivery guarantees that permits heterogeneity of the receiver set, and provide a scalable reliability service.

The InterGroup system comprises several components, executing at various sites within the system. Each component provides part of the services necessary to implement a group communication system for the wide-area. The components can be categorized as: (1) control hierarchy, (2) reliable multicast, (3) message distribution and delivery, and (4) process group membership.

We have implemented a prototype of the InterGroup protocols in Java, and have tested the system performance in both local-area and wide-area networks.

Contents

1 Introduction **1**

1.1 Related Work 3

1.2 The InterGroup System 5

1.3 Delivery Services 8

1.3.1 Unreliable Unordered Message Delivery 8

1.3.2 Reliable Source Ordered Delivery 8

1.3.3 Reliable Group Timestamp Ordered Delivery 9

2 Control Hierarchy **11**

2.1 Model 11

2.2 Data Structures 12

2.2.1 Objects 13

2.2.2 Control Messages 14

2.2.3 Internal Variables 17

2.3 Distribution Protocols 19

2.4 Distance Calculation 20

2.5 Self-Determination Protocol 23

2.6 Hierarchy Maintenance 25

2.6.1 Initialization of a Control Process 25

2.6.2 State Transitions 26

2.6.3 Fault Detection and Handling 27

3 Reliable Multicast **29**

3.1 Model 29

3.2 Data Structures 30

3.2.1 Messages 30
3.2.2 Internal Variables 31
3.3 Retransmission Requests 33
3.4 Retransmission of Data 34
3.5 Buffer Management 35

4 Data Transmission and Delivery 37
4.1 Model 38
4.2 Data Structures 39
4.2.1 Data Message 39
4.2.2 Keep Alive Message 40
4.2.3 Application Data Message 40
4.3 A Variation of Lamport Time 41
4.4 Determining Message Stability 41
4.5 Data Transmission 41
4.5.1 Internal Variables 41
4.5.2 Data Transmission Algorithms 42
4.5.3 Flow Control and Congestion Avoidance 42
4.6 Unreliable Unordered Delivery 43
4.7 Reliable Source Ordered Delivery 44
4.7.1 Setup 44
4.7.2 Ordering and Delivery 44
4.8 Reliable Group Timestamp Ordered Delivery 47
4.9 Future Work 50

5 Process Group Membership 51
5.1 Model 51
5.2 The Data Structures 52
5.2.1 Internal Messages (Events) 52
5.2.2 Network Messages 54
5.2.3 Internal Variables 57
5.3 The Process Group Membership Protocol 57
5.3.1 Bootstrap 58
5.3.2 Leaving the Receiver State 61
5.3.3 Leaving the Sender State 62

5.4 Membership Repair Algorithm 64
5.4.1 Data Structures 64
5.4.2 The Protocol 65
5.5 Receiver Membership Repair Algorithm 70
5.5.1 Data Structures 71
5.5.2 The Protocol 71
5.6 Process and Network Fault Detector 73

6 Implementation 75
6.1 The InterGroupThreadedModule 75
6.2 The Statistics Gathering Infrastructure 76
6.2.1 Client-side Implementation 77
6.2.2 Server-side Implementation 77
6.2.3 Statistics Messages 77
6.2.4 The WAN Testbed 78

7 Performance 79
7.1 Local-area Environment 80
7.1.1 Throughput Measurements 80
7.1.2 Latency Measurements 81
7.1.3 Protocol Overhead 85
7.2 Wide-area Environment 86

8 Conclusion and Future Work 89

List of Figures

2.1 The control hierarchy. 12
2.2 Determining the inter-message interval. 21
2.3 Pseudocode for the distribution of control messages. 22
2.4 Pseudocode for TO_CHILD state. 27

3.1 Dynamic adjustment algorithm for the request timer parameters. 34
3.2 Dynamic adjustment algorithm for the repair timer parameters. 35

4.1 Algorithm for detecting missing messages. 45
4.2 Algorithm for ordering and delivering messages for the reliable source ordered message delivery service. 46
4.3 Algorithm for source ordering messages for the reliable timestamp group ordered message delivery service. 48
4.4 Algorithm for ordering and delivering messages for the reliable group timestamp ordered message delivery service. 49

5.1 State machine for the initialization of the membership protocols. 59
5.2 How the *Receiver* state is left in the membership protocols. . . 61
5.3 How the *Sender* state is left in the membership protocols. . . . 63
5.4 The Membership Repair Algorithm 66
5.5 Pseudocode for consensus. 67
5.6 Reasons for restarting consensus. 68
5.7 The Receiver Membership Repair Algorithm. 72

7.1 The delivery latency distribution at full throughput for four nodes. 83
7.2 The application delivery latency distribution for four nodes. . 84
7.3 The application delivery latency distribution for sixteen nodes. 85

7.4 The application delivery latency between ANL and LBNL. . . 87

List of Tables

2.1 Distance calculation 22
2.2 Evaluating Appropriateness. 24
2.3 Conditions for determining whether the state change should occur. The change continues if these conditions are satisfied. . . 24
2.4 Timers based on appropriateness values. 25

7.1 Data throughput on the CS cluster. 80
7.2 Data Throughput on the Laboratory LAN. 82
7.3 Delivery latency at full throughput. 82
7.4 Delivery latency at 10 msgs/sec send rate at each sender. . . . 84
7.5 Protocol overhead. 86

Chapter 1

Introduction

Reliable group ordered delivery of multicast messages in a distributed system is a useful service that simplifies the programming of distributed applications. With such a delivery service, provided by multicast group communication protocols, all processes in each group of the application receive the same messages in the same order. Such a service helps to maintain the consistency of replicated information and to coordinate the activities of the various processes.

In addition, multicast group communication protocols provide a membership service that allows the system to make progress in the presence of process faults. Such protocols require the group membership and delivery order to obey a form of virtual synchrony [12, 42, 8, 21].[1] Virtual synchrony and its variations define consistency constraints on processes transitioning between memberships.

With the increasing popularity of the Internet, there is an increasing interest in multicast group communication protocols that are scalable to the environment of the Internet. IP multicast [19] provides a scalable best-effort multicast service for the Internet, and is a valuable building block for group communication systems. It might appear that the same group communication protocols that work well over LANs could be run over IP multicast. Unfortunately, these protocols do not, in general, scale well to large numbers of nodes

[1]Some protocols require that, for each message delivered, a membership regarding that message be delivered. This results in running a consensus algorithm for every message, resulting in severe scalability problems, so these protocols will not be discussed.

and wide-area networks, such as the Internet. The ordering, reliability and membership algorithms of existing LAN-based protocols are not scalable.

The main hindrances to scalability of existing group communication systems are the message delivery guarantees provided to the application and their effect on the way in which membership is maintained. Group communication protocols require the group membership and the delivery order to obey a form of virtual synchrony.

Such requirements result in expensive (in time and messages required) membership repair algorithms. There appears to be no way around those membership repair algorithms, which require a consensus decision to be made. The message cost of those algorithms is $O(n^2)$, where n is the number of processes in the group. Furthermore, the interval between membership changes is inversely proportional to n and, thus, if the value of n is large, too much time can be spent in the membership protocol itself.

To ensure virtual synchrony, many group communication protocols stop delivering messages while membership changes are taking place and, thus, for large values of n, they might deliver no messages at all. Nevertheless, to obtain a consistent view of the membership and to ensure the message delivery guarantees, membership repair algorithms must be run. In traditional group communication systems, every process in a process group is treated as an equal. When the system requires a consensus decision, such as that used in a membership repair algorithm, every process participates.

Another problem that arises with the increase in size of these systems is providing a scalable reliability service. The biggest concern in building such a service is in consuming excessive bandwidth to provide the service. There has been a large amount of research in this area recently, mostly in the single sender case. However, most of the solutions are geared towards providing a scalable solution for error recovery and correction, and do not provide ordering and membership services.

Also, in traditional group communication systems, the delivery of messages from a process group to the application, is determined, either system-wide or by the individual senders. Thus, every process must provide the same delivery services.

The InterGroup protocol suite, described in this dissertation, is a group communication system that is intended for the environment of the Internet (long latencies, high message loss rates, relatively frequent partitioning) with scalability to large numbers of sites and high latency links.

1.1 Related Work

To provide video conferencing and other multicast services, the MBone (Multicast Backbone) was created as an experimental overlay network within the Internet. Through the years, multicast capability has been added to routing equipment by equipment vendors. The IP Multicast capability on the Internet today is provided by native multicast capable routers and the remnants of the original overlay network.

IP Multicast-based routing was established to allow distributed applications to achieve "real-time" communication over wide-area IP networks through a lightweight model of communication. The IP multicast routing and delivery models comprise multiple protocols that provide a UDP-based best-effort delivery service to the applications.This means that, when a message is sent, it has some probability of getting to the group members. There is no service provided for determining who received the message or even who the group members are.

Reliable message delivery is an important part of network communication. TCP/IP [15] provides this service for unicast communication. Recently, it has been recognized that group communication applications have a wider range of requirements than unicast applications. This has made the idea of a single, generic reliable multicast protocol for all group communication applications infeasible. Thus, a lot of research has focused on the mechanisms for error recovery and correction.

The current work on error recovery and correction has identified three types of reliability mechanisms: (1) acknowledgment-based (ACK-based), (2) negative acknowledgment-based (NACK-based), and (3) forward error correction (FEC) based.[2] In ACK-based protocols[57, 56, 28, 48, 63], the receivers send messages acknowledging the reception of messages. In NACK-based protocols[23, 14, 18, 29, 55], the receivers do not acknowledge the reception

[2]Some of these protocols are actually hybrids that combine more than one mechanism; we will focus on the primary mechanism when discussing them.

of messages; instead, they send messages indicating that a message has not been received. FEC-based protocols [60, 47, 51] place redundant information in every message that they send, allowing the receivers to recover a subset of messages that they do not receive. Each mechanism has a number of properties associated with it, and the protocols are differentiated by their use of these properties.

The Reliable Multicast Transport (RMT) working group of the IETF aims to standardize protocols that provide mechanisms for error recovery and correction, and categorizes these protocols as reliable multicast transport protocols. Currently, the efforts of this working group are focused primarily on the standardization of the one-to-many transport of large amounts of data[30].

Group communication systems provide membership and ordering services in addition to error recovery and correction. Typical applications that might use a group communication system include state-machine replication ([20, 45, 25, 32, 22]), distributed transactions and database replication ([52, 33]), load balancing ([34]), system management ([4]), system monitoring ([3]), highly available servers ([20, 45, 41]), and collaborative computing ([50, 11, 1, 16, 36, 26]). All of these applications require similar delivery guarantees.

Group communication systems typically offer one or more of the following ordering properties: source ordered, causal ordered, group ordered and total ordered. With a group communication system, messages are delivered within the context of views, *i.e.,* every message delivered to the application has a view associated with it. A view has an associated group membership, and the membership mechanisms are in charge of installing view changes.

Early group communication systems were designed with a focus on providing a limited number of services, and good performance in an asynchronous setting. These systems include: Isis [13], Trans/Total[39, 43], Transis[7, 6], Newtop[21], Phoenix[38], RMP[62], and Totem[2, 44], In general, these protocols do not scale well to large numbers of nodes and wide-area networks. Additionally, the monolithic nature of these systems make them hard to retool.

Later systems started a move towards a "building block" approach to designing the protocols. These systems focused on functional decomposition of mechanisms to build additional services. These systems include Horus[59] and Ensemble[58].

Existing work on WAN-enabled group communication systems has focused on retooling an existing system or building hybrid protocols (Spread[5], Atomic Group[35]).

Spread combines the single-ring approach of Totem at the site level. Between sites the Spread protocols create their own topology for distributing data. Messages are ordered across the rings to provide a system-wide ordering.

Atomic Group is a group communication system designed for ATM networks. The Atomic Group protocols create their own topology for distributing data. Processes are organized into local groups where the initial group ordering is done. Ordering across groups occurs when processes join more than one group.

Group communication systems vary in their definitions of services offered and in the terminology used. This has led to attempts to define a comprehensive set of properties of group communication systems that reflect numerous existing group communication system implementations[42, 24, 9, 22, 27, 61]. Such an exercise is useful for defining the delivery guarantees of a group communication system and relating them to other systems.

1.2 The InterGroup System

The InterGroup protocols approach the scalable group communication problem from various directions. Our solution includes redefining the meaning of group membership, allowing voluntary membership changes, adding a receiver-oriented selection of delivery guarantees that permits heterogeneity of the receiver set, and providing a scalable reliability service.

We attempt to cut down on the cost of the membership repair algorithms using the following strategies. In the InterGroup system, not all processes are equal. In each process group, a process is classified by its recent activity. If the process has been sending data to the group recently, it is classified as an *active sender*. Active senders are members of the *sender group* of the process group. Only the senders in the process group need to participate in the majority of consensus decisions. We also use voluntary mechanisms for entering and leaving the group. Voluntary mechanisms can take advantage of the properties of the system to avoid executing the membership repair algorithms.

We step away from the traditional approach of choosing a delivery service to provide more flexibility to the application and to reduce the costs of the membership repair algorithms. In the InterGroup system, the delivery of messages from a process group is determined by the application, at the receiving end. Each process may choose a different delivery service. The InterGroup system provides the following delivery services within a process group:

- **Unreliable unordered**. Messages received from the process group are delivered directly to the application. Some messages might never be received, and multiple copies of the same message might be received. Moreover, there is no guarantee on the order in which messages are received. IP Multicast provides this functionality.

- **Reliable source ordered**. All messages from a particular source will be received by the application (unless a process failure occurs), and they will be delivered in sequence number order. This service is well suited to applications like multicast file transfers and other applications currently using TCP/IP [15]. The services provided are similar to TCP/IP.

- **Reliable group timestamp ordered**. Messages are received by the application in timestamp order over the entire process group. The membership service ensures that the receipt of messages by the application obeys virtual synchrony. This service is closest to the idea of agreed messages in group communication systems[2].

Although, we previously stated that the processes in the sender group can usually make consensus decisions by themselves there are some decisions that involve processes not in the sender group. The categorization of processes by their delivery service allows us to determine which processes must participate in those decisions. In the InterGroup system, only the processes that are in the sender group and/or have specified a delivery service of reliable group timestamp ordered need to participate in the consensus decisions required by the membership repair algorithms. This reduces the number of processes that need to run the consensus protocol and, thus, improves the scalability of the system.

The control information in the InterGroup system is communicated through a *control group*. The control group consists of exactly one *control process* from every site in the system. A control process runs independently of the other processes at a site. Every process at a site sends and receives control information via the control process of its site. A control process has no direct interaction with the applications. The control processes are arranged in a self-organizing hierarchical structure. The hierarchy organization mechanisms are based on those used with SRM[23, 54].

SRM provides a flexible mechanism for the retransmission of missing messages. The SRM retransmission algorithms require information about the latency between processes to provide their services. This information is gathered through the control hierarchy to provide additional scalability. The InterGroup system requires buffer management as part of the reliability services. The control hierarchy also provides mechanisms to gather and distribute acknowledgment information (used for buffer management) and for gathering consensus information within the system in a scalable manner.

The InterGroup system comprises a number of components, executing at the sites within the system. Each component provides part of the services necessary to build a group communication system for the wide-area. The components can be categorized as: (1) control hierarchy, (2) reliable multicast, (3) message distribution and delivery, and (4) process group membership.

The control hierarchy is used for the exchange of control information between the sites in the system. Each site has a control process that is responsible for the control information for all of the processes at that site. The control hierarchy also provides a mechanism for determining message stability, and a mechanism for reaching a weak form of consensus for the group. The control hierarchy is explained in detail in Chapter 2.

Reliable multicast provides a mechanism to request the retransmission of messages, a mechanism to retransmit messages, and a mechanism to detect whether a message can be recovered. The reliable multicast protocol is explained in Chapter 3.

Message distribution and delivery entails the following tasks: perform flow control on the messages sent to a process group, order and deliver messages to the application based on the delivery service chosen by the application for a process group, and detect missing messages (depending on the delivery service). The message distribution and ordering are explained in detail in Chapter 4.

The process group membership provides a view of the sender group membership. This view depends on the delivery service chosen by the application and on whether the process is a member of the sender group. The process group membership is explained in Chapter 5.

The interactions between the components and the implementation of the system are described in Chapter 6.

1.3 Delivery Services

In this section we present the services that the InterGroup protocols provide to the application. Each service (other than unreliable unordered message delivery) is specified by a pair of properties.

1.3.1 Unreliable Unordered Message Delivery

This service provides the delivery guarantee of the underlying communication network *i.e.,* IP multicast. We guarantee that there is no degradation of that service, other than small delays. The IP multicast service can reorder messages, lose messages and duplicate messages.

1.3.2 Reliable Source Ordered Delivery

The first property (1.1) of the source ordered message delivery service states that messages from the same sender arrive in the order in which they were sent.

Property 1.1 InterGroup Source Ordered Delivery. *If a process p sends two messages, then these messages are received in the order in which they were sent at every process that receives both of them.*

The second property (1.2) of the source ordered message delivery service states that there are no gaps in the source order as long as the sending process is a member of all views of the receiving process during a specified interval.

Property 1.2 InterGroup Reliable Source Ordering. *If a process p sends message m before message m', then any process q that receives m' receives m as well, provided that p has been a member of the membership set of all views between (and including) the sending of message m and the reception of message m' at process q.*

The notion of a view at a process that has subscribed to the reliable source-ordered message delivery service differs slightly from the usual notions of a view in group communication systems (GCSs). For this service, we do not require that the view installed at a process be consistent with the views of other processes, or that it be precise with respect to the process group. We are only concerned with receiving messages from one process in the process group and, thus, only require precision with respect to that process. Therefore, for a process p subscribing to this service from process q (*i.e.,* q is the source), we are only concerned with whether or not q appears in the view at process p.

1.3.3 Reliable Group Timestamp Ordered Delivery

The first property (1.3) of the reliable group timestamp ordered message delivery service states that messages are delivered in the same order at all of the processes that deliver them, while preserving the causal relationships between messages.

Property 1.3 InterGroup Timestamp Ordered Delivery. *Any two messages sent in the system are received in the same order at any process that receives both of them. Furthermore, causality is preserved.*

A causal relationship between messages is defined by a precedence relationship among the messages (first defined in [37]). The causal order extends the source order by requiring that a message m is delivered after all of the messages that were received at the sending process before it sent m.

The second property (1.4) of the reliable group timestamp ordered message delivery service states that there are no “causal holes” in the group order as long as the sending processes that produce the causal relationship are members of all views of the receiving process during a specified interval.

Property 1.4 InterGroup Reliable Timestamp Order. *If message m sent by process p causally precedes message m' sent by process p', and process q receives m', and p and p' have been members of all views at process q between (and including) the sending of m and the reception of m' by q, then q receives m.*

Chapter 2

Control Hierarchy

This chapter describes the design of the control hierarchy used to distribute control information in the InterGroup system. The participants are organized in a hierarchical structure. Because the environment in which the control protocols run has a dynamic membership and changing topology, this hierarchy is self-organizing. The self-organizing algorithms are based on those presented in [54], which are intended for use with SRM [23]. However, the requirements of the InterGroup system are different from those of SRM, so the self-organizing algorithms are different.

First, we present the model used in the design of the hierarchy. Next, we present the data structures used in the control protocols. Finally, we present the various protocols used within the control hierarchy.

2.1 Model

We consider an asynchronous distributed system consisting of n control processes $p_1, p_2, \ldots, p_n$ that communicate via messages over a network. The system is asynchronous in that no bound can be placed on the time required for a computation or for communication of a message. Control processes have access to local clocks. These clocks are not synchronized.

The control processes are organized as a bush, with the root control processes called the *coordinators*, and the leaf control processes called the *children*, as shown in Figure 2.1. Each control process may be a coordinator, a child, or

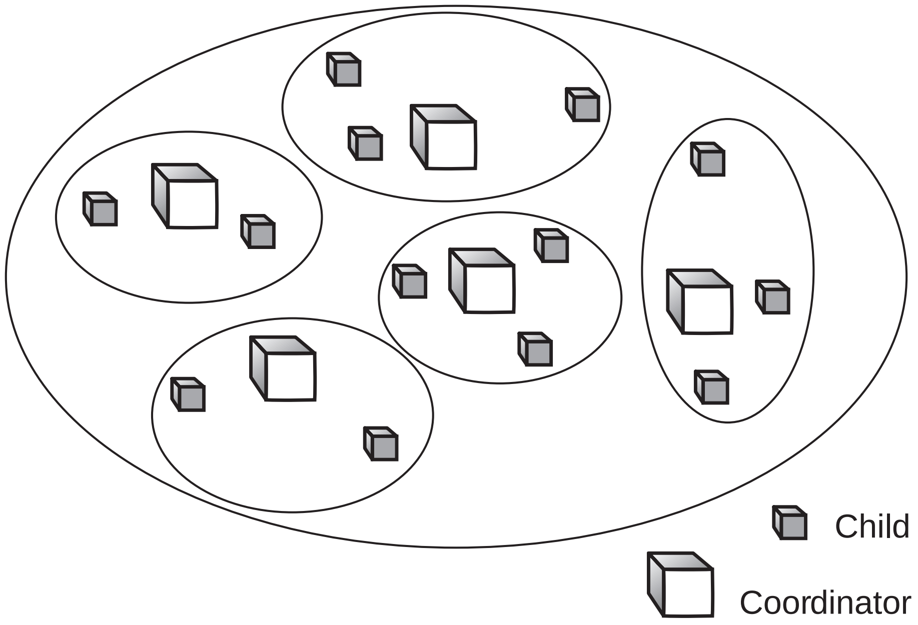

Figure 2.1: The control hierarchy.

both at any given time. Each child is associated with at least one coordinator at all times. The *local group* of a coordinator is composed of the children associated with that coordinator (including the coordinator itself). The *coordinator group* of the hierarchy consists of all coordinators in the hierarchy.

A child multicasts messages to the other control processes in its local group. A coordinator multicasts messages to the control processes in its local group and to the control processes in the coordinator group. A control process receives all of its own multicast messages.

Communication between control processes is unreliable and, thus, messages may be lost or arbitrarily delayed. Communication channels are not assumed to be FIFO. The network is allowed to partition and remerge.

2.2 Data Structures

The control hierarchy employs a number of message types to distribute the control information to the control processes. Each control process in the hierarchy maintains internal data constants and variables used in the control protocols. The messages and internal variables use several common data objects, which we present first. Next, we explain the control messages. Finally, we present the internal data constants and variables.

2.2.1 Objects

There are several data objects that represent the control information. These data objects are used in control messages and/or internal variables.

Timestamp Pair Object

A timestamp pair (TP) object contains the information about the sending and reception times of the last control message sent by a control process A and received by a control process B. These entries are used in calculating the distance (latency) between control processes in the control hierarchy. A TP object contains the following fields:

- *t1*: The time at which control process A sent the last control message that control process B received.

- *t2*: The time at which control process B received the last control message from control process A.

Control Process Information Object

A control process information (CPI) object for control process B, contains control information about a particular control process A. A CPI object contains the following fields:

- *processID*: The unique identifier of control process A.

- *timestampPair*: A TP object regarding control processes A and B.

- *ttl*: The distance (in TTL) between control process A and control process B.

- *distance*: The distance (in milliseconds) between control process A and control process B.
- *lastUpdated*: The timestamp at which this information was last updated.

2.2.2 Control Messages

Three message types are used to communicate between control processes. Each of the messages contains, in its header, the information necessary to process the messages and calculate the distance information. The reliable multicast protocol (Chapter 3) and process group membership protocol (Chapter 5) use the *data* fields contained in these control messages to calculate message stability and the time at which a membership change occurs.

Local Control Message

A local control (LCon) message is sent by a member of a local group to that local group. It is used for distributing control information between the members of the group. A LCon message contains the following fields:

- *type*: LCon
- *senderID*: The unique control process identifier of the sender.
- *ttl*: The TTL with which the message was sent.
- *timestamp*: The local time at which the message was sent.
- *coordinator*: The coordinator associated with the sender of this message. This field uniquely identifies the local group to which the sender belongs and, thus, the local group to which the message is sent.
- *numLocal*: The number of control processes in the local group with which the sender is currently exchanging information (this may be a subset of the local group).
- *timestampPairEntries*: An array, where the indexes are the identifiers of the control processes in the local group of which the sender is aware, and

the values are TP objects where B is the sender and A be is the control process identified by the index.

- *data*: The application-specific data.

Coordinator Control Message

A coordinator control (CCon) message is sent by a coordinator to its local group. It is used to distribute information from the coordinator to the members of its local group. A CCon message, is used by the coordinator because the coordinator needs to send extra information to its local members for two reasons: (1) the coordinator is the only control process in the local group that communicates with the other coordinators, and (2) the coordinator is the only control process in the local group that is able to communicate with every member of the local group. A CCon message contains the following fields:

- *type*: CCon
- *senderID*: The unique identifier of the sender. This field also uniquely specifies the local group to which the message is sent.
- *ttl*: The TTL with which the message was sent. This field is also, the TTL to the control process in this local group that is the farthest from this coordinator.
- *timestamp*: The local time at which the message was sent.
- *farDistance*: The distance (in milliseconds) to the control process, in this local group, that is the farthest from this coordinator.
- *numLocal*: The number of control processes in this coordinator's local group.
- *numCoords*: The number of coordinators in the coordinator group.
- *timestampPairEntries*: An array, where the indexes are the identifiers of the control processes in this local group, and the values are TP objects where A is the sender and B is the control process identified by the index.

- *localDistances*: An array, where the indexes are the identifiers of the control processes in this local group, and the values are the calculated distances (in milliseconds) from the sender to the control process identified by the index.

- *coordDistances*: An array, where the indexes are the identifiers of the control processes in the coordinator group, and the values are the calculated distances (in milliseconds) from the sender to the control process identified by the index.

- *data*: The application-specific data.

Global Control Message

A global control (GCon) message is sent by a coordinator to the coordinator group. It is used for exchanging control information between the coordinators. A GCon message contains the following fields:

- *type*: GCon

- *senderID*: The unique identifier of the sender.

- *timestamp*: The local time at which the message was sent.

- *numLocal*: The number of control processes in the sender's local group.

- *numCoords*: The number of coordinators in this coordinator group.

- *timestampPairs*: An array, where the indexes are the identifiers of the control processes in the coordinator group, and the values are TP objects where A is the sender and B is the control process identified by the index.

- *data*: The application-specific data.

LCon and CCon messages are sent only to the local group to which the sender belongs. Thus, these messages are called *local control messages*. GCon messages are sent to the coordinator group, but can reach control processes in the entire system. Thus, these messages are called *global control messages*.

2.2.3 Internal Variables

Each control process maintains several parameters that are constant:

- *LOW_STATE_CHNG_PARAM*: The low-end hierarchy parameter. It is used to determine the bounds on the scheduling times for a change of state between coordinator and child.

- *HIGH_STATE_CHNG_PARAM*: The high-end hierarchy parameter. It is used to determine the bounds on the scheduling times for a change of state between coordinator and child.

- *LOW_THRESH_COORD_SIZE*: Low threshold of coordinator group size. The number of coordinators in the hierarchy should be greater than this number (if there are enough control processes in the hierarchy).

- *HIGH_THRESH_COORD_SIZE*: High threshold of coordinator group size. The target number of coordinators in the hierarchy is less than this number.

- *LOCAL_TARGET_SIZE*: The target group size of the local groups.

- *TARGET_BANDWIDTH*: The target control bandwidth. This field is used for flow control.

- *BOUNDARY_TTL*: The value that determines whether this control process's coordinator is close to this control process. This value determines the maximum allowable radius of a local group, *i.e.,* the protocol attempts to keep the radius of local groups below this value. In the initial implementation, this value is set to 16.

- *COORD_CLOSENESS*: The value that determines whether another coordinator is close to this control process. This value specifies the target distance (in milliseconds) between any two coordinators in the group. In the initial implementation this value is set to 30ms.

Each control process maintains the following local variables for use in the control hierarchy protocols:

- *myID*: The unique identifier of this control process.[1]
- *state*: The state of this control process in the hierarchy: COORDINATOR, TO_COORDINATOR, CHILD, TO_CHILD.
- *amChild*: True if this control process is acting as a child; false otherwise.
- *amCoord*: True if this control process is a coordinator; false otherwise.
- *myCoord*: The coordinator associated with this control process. The local group of this control process is uniquely identified by the coordinator associated with this control process. If the *amCoord* variable is *true* and the *amChild* variable is *false*, the value of *myCoord* is equal to *myID*.
- *coordTTL*: The distance (in TTL) to the coordinator associated with this control process. If the *amChild* variable is *false*, the value of *coordTTL* is meaningless.
- *localGroupFarTTL*: The distance (in TTL) from *myCoord* to the local group control process that is the farthest (in terms of TTL) from it.
- *localGroupFarDistance*: The distance (in milliseconds) from *myCoord* to the local group control process that is the farthest (in terms of latency) from it.
- *closeCoordDistance*: The distance (in milliseconds) to the closest (smallest latency) coordinator outside my local group.
- *numCoords*: The number of coordinators in this control hierarchy.
- *numLocals*: The number of control processes in the local group to which this control process belongs (excluding the coordinator).
- *coords*: The control information about the coordinators in this control hierarchy. This is an array referenced by the unique identifier of a coordinator. The values in the array are the Control Process Info objects representing these control processes.

[1]This is currently the IP address of the machine the control process is running on. No two control processes may have the same identifier.

- *locals*: The control information about the members of this control process's local group. This is an array referenced by the unique identifier of a child. The values in the array are the Control Process Info objects representing these control processes.

- *appropriateness*: The likelihood of this control process to change from being a child, to being a coordinator, or vice-versa. This value is one of the following: APPROPRIATE, NEUTRAL, INAPPROPRIATE, or NO_CHANGE.

- *appropriatenessTimers*: The bounds of the timer values for different *appropriateness* states. These are determined from *LOW_STATE_CHNG_PARAM* and *HIGH_STATE_CHNG_PARAM* using Table 2.4.

- *averageGlobalPacketSize*: The weighted average size (in bytes) of the global control packets received by this control process. This information is used for flow control of GCon messages.

- *averageLocalPacketSize*: The weighted average size (in bytes) of the local control messages received by this control process. This information is used for flow control of LCon and CCon messages.

2.3 Distribution Protocols

LCon messages are multicast with a TTL equal to *coordTTL*. This is the smallest TTL with which a child may multicast messages without disrupting the flow of control information in the hierarchy (as long as CCon messages provide enough data). Higher TTLs are allowed, but are not recommended because this results in unnecessary use of bandwidth. CCon messages are multicast with a TTL equal to *farTTL*, this being the smallest TTL with which all control processes in the local group are reachable from the coordinator. GCon messages are multicast with a TTL equal to the system TTL, which is defined by the user.

The flow control for the distribution of control information is a simplified version of that used in RTCP (see [53]). The time to wait before sending the next message, called the *intermission interval*, is independently calculated at each control process after every control message sent by that control process. The intermission interval calculation is shown in Figure 2.2. The algorithm for the calculation of the intermission interval is the same for local and global control messages, but they are separate instances, using different information for determining their respective intermission intervals.

Control information is distributed through control messages. The pseudo-code for the distribution of control messages is shown in Figure 2.3. Every time the protocol receives a local control message from a control process represented in *locals* or from *myCoordinator*, it updates the *averageLocalPacketSize*, as shown in Figure 2.3. Every time the protocol receives a global control message from a control process represented in *coords*, it updates the *averageGlobalPacketSize*, as shown in Figure 2.3.

2.4 Distance Calculation

One of the objectives of exchanging control information is to calculate the distance (latency) between control processes. This information is used for the self-determination algorithms (Section 2.5), and in the reliable multicast protocol (Chapter 3).

The distance data are distributed and that information is gathered to calculate the distances to a subset of control processes in the system. This subset is sufficient to approximate the distance to any control process in the system, while using minimal resources to store this information. Each control process maintains the distances to all of the other control processes in its local group (in the *locals* variable) and to all of the control processes in the coordinator group (in the *coords* variable). The formulas shown in Table 2.1 are used to determine the approximate distance to any other control process.

The distances are calculated using a simplified version of the NTP[40] algorithm. Each control message includes a local timestamp, that records when the message was sent. Let the control process sending a control message with a local timestamp T3, be control process A, and let the control message, be msg. When another control process receives a control message, it records its local timestamp. Another control process B, receives the control message msg at local time T4. In the message there is a Timestamp Pair representing control

Computes the time until the next control packet should be sent.
Algorithm based on Appendix A.7 Computing the RTCP Transmission
Interval of RFC 1889.

Constants:
- RTCP_MIN_TIME = 1
  - Minimum time between control packets from this site (in seconds).
- RTCP_SIZE_GAIN = 1/16
  - Gain (smoothing constant) for the low-pass filter that estimates the average control packet size.

Parameters:
- **members** the estimated number of group members; on the first call, this parameter should have the value 1
- **controlBw** the target control bandwidth, in bytes per second; we use a parameter supplied to the hierarchy at startup
- **packetSize** the size of the control packet just sent, in bytes
- **avgPacketSize** estimator for control packet size; updated for the packet just sent, and also updates for every control packet received
- **initial** flag that is true for the first call upon startup

```
controlPacketInterval(int members, double controlBw, int packetSize,
                      int avgPacketSize, boolean initial)

    The first call at application start-up uses half the min delay for quicker notification. The
    average control packet size is initialized to 128 bytes which is a conservative estimate.

    if (initial)
      rtcpMinTime = RTCP_MIN_TIME/2
      avgPacketSize = 128
    else
      rtcpMinTime = RTCP_MIN_TIME

    The effective number of sites times the average packet size is the total number of bytes
    sent when each site sends a report. Dividing this by the effective bandwidth gives the time
    interval over which those packets must be sent to meet the bandwidth target, with a
    minimum enforced. In that time interval the protocol sends one report so this time is also
    the average time between reports.

    t = (avgPacketSize) * members / controlBw
    if (t < rtcpMinTime)
      t = rtcpMinTime

    To avoid traffic bursts from unintended synchronization with other sites, the protocol then
    picks the actual next report interval as a random number uniformly distributed
    between 0.5*t and 1.5*t.

    return t * (Math.random() + 0.5)
```

Figure 2.2: Determining the inter-message interval.

```
Local send module:
  $avgLocalTime$ = **controlPacketInterval**(1, $TARGET\_BANDWIDTH$, 0, 0, true)
  sleep($avgLocalTime$)

  loop
    lastLocalPacketSize = 0
    if ($amChild = true$)
      msg = createLocalMessage()
      send(msg)
      lastLocalPacketSize = msg.length
    if ($amCoord = true$)
      msg = createCoordMessage()
      send(msg)
      lastLocalPacketSize = lastLocalPacketSize + msg.length
    $avgLocalPacketSize$ = $avgLocalPacketSize$ +
                     (lastLocalPacketSize − $avgLocalPacketSize$) $*$ $RTCP\_SIZE\_GAIN$
    localSleepFor = **controlPacketInterval**($numLocal$, $TARGET\_BANDWIDTH$,
                              lastLocalPacketSize, $avgLocalPacketSize$, false)
    $avgLocalTime$ = $avgLocalTime$ + (localSleepFor − $avgLocalTime$) $*$ $RTCP\_SIZE\_GAIN$
    sleep(localSleepFor)

Global send module:
  $avgGlobalTime$ = **controlPacketInterval**(1, $TARGET\_BANDWIDTH$, 0, 0, true)
  sleep($avgGlobalTime$)

  loop
    msg = createGlobalMessage()
    send(msg)
    lastGlobalPacketSize = msg.length
    $avgGlobalPacketSize$ = $avgGlobalPacketSize$ +
                     (lastGlobalPacketSize − $avgGlobalPacketSize$) $*$ $RTCP\_SIZE\_GAIN$
    globalSleepFor = **controlPacketInterval**($numCoord$, $TARGET\_BANDWIDTH$,
                              lastGlobalPacketSize, $avgGlobalPacketSize$, false)
    $avgGlobalTime$ = $avgGlobalTime$ + (globalSleepFor − $avgGlobalTime$) $*$ $RTCP\_SIZE\_GAIN$
    sleep(globalSleepFor)
```

Figure 2.3: Pseudocode for the distribution of control messages.

	q is a child	q is a coordinator
p is a child	$dist(p, coord_p)+$ $dist(coord_p, coord_q)$	$dist(p, coord_p)+$ $dist(coord_p, q)$
p is a coordinator	$dist(p, coord_q)$	$dist(p, q)$

$coord_x$ is the coordinator associated with control process x
$dist(x, y)$ is the distance from control process x to control process y

Table 2.1: Distance calculation

process B (T1,T2). Thus, upon receiving msg, control process B can calculate the one-way distance to control process A as:

$$distance = (T4 - T3 + T2 - T1)/2$$

This calculation of the distance assumes symmetric paths (which is not always the case), but it does not assume synchronized clocks.

Children exchange distance information with control processes in their local group through LCon messages. Coordinators exchange distance information with control processes in their local group via CCon messages, and with the other coordinators through GCon messages. As mentioned earlier, CCon messages include more information than the Timestamp Pairs. They include (1) the distances from this coordinator to all other coordinators, which helps the children determine distances to the control processes outside their local group, and (2) the distances from this coordinator to all of the control processes in this local group, which allow a child to estimate the distance to control processes in its local group with which it cannot communicate directly (due to TTL restrictions).

2.5 Self-Determination Protocol

The self-determination protocol is used to determine whether a control process in the hierarchy should change states (either from being a child to being a coordinator or vice versa). The determination of a state change is a purely local decision based entirely on the control information gathered up to the time the protocol is executed, and on a predefined set of rules adapted from [54].

The protocol for self-determination follows a set of nested rules. First, an appropriateness value for changing states is determined from the rules presented in Table 2.2. This value is stored in the *appropriateness* variable at the control process.

Based on the *appropriateness* determined by the protocol, another set of rules (shown in Table 2.3) is consulted to determine whether to continue.

If the conditions are satisfied and the protocol decides to continue, a randomized timer is set. This timer is based on the *appropriateness* of the change and is chosen from an interval shown in Table 2.4. The constants

Appropriateness	Coordinator	Child
NO_CHANGE	$numCoords < THL$	
APPROPRIATE	$numLocals = 0$ and *close* to another coordinator	$numLocals > LTS$ and *far* from my coordinator
NEUTRAL	$numLocals < LTS$ and *close* to another coordinator	$numLocals \leq LTS$ and *far* from my coordinator
INAPPROPRIATE	All other cases	All other cases

$THL = LOW\_THRESH\_COORD\_SIZE$
$LTS = LOCAL\_TARGET\_SIZE$

Table 2.2: Evaluating Appropriateness.

Appropriateness	Coordinator	Child
APPROPRIATE	always	always
NEUTRAL	$numCoords > (THL + 3 * THH) / 4$	$numCoords < (3 * THL + THH) / 4$
INAPPROPRIATE	$numCoords > THH$	$numCoords < THL$

$THL = LOW\_THRESH\_COORD\_SIZE$
$THH = HIGH\_THRESH\_COORD\_SIZE$

Table 2.3: Conditions for determining whether the state change should occur. The change continues if these conditions are satisfied.

LOW_STATE_CHNG_PARAM and *HIGH_STATE_CHNG_PARAM* are set at the start of the protocol by the application.

Once the timer expires, a coordinator checks whether one of the following events has occurred during the waiting period: (1) a new child has contacted this control process asking this control process to be its coordinator, (2) this control process has sent a response to such a request, or (3) a child has been added to this control process's local group. If any of these events has occurred, the coordinator aborts the state change. Otherwise, it rechecks the rules in Table 2.3. If those conditions are still satisfied, it notifies the hierarchy maintenance protocol (Section 2.6) to start the state change to becoming a child.

A child, upon the expiration of the timer, rechecks the rules in Table 2.3. If those conditions are still satisfied, it notifies the hierarchy maintenance protocol (Section 2.6) to start the state change to become a coordinator.

This protocol is executed periodically. It is prevented from executing only when the hierarchy maintenance protocol is in the middle of a state change.

Appropriateness	Interval for Timer
APPROPRIATE	[$S1$, $S2$]
NEUTRAL	[$S1 + S2$, $S1 + 2 * S2$]
INAPPROPRIATE	[$S1 + 2 * S2$, $S1 + 3 * S2$]

$S1 = LOW\_STATE\_CHNG\_PARAM$
$S2 = HIGH\_STATE\_CHNG\_PARAM$

Table 2.4: Timers based on appropriateness values.

2.6 Hierarchy Maintenance

First, we describe how a control process joins the hierarchy, *i.e.,* the initialization of a control process. Next, we show the states a control process may be in, and the transitions between states. Finally, we present the fault detection and handling mechanisms of each control process in the hierarchy.

2.6.1 Initialization of a Control Process

A control process, upon startup, checks to see how many coordinators are present in the hierarchy, by checking the messages sent in the coordinator group. [2] If the number of coordinators is less than or equal to $(LOW\_THRESH\_COORD\_SIZE + HIGH\_THRESH\_COORD\_SIZE)/2$ or the control process does not receive a GCon message within a given time, the control process decides to start up as a coordinator. Otherwise, the control process decides to start up as a child. The average of the coordinator group size threshold parameters is used for the initial decision, because such a calculation can be made quickly (from only one GCon message), and gives room for error if many control processes join simultaneously.

A control process, starting up as a coordinator, starts sending GCon messages and accepting requests in order to be the coordinator for the control processes. Once these mechanisms are initialized, the control process is considered to be in the coordinator group, and enters the COORDINATOR state.

A control process starting up as a child, needs to find a coordinator that will accept it into its local group. This step is accomplished by using an

[2]Only one GCon message is necessary to make this determination.

expanding ring search. When it finds a coordinator, the control process starts sending LCon messages to the coordinator's local group. By sending LCon messages to the local group, the control process becomes a child in the group, and enters the CHILD state. If the expanding ring search doesn't provide a coordinator within a given time, the control process starts up as a coordinator, instead.

2.6.2 State Transitions

A control process may be in one of four states: COORDINATOR, TO_COORDINATOR, CHILD, or TO_CHILD. Each control process is normally in the COORDINATOR or CHILD state.

The transition from the COORDINATOR state to the TO_CHILD state is made when the self-determination protocol determines that this control process should leave the coordinator group. In this state, the control process (1) attempts to find a coordinator, whose local group it can join, and (2) waits for all the children in its local group to leave that local group. Once both of these tasks are accomplished, it transitions to the CHILD state. The pseudocode for this control process, while it is in the TO_CHILD state, is shown in Figure 2.4.

The transition from the CHILD to the TO_COORDINATOR state is started when the self-determination protocol decides that the control process should become a coordinator. On entering this state, the control process joins the coordinator group (by starting the sending of GCon messages) and leaves its local group (by stopping the sending of LCon messages). Once this occurs, it transitions to the COORDINATOR state.

During the two transitional states (TO_CHILD and TO_COORDINATOR) the control process may be both a coordinator and a child. This duality is permitted, so that the flow of control information may continue uninterrupted during these states. In the TO_CHILD state, the control process stops being a coordinator once all of the children in its local group leave that group. Thus, when it transitions to the CHILD state, it is only a child. However, when the control process transitions from the TO_COORDINATOR state to the COORDINATOR state, it is no longer a child locally, but other control processes in the system may think of it as still being both. The fault detection and handling protocol (Section 2.6.3) eventually removes the control process

```
TO_CHILD state processing:
    stop allowing new children
    if (any control processes in $coords$ are in $locals$ OR
      $myID$ is the only one in $coords$)
        allow new children
        return to COORDINATOR state
    send Ring Query messages using an expanding ring search
    start sending Coordinator Leave messages to my local group
    loop
      if (ring search has failed)
        allow new children
        stop sending Coordinator Leave messages to my local group
        return to COORDINATOR state
      if (received a Search Response message)
        join the senders local group
        iLeft = $true$
      if (all of the children in my local group have left)
        stop sending Coordinator Leave messages to my local group
        localLeft = $true$
      if (iLeft AND localLeft)
        go to CHILD state

When a child receives a Coordinator Leave message it attempts to find
another coordinator.
```

Figure 2.4: Pseudocode for TO_CHILD state.

from the local group in which it was a child.

2.6.3 Fault Detection and Handling

The detection of control process failures is accomplished via an algorithm that runs periodically. If the information from a control process has not been updated recently, the control process is removed from the data structures and thus removed from this control process's view of the membership of the hierarchy.

Any time a Control Process Info object is updated, the *lastUpdated* field is updated with the current time. A Control Process Info object representing control process *processID* is updated: (1) during the processing of a control message from the control process identified by *processID*, (2) if an entry for that control process is contained in the *localDistances* field of a CCon message, or (3) if an entry for that control process is contained in the *coordDistances* field of a CCon message.

The detection algorithm checks the *lastUpdated* field for every Control ProcessInfo object in the *locals* and *coords* variables. For each control process, if that field has not been updated within a given time interval, the Control Process Info object is removed from the appropriate variable.

Chapter 3

Reliable Multicast

Reliable multicast is an essential part of any group communication system. For the InterGroup systems we have designed a reliable multicast that is an adaptation of SRM[23]. The reliable multicast service provides a means to recover messages, without enforcing any particular delivery order. The recovery of messages is accomplished via retransmission requests (Section 3.3) and retransmitted data (Section 3.4) Thus, detection of losses and the ordering of messages is relegated to the process group delivery mechanisms explained in Chapter 4.

For the InterGroup system, we do not assume infinite buffers, thus the reliable multicast is also in charge of buffer management. This part of the reliable multicast is presented in Section 3.5.

3.1 Model

We consider an asynchronous distributed system consisting of n processes $p_1, p_2, \ldots, p_n$ that communicate via messages over a network. Each *process* within the system is represented by a unique identifier. The system is asynchronous in that no bound can be placed on the time required for a computation or for communication of a message. Processes have access to local clocks on their processor. These clocks are not synchronized.

Each process has access to a control process in the control hierarchy (Chapter 2). During the lifetime of a process, it may associate itself with one and only one control process in the control hierarchy. A process communicates with its associated node using a reliable FIFO communication channel.

Processes multicast messages to the other processes in the group. A process receives all of its own multicast messages. Communication between processes is unreliable and, thus, messages may be lost or arbitrarily delayed. Communication channels are not assumed to be FIFO. The network is allowed to partition and remerge.

3.2 Data Structures

3.2.1 Messages

The reliable multicast protocols use several different types of messages.

Retransmission Request Ticket

A retransmission request ticket (RTRTicket) contains the information that is necessary to identify a message uniquely and to recover a missing message. In the InterGroup system, a message is uniquely identified by the identifier of the process that sent the message[1] process and the sequence number that the sending process assigned to it. An RTRTicket contains the following fields:

- *sender*: The process identifier of the sender of the missing message.
- *coordinator*: The identifier of the control process that is the coordinator associated with the sender. This field is required for estimating the distance to the sender of the missing message (see Section 2.4).
- *seq*: The sequence number of the missing message.

Retransmission Request Message

A retransmission request (RTR) message is used to request the retransmission of a message from the process group. It contains the following fields:

[1]The identifier of a process consists of an the IP address of the machine that the process is running on and a number. The number is assigned to the process, such that no other process running on the same machine has the same number.

- *type*: RTR
- *senderID*: The process identifier of the sender.
- *coordinator*: The identifier of the control process that is the coordinator associated with the sender. This field is required for estimating the distance to the sender of the missing message (see Section 2.4).
- *ticket*: The RTRTicket describing the missing message.

Retransmitted Message

A retransmitted (RTX) message is used for retransmitting a message to the process group. It contains the following fields:

- *type*: RTX
- *senderID*: The process identifier of the sender.
- *origMsg*: The original message as it was first sent to the process group.

3.2.2 Internal Variables

The following constants are used in the reliable multicast protocols:

- *AVG_DUP_REQ*: Threshold for the number of average duplicate retransmission requests received. In the prototype implementation this value is set to 1.
- *AVG_REQ_DELAY*: Threshold for the average delay in sending a retransmission request. In the prototype implementation this value is set to 1.
- *AVG_RTX_DELAY*: Threshold for the average delay in sending a retransmission. In the prototype implementation this value is set to 1.
- α: Parameter for calculating an exponential-weighted moving average. [2] In the prototype implementation this value is set to 0.25.

[2] $avg = (1 - \alpha) * avg + \alpha * lastValue$

- ϵ: Parameter for modifying threshold constants by a small amount in some calculations. In the prototype implementation this value is set to 0.1.

The reliable multicast protocol uses the following variables:

- *lowReqParam*: The low-end request timer parameter. The request timer is used in determining how long the protocol should wait between receiving a notification of a missing message and sending an RTR message in response. This variable is initialized to 2, and constrained to the range [0.5, 2].

- *highReqParam*: The high-end request timer parameter. This variable is initialized to 2, and constrained to the range [1, 100].

- *lowRtxParam*: The low-end retransmission timer parameter. The retransmission timer is used in determining how long to wait between receiving an RTR message and sending an RTX message. This variable is initialized to 10, and constrained to the range [0.5, 10].

- *highRtxParam*: The high-end retransmission timer parameter. This variable is initialized to 10, and constrained to the range [1, 100].

- *avgDupReq*: The exponentially weighted moving average of the number of duplicate retransmission requests received during a period. This period is defined as the interval between a message loss notification and the completion of processing for that loss (either by sending a retransmission request or by the notification of the cancellation of processing for that loss) or of two successive message loss notifications.

- *avgReqDelay*: The exponentially weighted moving average of the duration of processing a message loss notification.

- *avgRtxDelay*: The exponentially weighted moving average of the duration of processing a retransmission request.

- *msgBuffer*: A collection of messages that this process has recently received.

- *stableTime*: Timestamp. Messages that were sent with a timestamp smaller than this variable do not need to be stored in the *msgBuffer*.

3.3 Retransmission Requests

Notification of a missing message is an event triggered externally. Upon receiving a notification, the protocol schedules a retransmission request. The scheduling uses random timeouts to suppress requests from multiple processes sharing a loss. The remainder of this section considers events that occur between the notification of a missing message, and the successful sending or cancellation of sending a retransmission request to the system.

The notification of a missing message is represented by a RTR Ticket. The scheduling timeout is randomly chosen from an interval, which is a function of this process's estimated distance to the sender of the missing message. The timeout is thus chosen from the uniform distribution

$$3^i * [lowReqParam * d_S, (lowReqParam + highReqParam) * d_S]$$

seconds, where d_S is the estimated distance from this process to the control process or coordinator associated with the sender of the missing message, obtained from the control hierarchy. The variables *lowReqParam* and *highReqParam* are adjustable via an adaptive algorithm and are discussed in more detail in [23] (where they are referred to as C_1 and C_2). The parameter i is the number of times the timeout calculation has been performed for the missing message. The parameter i is set to 0 in the first timeout calculation.

If the process receives a retransmission request for a message for which it has scheduled a timer, it performs a random exponential back-off. It recalculates the timeout with the parameter i incremented by 1 and reset the timer to the new timeout.

The protocol uses an adaptive algorithm to adjust the timer parameters *lowReqParam* and *highReqParam* in response to the past behavior of the reliable multicast protocol. Figure 3.1 shows the adaptive adjustment algorithm used to adjust these parameters.

```
After sending a request:
    $avgReqDelay$ = (1 - $\alpha$)*$avgReqDelay$ +
                    $\alpha$*(time it took last request)
    $lowReqParam$ = $lowReqParam$ - 0.05

Before each new request timer is set:
    $avgDupReq$ = (1 - $\alpha$)*$avgDupReq$ +
                  $\alpha$*(number of duplicate requests during the last period)
    if ($avgReqDelay \geq$ AVG_REQ_DELAY)
      $lowReqParam$ = $lowReqParam$ - 0.05
      if ($avgDupReq <$ (AVG_DUP_REQ - $\epsilon$)
        $highReqParam$ = $highReqParam$ - 0.1
    else if ($avgDupReq \geq$ AVG_DUP_REQ)
      $highReqParam$ = $highReqParam$ + 0.5
      if ($avgReqDelay <$ (AVG_REQ_DELAY - $\epsilon$))
        $lowReqParam$ = $lowReqParam$ + 0.05
    else if ( $avgDupReq <$ (AVG_DUP_REQ - $\epsilon$) AND
           $avgReqDelay <$ (AVG_REQ_DELAY - $\epsilon$))
             $highReqParam$ = $highReqParam$ - 0.1
```

Figure 3.1: Dynamic adjustment algorithm for the request timer parameters.

3.4 Retransmission of Data

A process that receives a retransmission request checks its *msgBuffer* for the message identified in the retransmission request. If that message appears in the *msgBuffer*, the process can fulfill the retransmission request.

If a process can fulfill a retransmission request, it schedules the sending of the message that is to be retransmitted. The timeout used for the scheduling is chosen from the uniform distribution

$$[lowRtxParam * d_R, (lowRtxParam + highRtxParam) * d_R],$$

where d_R is the estimated distance from this process to the control process or coordinator associated with the sender of the retransmission request, obtained from the control hierarchy. The variables *lowRtxParam* and *highRtxParam* are adjustable via an adaptive algorithm and are discussed in more detail in [23] (where they are referred to as D_1 and D_2). The algorithm is used to adjust the timer parameters *lowRtxParam* and *highRtxParam* in response to the past behavior of the reliable multicast protocols. Figure 3.2 shows the adaptive adjustment algorithm used to adjust these parameters.

If a process that has scheduled the retransmission of a message receives a

```
After sending a retransmission:
        $avgRtxDelay$ = (1 - $\alpha$)*$avgRtxDelay$ +
                            $\alpha$*(time it took last retransmission)
        $lowRtxParam$ = $lowRtxParam$ - 0.05

After the cancellation of a retransmission:
        $avgRtxDelay$ = (1 - $\alpha$)*$avgRtxDelay$ +
                            $\alpha$*(time it took last retransmission)
        if ($avgRtxDelay \geq$ AVG_RTX_DELAY)
          $highRtxParam$ = $highRtxParam$ - 0.05
        else
          $highRtxParam$ = $highRtxParam$ + 0.1

Before each new request timer is set:
        if ($avgRtxDelay \geq$ AVG_RTX_DELAY)
          $lowRtxParam$ = $lowRtxParam$ - 0.05
        else if ($avgRtxDelay <$ (AVG_RTX_DELAY - $\epsilon$))
          $lowRtxParam$ = $lowRtxParam$ + 0.05
```

Figure 3.2: Dynamic adjustment algorithm for the repair timer parameters.

retransmission of that message from another process, then this process cancels the timer and does not send the message. Otherwise, when the timer expires, the process sends the retransmission of the message.

3.5 Buffer Management

The *msgBuffer* is used to store messages, so that they can be retransmitted if necessary. To keep the *msgBuffer* size managable, we need to determine when a message will no longer be requested for retransmission. A message will no longer be requested for retransmission if that message has been received by every process. Such messages are referred to as *stable* messages. A stable message may be removed from the *msgBuffer*. We provide a protocol that determines message stability for buffer management.

The message stability protocol is based on the protocol in [10], that uses a timestamp acknowledgment mechanism for stability determination. Each process keeps track of the timestamp of the most recent "relevant" message delivered to the application. Every process determines what "relevant" means depending on its delivery guarantees and whether it is sending data messages. The only restriction is that the timestamp must not be larger than the timestamp of the next message to be delivered that was sent by that process. This value is periodically communicated to the control process associated with the process.

The control process gathers the timestamps from all of the processes associated with it, and chooses the smallest one to be that control process's *local acknowledgment*. Children send their local acknowledgments in the data portions of LCon messages. Coordinators gather the local acknowledgments from all of the control processes in their local group, and choose the smallest one to be the *group acknowledgment*. The group acknowledgment information is sent in the data portions of GCon messages. The coordinators gather the group acknowledgments from all of the control processes in the coordinator group, and choose the smallest one to be the *system acknowledgment*. The system acknowledgment is sent in the data portion of CCon messages. A control process receives the system acknowledgment from its coordinator and reports that value to the processes associated with it. All messages with timestamps smaller than the system acknowledgment are marked as stable and removed from the *msgBuffer*.

Chapter 4

Data Transmission and Delivery

In traditional group communication systems the delivery of messages from a process group to the application, is determined system-wide by the individual senders, or on a per message basis at a sender. Thus, every process must provide the same delivery guarantees to the application.

In the InterGroup system, the delivery of messages from a process group is determined by the application, at the receiving end. Each process may choose a different delivery guarantee. The InterGroup system currently provides the following delivery guarantees within a process group:

- **Unreliable unordered**. Messages received from the process group are delivered without delay for ordering or reliability to the application. Some messages might never be delivered and multiple copies of the same message might be delivered. Moreover, there is no guarantee on the order in which messages are delivered. The underlying communication network, IP Multicast, provides this same"best-effort" delivery service.

- **Reliable source ordered**. This service guarantees that all messages from a particular message sender will be delivered to the application (unless a process failure occurs) and they will be delivered in sequence number order for each of these sources. This service is similar to TCP/IP and is well suited to multicast file transfers and many applications currently using TCP/IP [15].

- **Reliable group timestamp ordered**. Messages are delivered to the application in timestamp order over the entire process group. A membership service ensures that the delivery of messages to the application obeys virtual synchrony. This service is closest to the idea of agreed messages in group communication systems.

In this chapter, we first outline the assumptions made in order to create the data transmission and delivery component (Section 4.1). Then, we present the data structures used (Section 4.2). A variation of a Lamport clock is presented next (Section 4.3). Then, we present the impact of message stability on the InterGroup protocols (Section 4.4). The data transmission algorithms are presented next (Section 4.5). Then, we present the ordering and delivery algorithms used by the different services offered by the InterGroup protocols (Sections 4.6 - 4.8). Finally, we discuss other delivery guarantees that might be added to the system in the future (Section 4.9).

4.1 Model

We consider an asynchronous distributed system consisting of n processes $p_1, p_2, \ldots, p_n$ that communicate via messages over a network. Each process within the system has a unique identifier. The system is asynchronous in that no bound can be placed on the time required for a computation or for communication of a message. Processes have access to Lamport clocks.

The system consists of m *process groups* $g_1, g_2, \ldots, g_m$. Each process group g_i consists of l_i processes $q_1, q_2, \ldots, q_{l_i}$ that communicate via messages over a network. Each process within the system may belong to multiple process groups. Each process group is uniquely mapped to an IP multicast group.[1]

The rest of the discussion is restricted to a single process group.

A *view* is defined as in Chapter 1. A view is uniquely identified by its *membership*, and *view identifier*. The membership consists of a set of process identifiers. The view identifier, consists of three fields:

- *startTime*: the logical time at which this view begins

[1] A process group is identified by the IP address of the IP multicast group it is using to communicate.

- *leader*: the identifier of a process chosen deterministically from the membership

- *leaderSeq*: the leader sequence number

The *current view* is the most recent view of the process group provided to the application (*i.e.,* the result of $viewof(t_i, p)$, where t_i is the most recent event and p is the process in question).

The process group comprises two disjoint groups: the *sender group*, which consists of the processes that must obey the Sender Self Inclusion property for the current view, and the *receiver group* which consists of all the other processes in the group. Each process in the process group is in only one of these groups at a time.

The rest of the discussion is restricted to a single view. Thus, the membership is static.

Only processes that are in the sender group multicast messages to the other processes. A process receives all of its own multicast messages. Communication between processes is unreliable and, thus, messages may be lost or arbitrarily delayed. Communication channels are not assumed to be FIFO.

4.2 Data Structures

The data transmission and delivery protocols employ three message types: Data messages, Keep Alive (KA) messages, and Application Data messages. We discuss these below.

4.2.1 Data Message

Data messages are used for exchanging application messages. A Data message contains a Data message header, and a byte array containing an application message. The following fields are contained in a Data message header:

- *type*: The message type of this message.

- *sender*: The unique identifier of the process that sent this message.

- *coord*: The unique identifier of the coordinator that was associated with the process that sent this message at the time this message was sent. This field is necessary, because we need to keep track of the coordinator associated with each process for use by the retransmission mechanisms.

- *seq*: The sequence number of this message. If another message from this sender has the same value of this field, that message must be identical to this one or a Keep Alive message. This field is used to detect message loss and allow FIFO ordering for messages from individual processes.

- *timestamp*: The timestamp of this message. This field is used to determine a process group wide ordering of messages.

The Data message header fields allow the protocols to satisfy the most stringent guarantees offered by the InterGroup protocols.

4.2.2 Keep Alive Message

Keep Alive (KA) messages are used by the protocols to help to detect lost messages, to ensure progress in the delivery of messages, and to help maintain liveness. They contain the fields of a Data message header.

4.2.3 Application Data Message

Application Data messages are used for the delivery of data messages to the application. The following fields are contained in an Application Data message:

- *type*: APP_DATA.

- *sender*: The unique identifier of the process that sent this message.

- *pg*: The unique identifier of the process group to which this message belongs.

- *data*: A byte array containing the application message.

4.3 A Variation of Lamport Time

All of the processes in the process group keep a local reference of the system time. This reference is based on a Lamport clock [37], and used to preserve the causality between messages. This algorithm is actually a modification of the Lamport time algorithm. It updates the Lamport time to match the local processor time whenever the Lamport time lags behind the local time. The local processor time is used as an efficiency. It allows the time value to increase in the absence of messages, providing a tighter synchronization mechanism for the clocks, and thus leading to lower delivery latencies for the reliable group timestamp ordered delivery service.

4.4 Determining Message Stability

To ensure that messages are available for retransmission to the processes of a process group, every process in the sender group must buffer a message it has sent until that message becomes *stable*. A process determines that a message is *locally stable* if the InterGroup protocols have finished processing the message at that process. A message becomes stable in the process group when every process in the process group has determined that the message is *locally stable*. A process will make the determination of local stability based on the delivery guarantee it chooses. Each process, periodically sends this information via the control hierarchy, where this information is distributed and the group stability is calculated.

4.5 Data Transmission

Processes in the sender group, multicast messages to the group. In this section, we discuss the algorithms for the transmission of data and the contents of the messages used by the protocol. First, we discuss the internal variables used.

4.5.1 Internal Variables

The transmission algorithms use the following variables:

- *myID*: The process identifier of this process.
- *coord*: The coordinator[2] associated with this process.
- *seq*: The sequence number of the last Data message sent.
- *time*: The current Lamport time.

4.5.2 Data Transmission Algorithms

The data transmission algorithms are quite simple. First, the Keep Alive timer is set. The process then waits for an event. The two possible events are: (1) reception of a message from the application, and (2) the expiration of the timer. If a message is received from the application, then the *seq* variable is incremented, a Data message including the application message is sent to the group, the timer is reset and the process waits for the next event. The header of the Data message is filled with the information from the internal variables previous to sending.

If the timer expires, a Keep Alive message is sent to the group, the timer is then reset and the process waits for the next event. The header of the Keep Alive message is contains the information from the internal variables previous to the message being sent.

4.5.3 Flow Control and Congestion Avoidance

The InterGroup system uses an adaptation of the Real Time Control Protocol (RTCP) flow control algorithms. We use the RTCP algorithm for calculating the time to wait until we send the next message. However, we vary the bandwidth parameter used in the calculation, depending on the message losses in the system. The bandwidth parameter is slowly increased while no losses are reported. Losses in the InterGroup system are reported using RTRMessages. When a process receives an RTRMessage, it reduces the bandwidth parameter, thus reducing the amount of traffic that the process attempts to send to the group.

[2]see Chapter 2 for definition

We also add a wrapper around the RTCP algorithm that checks to see if the process has too many outstanding messages. If that is the case, the process sends Keep Alive messages until the number of outstanding messages falls below a threshold. This is the fallback feedback mechanism in our flow control. If a process does not manage to receive RTRMessages which signal it to slow down its sending, this mechanism abruptly stops sending new messages until the system stabilizes.

This approach is conservative. We have not tried to optimize the flow control mechanisms. The topic of flow control is still an open research topic and is not the focus of this dissertation.

4.6 Unreliable Unordered Delivery

The unreliable unordered (UU) delivery service provides the delivery services of IP multicast, *i.e.,* "best-effort" delivery. The InterGroup protocols receive messages via IP multicast. Upon reception, all Data messages are delivered to the application and all other messages (including Retransmitted messages[3]) are discarded. When this delivery service is chosen, the protocol makes no effort to order messages or to recover lost messages. The messages are delivered to the application without delay for ordering or reliability.

Applications receiving messages using the UU delivery service, may suffer additional overhead, when compared to the delivery service offered by IP multicast. This is due to two factors. First, every message sent using the InterGroup protocols is converted to a Data message before sending. Thus, there is an additional header associated with the message, increasing bandwidth usage and additional latency in creating the Data message. Second, prior to delivery to the application, the message is converted to an Application Data message, which incurs additional latency.

Due to the nature of this service, messages do not need to be buffered at the receiver. However, as an efficiency, the protocol may buffer Data messages locally at the receiver to satisfy retransmission requests.

[3]See Chapter 3 for definition.

4.7 Reliable Source Ordered Delivery

This service guarantees that all messages from a source will be delivered to the application (unless a process failure occurs at the source) in sequence number order, provided that the source is in the membership of the current view at the receiver. This service is similar to TCP/IP and, thus, well-suited to multicast file transfers and many applications currently using TCP/IP.

4.7.1 Setup

An application specifies a source to activate this service. The receiver and source then have to determine the starting point of the service. First, the receiver adds this source to its local stability reports. This step prevents new messages from becoming stable in this process group until the receiver successfully activates the service. The protocol then performs a handshake with the source. Through the handshake, the source provides the receiver with an initial membership view and the sequence number of the first message to be delivered from the source. The view is delivered to the application and thus the source becomes an *active source* and the sequence number becomes the *activation point* for this source. If the handshake protocol is unsuccessful, the source is removed from the receiver's local stability reports, and the protocol notifies the application that the service is currently unavailable.

A user may choose more than one source for this delivery service. If multiple sources are chosen, there is no guarantee on the interleaving of message delivery across the sources.

4.7.2 Ordering and Delivery

The first message delivered to the application is the message from the source whose sequence number matches the activation point. The ordering and delivery algorithms for this service deliver the messages from this source, with sequence numbers greater than the activation point in FIFO order, as long as the source is a member of the view installed at this process.

Because the underlying communication network may lose messages, an algorithm is provided to detect missing messages (Figure 4.1). The algorithm

```
constants:
    activationPoint[source] - the sequence number of the first message
                              delivered to the application from the process
                              indexed by source
variables:
    highSeq[source] - the largest sequence number received from
                      the process indexed by source

algorithm:
    receive message msg
    if (msg.sender is an active source)
      if (msg.type = KEEP_ALIVE)
        if (msg.seq > highSeq[msg.sender])
          request retransmission of messages with sequence numbers
              (highSeq, seq] for msg.sender
      else
        if (msg.seq - 1 > highSeq[msg.sender])
          request retransmission of messages with sequence numbers
              (highSeq, seq) from msg.sender
```

Figure 4.1: Algorithm for detecting missing messages.

detects missing messages by looking for gaps in the sequence numbers of the messages received from this source. Keep Alive messages are used to aid in loss detection. The sequence number of a Keep Alive message is equal to the sequence number of the most recent Data message sent by the source. Thus, Keep Alive messages allow us to determine the loss of the most recent Data message sent by the source. When a message loss is detected, a request for the retransmission of the missing message is sent to the reliable multicast component.

The algorithm in Figure 4.2 shows the ordering and delivery process for this service.

```
constants:
   $activationPoint[source]$ - the sequence number of the first message
                               delivered to the application from the process
                               indexed by source
variables:
   $highSeq[source]$ - the largest sequence number received from
                       the process, indexed by source
   $lastSeq[source]$ - the largest sequence number delivered from
                       the process, indexed by source

initialization:
   $lastSeq[source] = activationPoint[source]$

algorithm:
   receive message $msg$
   if ($msg.type = DATA$)
     if have not received a data message denoted by ($msg.sender$, $msg.seq$)
       if ($msg.sender$ is an active source)
         if ($msg.seq = lastSeq[msg.sender] + 1$)
           updateAndDeliver($msg$)
           while (have stored message ($msg$) with $msg.seq = lastSeq[msg.sender] + 1$)
           updateAndDeliver($msg$)
         else
           store $msg$
           if ($msg.seq > highSeq[source]$)
             $highSeq[msg.sender] = msg.seq$
       else
         store $msg$
         if ($msg.seq > highSeq[source]$)
           $highSeq[msg.sender] = msg.seq$

updateAndDeliver($msg$):
   $lastSeq[msg.sender]$++
   if ($highSeq[msg.sender] < msg.seq$)
     $highSeq[msg.sender] = msg.seq$
   deliver $msg$
```

Figure 4.2: Algorithm for ordering and delivering messages for the reliable source ordered message delivery service.

4.8 Reliable Group Timestamp Ordered Delivery

This service guarantees that messages are reliably delivered to the application in timestamp order across the sender group membership of the current view. Furthermore, it guarantees that the messages are delivered in the same order to all applications that have the same current view and have requested the same delivery service.

At the start of a view, the protocol sets the activation point for each process in the membership of that view. Only messages after the activation point for each process are considered for delivery. These activation points are determined by the membership protocols (see Chapter 5). For this service, messages are delivered from the members of the view in timestamp order.

The algorithm used to detect missing messages for this service is the same as the algorithm for the reliable source ordered message delivery service (Figure 4.1).

The algorithms in Figures 4.3 and 4.4 show the ordering and delivery process for this service. The messages are first ordered in sequence number order for each source in the view (Figure 4.3). The algorithm that performs the source ordering is similar to the one for the reliable source ordered message delivery service. However, the timestamp ordering algorithm delivers a message to the application only if a message from each of the processes in the membership of the current view is currently stored in that ordering component. To allow progress in the delivery of messages to the application in the absence of Data Messages, the source ordering algorithm in Figure 4.3 also processes Keep Alive messages. Keep Alive messages contain a timestamp that is greater than the timestamp of the most recent Data message sent at a source. Thus, if there are no Data messages available for delivery in the source ordering algorithm, a Keep Alive message may be delivered to allow progress of the timestamp ordering algorithm.

The algorithm in Figure 4.4 assumes that messages are received reliably and in source order for all of the processes in the membership of the current view. This is accomplished by the source ordering algorithm. A message is delivered to the application only if a message from each of the processes in the membership of the current view is currently stored in that ordering

```
constants:
   activationPoint[source] - the sequence number of the first message
                             delivered to the application from the process
                             indexed by source
variables:
   highSeq[source] - the largest sequence number received from
                     the process, indexed by source
   lastSeq[source] - the largest sequence number delivered from
                     the process, indexed by source
   highTime[source] - the largest timestamp received from the
                      process, indexed by source

initialization:
   lastSeq[source] = activationPoint[source]

algorithm:
   receive message msg
   if (msg.type = KEEP_ALIVE)
     if (msg.sender is an active source)
       if (msg.seq > lastSeq[msg.sender])
         if (highTime[msg.sender] < msg.timestamp)
           highTime[msg.sender] = msg.timestamp
           deliver msg
       else if (msg.seq > lastSeq[msg.sender])
         if have received a data message (dm) denoted by
           (msg.sender, msg.seq)
                if (dm.timestamp < msg.timestamp)
                  dm.timestamp = msg.timestamp
   else if have not received a data message denoted by (msg.sender, msg.seq)
       if (highTime[msg.sender] < msg.timestamp)
         highTime[msg.sender] = msg.timestamp
       if (msg.sender is an active source)
         if (msg.seq = lastSeq[msg.sender] + 1)
           updateAndDeliver(msg)
           while (have stored message (msg) with msg.seq = lastSeq[msg.sender] + 1)
           updateAndDeliver(msg)
         else
           store msg
           if (msg.seq > highSeq[source])
             highSeq[msg.sender] = msg.seq
       else
         store msg
         if (msg.seq > highSeq[source])
           highSeq[msg.sender] = msg.seq

updateAndDeliver(msg):
   lastSeq[msg.sender]++
   if (highSeq[msg.sender] < msg.seq)
     highSeq[msg.sender] = msg.seq
   deliver msg
```

Figure 4.3: Algorithm for source ordering messages for the reliable timestamp group ordered message delivery service.

```
variables:
   queue[source] - a FIFO queue, holding the messages that are
                   ready to be checked for delivery for the process
                   denoted by source
   memb - a set containing the processes in the sender group of the
          current configuration

algorithm:
   receive message msg
   if (msg.type = KEEP_ALIVE)
     if (have stored a keep alive message (kam) denoted by (msg.sender, msg.seq))
       if (msg.timestamp > kam.timestamp)
         replace kam with msg
     else
       add msg to queue[msg.sender]
   else
     add msg to queue[msg.sender]
     remove all KEEP_ALIVE messages from the queue
   loop
     msgToDeliver.timestamp = $\infty$
     for (for all $x \in$ memb)
        if (queue[x] is empty)
          exit loop
        tmpMsg = get next message from queue[x]
        if (tmpMsg.timestamp < msgToDeliver.timestamp)
          msgToDeliver = tmpMsg
        else if (tmpMsg.timestamp = msgToDeliver.timestamp AND
              tmpMsg.sender < msgToDeliver.sender)
               msgToDeliver = tmpMsg
     remove next message from queue[msgToDeliver.sender]
     deliver msgToDeliver
   end loop
```

Figure 4.4: Algorithm for ordering and delivering messages for the reliable group timestamp ordered message delivery service.

component. The message that is delivered is the one with the lowest timestamp. If more than one message carries the lowest timestamp, the algorithm delivers the message, from this set, that is from the process with the lowest process identifier. If this message is a Keep Alive message, the algorithm does not deliver it to the application.

4.9 Future Work

The InterGroup protocols are designed to allow for other delivery services. In the future we plan to add the following delivery services:

- **Unreliable source ordered**. The messages are delivered to the application in sequence number order for each source. Once a message with sequence number *seq*, from source *src*, has been delivered to the application any message with a lower sequence number, from source *src* is discarded. This type of service is commonly used for video and audio data. The Real Time Protocol (RTP)[53] provides a similar type of service.

- **Unreliable group timestamp ordered**. Messages are delivered to the application in timestamp order over the entire process group. This type of service can be thought of as an extension of the unreliable source ordered service. It may be used when synchronization of unreliable data (*e.g.,* video and audio) from different sources is needed.

- **ALF-enabled service**. This service provides the application with the ability to request the retransmission of a message it considers to be missing. This service is based on application level framing (ALF) [17]. To implement this service in the InterGroup protocols, the headers of the data message types would need to be changed, to allow for a scalable naming structure as in [49].

Ordering across groups can also be implemented by building components on top of the existing delivery services.

Chapter 5

Process Group Membership

The process group membership is used to maintain a valid view of the group memberships and allow the delivery of messages to continue.

5.1 Model

We consider an asynchronous distributed system consisting of n processes $p_1, p_2, \ldots, p_n$ that communicate via messages over a network. Each process within the system has a unique identifier. The system is asynchronous in that no bound can be placed on the time required for a computation or for communication of a message. Processes have access to Lamport clocks.

The system consists of m *process groups* $g_1, g_2, \ldots, g_m$. Each process group g_i consists of l processes $q_1, q_2, \ldots, q_l$ that communicate via messages over a network. Each process within the system may belong to multiple process groups.

The rest of the discussion is restricted to a single process group.

A *view* is defined as in Chapter 1. A view is uniquely identified by its *membership*, and *view identifier*. The membership consists of a set of process identifiers. The view identifier consists of three fields:

- *timeEntered*: the logical time at which this view begins
- *leader*: the identifier of a process chosen deterministically from the membership

- *leaderSeq*: the leader sequence number

The *current view* is the most recent view of the process group provided to the application (*i.e.,* the result of $viewof(t_i, p)$, where t_i is the time of the most recent event and p is the process in question).

The process group comprises two disjoint groups: the *sender group*, which consists of the processes that must obey the Sender Self Inclusion property for the current view, and the *receiver group* which consists of all the other processes in the group. Each process in the process group is in only one of these groups at a time.

Only processes, that are in the sender group or that wish to join the sender group, multicast messages to the other processes. Multicast messages are assumed to be delivered to the membership protocols, reliably and in source order.

The network is allowed to partition and remerge.

We assume all process have chosen the Reliable Group Timestamp Ordered delivery guarantee.

5.2 The Data Structures

The process group membership protocol employs a number of message types to ensure that a valid membership view is executing. First, we introduce the messages that are internal to a process, and are not sent out on the network. Then, we discuss the data message type whose fields are included in all reliable messages sent over the network. Next, we present the network messages used in the membership protocols. Finally, we discuss the internal data structures.

5.2.1 Internal Messages (Events)

Process Failure Message

Process Failure (PFailure) messages are used to signal the suspicion that a process in the current view or proposed membership has failed. A PFailure message contains the following fields:

- *type*: PFailure

- *proc*: The process identifier of the process that is suspected to have failed.

Process Foreign Message

Process Foreign (PForeign) messages are used to signal the reception of a message from a process outside the current view. A PForeign message contains the following fields:

- *type*: PForeign
- *proc*: The process identifier of the process that is suspected not to be in the current view.

To Sender Message

To Sender (ToSender) messages are used to signal that this process has decided to enter the sender group. A ToSender message contains the following fields:

- *type*: ToSender

To Receiver Message

To Receiver (ToRec) messages are used to signal that this process has decided to exit the sender group. A ToRec message contains the following fields:

- *type*: ToReceiver

User Membership Change Message

User Membership Change (UMC) messages are used to signal a view change to the application. They are delivered to the application at the point in the data stream when the view change occurs. A UMC message contains the following fields:

- *type*: UMC

- *membID*: The unique identifier of the view whose beginning this message signals.
- *cut*: The time at which this view begins.
- *memb*: The set of processes in the sender group of the view whose beginning this message signals.
- *transSet*: The set of processes that were in the sender group of the previous view, and are in the sender group of the view whose beginning this message signals.

5.2.2 Network Messages

Membership Change Message

This message extends the Data message. Membership Change (MC) messages signal that the new view should be installed. A MC message contains the following field which is a constant:

- *type* = MC

The following fields are added for a MC message:

- *membID*: The unique identifier of the new view.
- *procs*: The set of all of the identifiers of the processes that are in the sender group of the new view.
- *oldIDs*: A set of identifiers of all of the latest views of all processes in *procs*.
- *cutInfo*: An array indexed by the identifiers in *oldIDs*, whose values represent the last message delivered in the view represented by the index.
- *lastSeq*: An array indexed by the identifiers in *procs*, whose values represent the sequence number of the first message to be delivered from the process represented by the index.

Process Add Message

This message extends the Data message. Process Add (PAdd) messages are used for voluntary joins to the sender group. They are sent by a process in the sender group of the current view, on behalf of a process that wishes to join the sender group. A PAdd message contains the following field which is a constant:

- *type* = PAdd

The following field is added for a PAdd message:

- *proc*: The unique identifier of the process that wishes to join the sender group.

Process Leave Message

This message extends the Data message. Process Leave (PLeave) messages are used for voluntary leaves from the sender group. A process that wishes to leave the sender group, creates this message and sends it to the process group. A PLeave message contains the following field which is a constant:

- *type* = PLeave

Process Join Message

This message extends the Data message. Process Join (PJoin) messages are used to reach consensus on the processes that will be in the sender group of the next view (*proposed membership*). A PJoin message contains the following field which is a constant:

- *type* = PJoin

The following fields are added for a PJoin message:

- *membID*: The unique identifier of the current view at the message sender.

- *procs*: A set of identifiers of the processes that, at the time this message was sent, were being considered for the sender group of the next view at the message sender.
- *fail*: A set of identifiers of the processes that, at the time this message was sent, were considered as having failed. This set is a subset of procs.

Process Failure Block Message

This message extends the Data message. Process Failure Block (PFailBlock) messages are used to signal a suspicion that a process in the current view or proposed membership has failed. These messages are generated if this process is not able to receive a retransmission of a message for a specified period of time. A PFailBlock message contains the following field which is a constant:

- *type* = PFailBlock

The following fields are added for a PFailBlock message:

- *proc*: The unique identifier of the process that is suspected to have failed.
- *procSeq*: The sequence number of the message sent by *proc* that has not been received.

Recovery Information Message

This message extends the Data message. Recovery Information (RecInfo) messages are used to distribute the information regarding the time when the view change should occur. A RecInfo message contains the following field which is a constant:

- *type* = RecInfo

The following fields are added for a RecInfo message:

- *membID*: The unique identifier of the current view.
- *cut*: The timestamp of the last message delivered to the user.

- *seqCut*: An array referenced by the unique identifier of the processes in the sender group of the current view that have been tagged as being suspected of failing. The values in the array are the sequence numbers of the last message that the sender of this message considers can be delivered in reliable source order from each of these processes.

Ready To Commit Message

This message extends the Data message. Ready To Commit (RTC) messages are used to signal that the sending process is ready to install the proposed membership. A RTC message contains the following field which is a constant:

- *type* = RTC

The following field is added for a RTC message:

- *firstSeq*: The sequence number of the first message to be delivered from this process in the next view, if it is installed.

5.2.3 Internal Variables

The membership protocols use the following internal variables:

- *state*: The state of the membership protocol. These states are described in Section 5.3.

- *blockSet*: The set of processes that are blocked from consideration for further memberships. Processes are added to this set during an execution of the MRA. Processes are removed from this set after a certain amount of time (this amount of time should be set to at least 100 times the expected latency between the most distant processes in the group), though they may not be removed while in the *MRA* state.

5.3 The Process Group Membership Protocol

The process group membership protocol is used to keep the processes in the process group executing correctly and with consistent views of the membership boundaries, despite changes in the membership (voluntary or not). We describe here the states of the membership protocol and the processing involved to keep the process group executing. There are two states in the membership protocol that may be considered as normal or stable. These are the *Receiver* state and the *Sender* states. The process will always tend to one of these states.

First, we describe how the membership protocol is initialized. After that, we describe what occurs when a process is forced out of one of the stable states. The protocols used in the case of faults in the process group (the repair algorithms) are described in detail in Sections 5.4 and 5.5.

5.3.1 Bootstrap

The state machine for the initialization or bootstrap of the membership protocols is shown in Figure 5.1. When a process wishes to join a process group, it must do so using the membership protocol, entering through the *Bootstrap* state. A joining process may specify whether it wishes to join the sender group directly, or whether it wishes to join the process group as a receiver. This is specified by an argument to the membership protocols and stored in the variable *initialState*. The allowed values for *intialState* are SENDER (in case the process wishes to join the sender group directly) and RECEIVER (otherwise).

The *Bootstrap Receiver* state is entered after the *initialState* variable is initialized. A process's objective in this state is to obtain enough information about the sender group of the current view, so that this process can join the process group as a receiver. To obtain this information, the process must first find a process that is a member of a sender group in this process group. This step is accomplished by receiving messages addressed to the process group and attempting to contact the senders of those messages. Once a process in the sender group is contacted, the necessary information is obtained from that process. This information consists of:

- The unique identifier of the current view.
- The identifiers of the process in the sender group of the current view.
- A sequence number for each process in the sender group of the current view. This process will begin reliable delivery of messages for each of

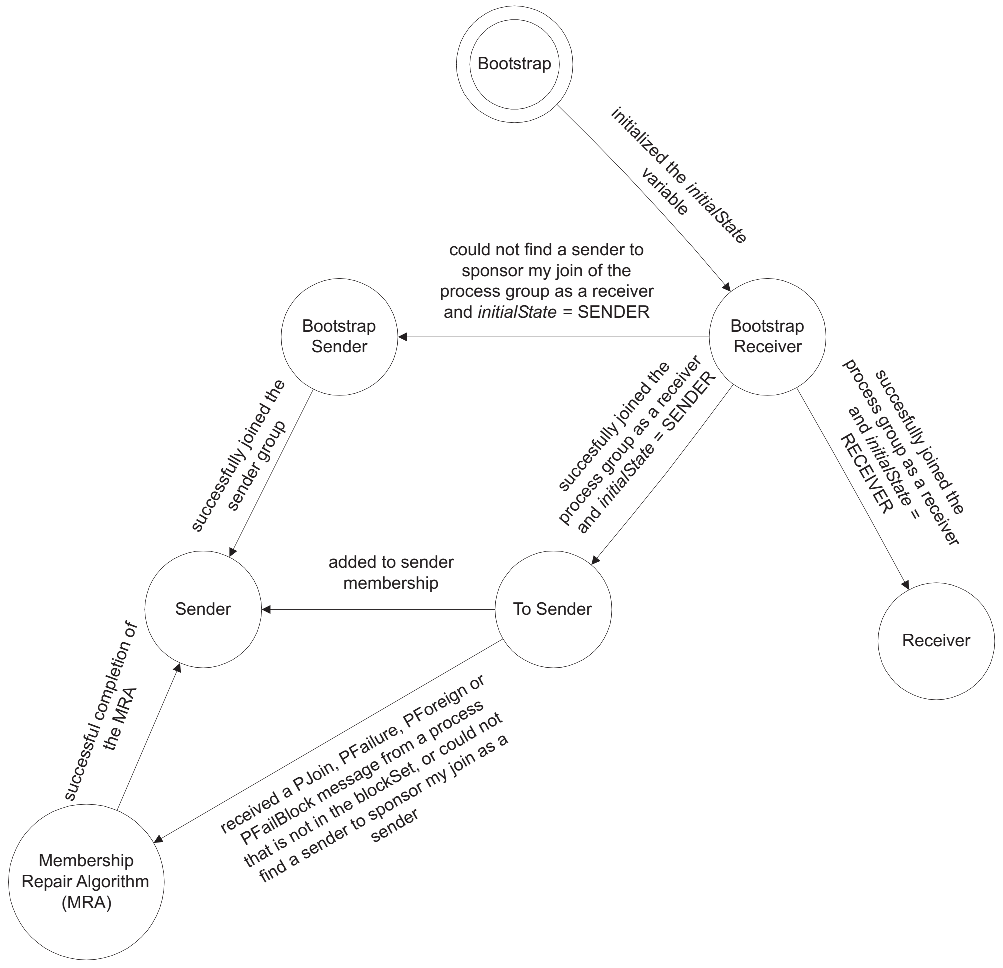
Bootstrap
initialized the *initialState* variable
could not find a sender to sponsor my join of the process group as a receiver and *initialState* = SENDER
Bootstrap Sender
Bootstrap Receiver
successfully joined the sender group
succesfully joined the process group as a receiver and *initialState* = SENDER
succesfully joined the process group as a receiver and *initialState* = RECEIVER
added to sender membership
Sender
To Sender
Receiver
successful completion of the MRA
received a PJoin, PFailure, PForeign or PFailBlock message from a process that is not in the blockSet, or could not find a sender to sponsor my join as a sender
Membership Repair Algorithm (MRA)

Figure 5.1: State machine for the initialization of the membership protocols.

these processes with the message that has a sequence number matching this sequence number.

- The timestamp at which this process will install the current view, *i.e.,* this process will deliver messages, after this timestamp, in the current view.

Once this process installs the current view, it enters the *Receiver* state if *initialState* is RECEIVER and finishes the bootstrap process. If this process installs the current view and *initialState* is SENDER, it enters the *To Sender* state. If this process is not able to contact a member of the sender group and *initialState* is SENDER, it enters the *Bootstrap Sender* state.

When a process that is in the receiver group, wishes to join the sender group, it enters the *To Sender* state. First, the process tries to find a process in the sender group of the current view that will sponsor its join of the sender group. If it cannot find one, it transitions to the *MRA* state, described in Section 5.4. If this process does find such a process, called a *sponsor*, this process requests its sponsor to send a PAdd message with this process' unique identifier as the *proc* field of that message. Once this request is confirmed, this process waits for that PAdd message to be delivered to the user. If the request is not confirmed, this process looks for another sponsor. If one of the following events occurs while it awaits delivery of the PAdd message:

- Arrival of a PJoin message from a process that is not in the *blockSet*.
- Arrival of a PFailure message.
- Arrival of a PForeign message regarding a process that is not in the *blockSet*.
- Arrival of a PFailBlock message from a process that is in the sender group of the current view.

this process transitions to the *MRA* state. Once the PAdd message adding this process to the sender group (actually a UMC message denoting the start of a new view in which this process is in the sender group) is delivered, this process transitions to the *Sender* state and the bootstrap is finished.

The *Bootstrap Sender* state is used to add this process to the sender group, if *initialState* is SENDER and this process couldn't successfully join the process group as a receiver first. The processing inside this state is the same as in the *MRA* state. The sending of user messages is disabled until the repair algorithm is concluded and this process transitions to the *Sender* state, upon which the bootstrap is finished.

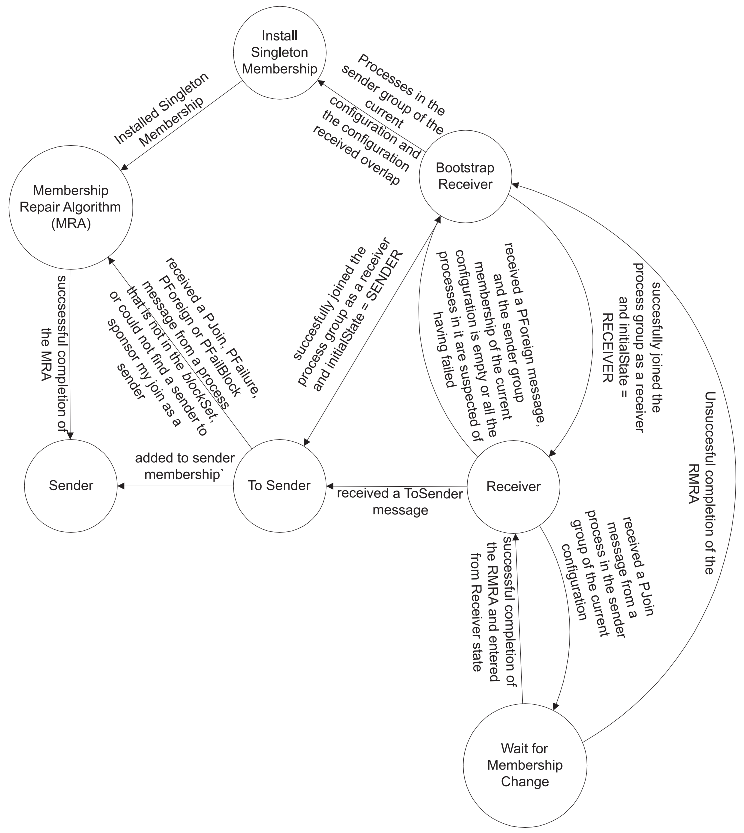
Install Singleton Membership
Installed Singleton Membership
Processes in the sender group of the current configuration and the configuration received overlap
Membership Repair Algorithm (MRA)
Bootstrap Receiver
successful completion of the MRA
received a PJoin, PFailure, PForeign or PFailBlock message from a process that is not in the blockSet, or could not find a sender to sponsor my join as a sender
succesfully joined the process group as a receiver and initialState = SENDER
received a PForeign message, and the sender group membership of the current configuration is empty or all the processes in it are suspected of having failed
succesfully joined the process group as a receiver and initialState = RECEIVER
Unsuccesful completion of the RMRA
Sender
added to sender membership`
To Sender
received a ToSender message
Receiver
received a PJoin message from a process in the sender group of the current configuration
successful completion of the RMRA and entered from Receiver state
Wait for Membership Change

Figure 5.2: How the *Receiver* state is left in the membership protocols.

5.3.2 Leaving the Receiver State

The state diagram showing what happens when a process leaves the *Receiver* state is shown in Figure 5.2. When a process is in the *Receiver* state, it is in a normal or stable state. The process executes as if the membership of the process group was static. There are only four reasons for the process to leave this state: (1) it wishes to join the sender group, (2) a process in the sender group of the current view has started a membership repair algorithm, (3) this process suspects that all of the processes in the sender group have failed and it has received a message from a process that is in a sender group of another view, or (4) the sender group is empty in the current view and this process has received a message from a process that is in a sender group of another view. We describe the results of these four scenarios below, starting from the last and moving forward.

A view with an empty sender group can be installed only if the last member of the sender group leaves the group voluntarily. At this point, the process in the *Receiver* state has nothing to do. If the process receives a PForeign message while the sender group is empty, it behaves as if it is initializing the membership protocol with *initialState* set to RECEIVER. Thus, it transitions to the *Bootstrap Receiver* state and follows the transitions described previously.

If this process suspects that all of the processes in the sender group have failed, it behaves as if the sender group were empty, and switches to the *Bootstrap Receiver* state. However, if any of the processes that were in its sender group appear in the sender group of the view it receives while it is in the *Bootstrap Receiver* state, it follows a new transition. In this case it transitions to the *Install Singleton Membership* state, where it installs a view whose sender group consists of only itself. Once that is accomplished, it transitions to the *MRA* state.

When a process in the sender group starts a membership repair algorithm, it sends a PJoin message as part of that algorithm. Thus, on reception of a PJoin message from a process in the sender group of the current view, this process, in the *Receiver* state, transitions to the *Wait for Membership Change* state, described in Section 5.5.

When a process in the *Receiver* state wishes to join the sender group, it switches to the *To Sender* state. The *To Sender* state and the transitions from it were described in Section 5.3.1.

5.3.3 Leaving the Sender State

The state diagram that indicates what happens when a process leaves the *Sender* state is shown in Figure 5.3. When a process is in the *Sender* state, it is in a normal or stable state. The process executes as if the membership of the process group were static. There are only three reasons for the process to leave this state: (1) it wishes to leave the sender group, (2) a process in the sender group is suspected of having failed, or (3) it receives a message from a process that is not in the sender group of the current view. We describe these

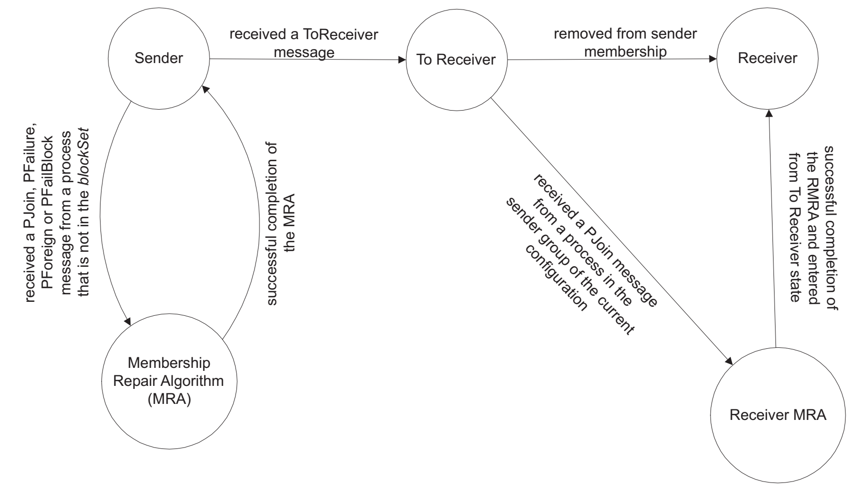
Sender
received a ToReceiver message
To Receiver
removed from sender membership
Receiver
received a PJoin, PFailure, PForeign or PFailBlock message from a process that is not in the blockSet
successful completion of the MRA
received a PJoin message from a process in the sender group of the current configuration
successful completion of the RMRA and entered from To Receiver state
Membership Repair Algorithm (MRA)
Receiver MRA

Figure 5.3: How the *Sender* state is left in the membership protocols.

three scenarios below.

If a process that is in the *Sender* state wishes to leave the sender group, it sends a PLeave message to the process group, stops sending new messages to the group, and transitions to the *To Receiver* state. Once in the *To Receiver* state, this process waits for the PLeave message it sent to be delivered to the user. When this message is delivered, the process transitions to the *Receiver* state. However, if the process receives a PJoin message from a process in the sender group of the current view before it receives the PLeave message, then the process transitions to the *Wait for Membership Change* state, described in Section 5.5.

If a process in the sender group of the current view is suspected of having failed, one of the following events occurs:

- The fault detector of this process generates a PFailure message concerning the process suspected to have failed, and the membership protocol receives that message.

- The process receives a PJoin message from a process in the sender group of the current view.

- The process receives a PFailBlock message from a process in the sender group of the current view.

The occurrence of any one of these events will cause the process to transition out of the *Sender* state into the *MRA* state, described in Section 5.4.

If the process receives a PForeign message regarding a process that is not in the sender group of the current view, and is not in the *blockSet*, it transitions to the *MRA* state. This event signals that there is at least one more view installed at a different member of the process group. By switching to the *MRA* state and running the repair algorithm, the protocol attempts to merge these views into one.

5.4 Membership Repair Algorithm

The membership repair algorithm (MRA) is executed at a process that wishes to be in the sender group of the next view. It is run while the process is in the *Bootstrap Sender* state, described in Section 5.3.1, and in the *MRA* state of the process group membership protocol. The entry conditions of the *MRA* state have been discussed in Section 5.3. A process leaves the *MRA* state only after it successfully completes the MRA and then transitions to the *Sender* state.

We discuss the MRA in detail, because it allows the membership protocols to deal with failures of processes, partitioning of the process group, and merging of components of the partitions. We discuss the messages and internal variables first, followed by the protocol.

5.4.1 Data Structures

The MRA employs the following messages: PJoin, PFailure, PFailBlock, RecInfo, RTC, MC and UMC. All of these messages are described in Section 5.2. In addition to the *blockSet* variable common to the entire membership protocol, the MRA uses the following internal variables:

- *procSet*: The set of processes that are being considered for the next membership. This set can only grow during the run of an MRA.

- *failSet*: The set of processes that are being considered as failed for the next membership. This set is a subset of *procSet*. It can only grow during the execution of an MRA.

- *lastProcTable*: A mapping where the keys are processes in the setMinus (*procSet*, *failSet*) and the entries are sets containing the last known *procSet* for each of these processes. The sets are obtained from PJoin messages sent by the processes.

- *lastFailTable*: A mapping where the keys are processes in the setMinus (*procSet*, *failSet*) and the entries are sets containing the last known *failSet* for each of these processes. The sets are obtained from PJoin messages sent by the processes.

5.4.2 The Protocol

The state machine for the MRA is shown in Figure 5.4. The process enters the MRA by entering the *Initial* state, where the MRA is initialized. The delivery of messages to the user is halted and the sending of PLeave messages is disallowed. The delivery of messages to the user needs to be halted, because a *cut* (the place in the data stream delivered to the user) where the current view ends needs to be agreed upon in the membership repair algorithm. If message delivery were allowed, the cut that a process sends to the process group might not be valid. The sending of PLeave messages is disallowed, because it would cause the protocol to violate the termination property.

Upon the completion of these steps, the process transitions to the *Consensus* state. In the *Consensus* state the processes participating in the MRA attempt to reach agreement on the processes that will be in the sender group of the next view. The pseudo-code for the *Consensus* state is shown in Figure 5.5. Upon the initial entry to the *Consensus* state, the *procSet* is initialized to the sender group of the current view, and the *failSet* is initialized to the empty set. The main loop of the consensus algorithm receives membership messages, updates the appropriate variables, and sends membership messages in response. The termination condition for the consensus algorithm is that all of the processes in the set difference of the *procSet* and the *failSet*, called the

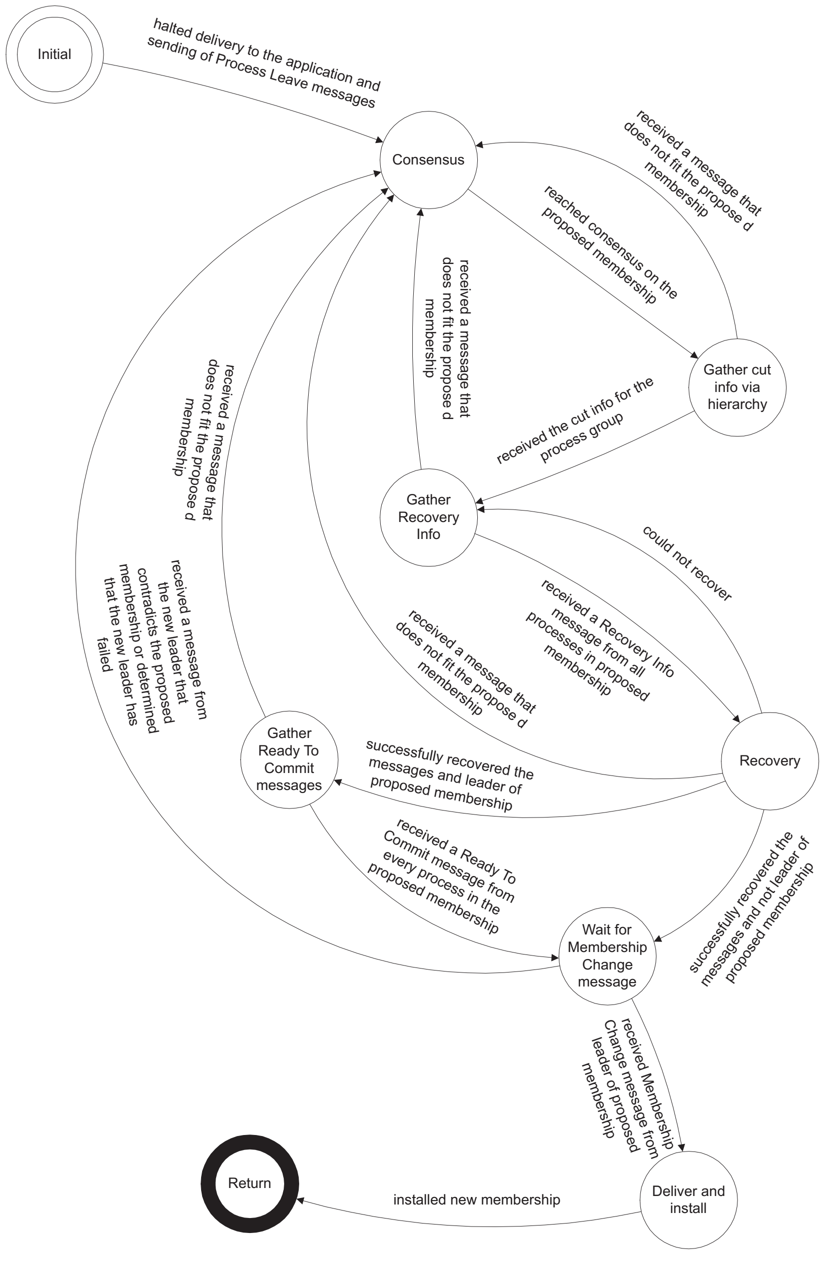
Initial
halted delivery to the application and sending of Process Leave messages
Consensus
received a message that does not fit the propose d membership
reached consensus on the proposed membership
Gather cut info via hierarchy
received the cut info for the process group
received a message that does not fit the propose d membership
Gather Recovery Info
could not recover
received a Recovery Info message from all processes in proposed membership
received a message that does not fit the propose d membership
received a message that does not fit the propose d membership
received a message from the new leader that contradicts the proposed membership or determined that the new leader has failed
Recovery
Gather Ready To Commit messages
successfully recovered the messages and leader of proposed membership
successfully recovered the messages and not leader of proposed membership
received a Ready To Commit message from every process in the proposed membership
Wait for Membership Change message
received Membership Change message from leader of proposed membership
Deliver and install
installed new membership
Return

Figure 5.4: The Membership Repair Algorithm

```
consensus:
    if (first time in consensus during this execution of the MRA)
       $procSet = currentMembSet$
       $failSet = \emptyset$
    multicast a PJoin message
    while consensus not reached
           $msg$ = next message
           switch ($msg.type$)
           case PJoin:
                 if processJoin($msg$) return (consensus reached)
           case PForeign:
                 $procSet = procSet \cup \{ msg.sender \}$
                 multicast a PJoin message
           case PFailure:
                 $procSet = procSet \cup \{ msg.proc \}$
                 $failSet = failSet \cup \{ msg.proc \}$
                 multicast a PJoin message
           case PFailBlock:
                 if (this process has ever delivered a message from $msg.proc$
                    with sequence number $msg.procSeq$)
                          $failedProc = msg.sender$
                 else
                          $failedProc = msg.proc$
                 $blockSet = blockSet \cup \{ failedProc \}$
                 if ($failedProc \in procSet$)
                    $failSet = failSet \cup \{ failedProc \}$
                    multicast a PJoin message

boolean processJoin($msg$)
    if ($msg.procs = \emptyset$)
       multicast a PJoin message
       return false
    if ($myID \in msg.fail$)
       $procSet = procSet \cup \{ msg.sender \}$
       $failSet = failSet \cup \{ msg.sender \}$
       multicast a PJoin message
       return false
    if ($msg.sender \notin procSet$)
       incorporateJoin($msg$)
       multicast a PJoin message
       return false
    if ($msg.sender \notin failSet$)
       if ($msg.procs \not\subset procSet$ OR
          $msg.fail \not\subset failSet$)
               incorporateJoin(msg)
               multicast a PJoin message
       else
               incorporateJoin($msg$)
               if (consensusReached())
                  return true
    return false

incorporateJoin($msg$)
    $procSet = procSet \cup msg.procs$
    $lastProcTable[msg.sender] = msg.procs$
    $failSet = failSet \cup msg.fail$
    $lastFailTable[msg.sender] = msg.fail$

boolean consensusReached()
    $propMemb = \{ procSet - failSet \}$
    for every $proc \in propMemb$
        if ($procSet \neq lastProcTable[proc]$)
           return false
        if ($failSet \neq lastFailTable[proc]$)
           return false
    return true
```

Figure 5.5: Pseudocode for consensus.

The membership repair algorithm might "believe" that the message in question does not fit the proposed membership for one or more of the following reasons:

- The message is a PJoin message whose *procs* set is empty.
- The message is a PJoin message whose *sender* is not in the *procSet*.
- The message is a PJoin message whose *procs* set is not a subset of *procSet* and the *sender* is in the proposed membership.
- The message is a PJoin message whose *fail* set is not a subset of *failSet* and the *sender* is in the proposed membership.
- The message is a PForeign message whose *proc* identifier is not in the *procSet*.
- The message is a PFailure message whose *proc* identifier is in the proposed membership.
- The message is a PFailBlock message that this process has delivered and the *sender* of the PFailBlock message is in the proposed membership (the *sender* of this message is added to the *blockSet* as well).
- The message is a PFailBlock message that this process has not delivered and the process characterized by the *proc* field of the PFailBlock message is in the proposed membership is added to the *blockSet*.

Figure 5.6: Reasons for restarting consensus.

proposed membership, have agreed on the same *procSet* and *failSet*. Once this condition is met, the process transitions to the *Gather Cut Info* state.

The process, upon entering the *Gather Cut Info* state, determines the local cut information for its current view, and sends it via the control hierarchy. The control hierarchy, aggregates the cut information of all of the processes in that view, and returns a single cut value for that view. This cut value represents the maximum information that can be recovered from this view. Upon receiving this information from the hierarchy, the process sends a RecInfo message, as a representation of this information, to the process group, and transitions to the *Gather Recovery Info* state. If the process, while in this state, receives a message that does not fit the proposed membership (for explanation, see Figure 5.6), it transitions back to the *Consensus* state.

Once in the *Gather Recovery Info* state, the process waits to receive a RecInfo message from every process in the proposed membership. The gathering of the cut information through the hierarchy is not enough to obtain the correct cut information. The gathering of RecInfo messages from all of the processes in the proposed membership, provides a clearer cut for the process group. It allows processes that enter from the same view to rule out some of the inconsistencies between the structure of the process group and the control group. It also allows processes that enter from different views to share cut information and allows the process to determine the time at which the next view should be installed. Upon reception of all of the RecInfo messages, the process transitions to the *Recovery* state. If the process, while in this state, receives a message that does not fit the proposed membership (for explanation see Figure 5.6), it transitions back to the *Consensus* state.

The *Recovery* state is the state in which the process attempts to acquire all of the messages in its current view, that were sent before the cut for that view. Upon entry to this state, the process calculates the cut for its current view, and the time at which the next view is to be installed, from the information in the RecInfo messages. Using this calculated information, the protocol obtains and orders all of the messages necessary to install the next view. When all of these messages are ready to be delivered, the process sends a RTC message (with the sequence number of the first message to be delivered from this process in the next view). After the RTC message is sent, the process transitions to the *Gather Ready To Commit Messages* state if it is the leader of the next view, or to the *Wait for Membership Change Message* state otherwise. The *leader of a view* is the process that has the lowest unique process identifier in the sender group of a view. Thus, each process can determine who is the leader of the next view locally, by the lowest process identifier in the proposed membership. If the process, while in this state receives a message that does not fit the proposed membership (for explanation, see Figure 5.6), the protocol must roll back the messages that are ready to be delivered to the point at which recovery started Once this is done, the process transitions back to the *Consensus* state. If this process is unable to recover all of the messages due to a failure of a process that was the only one holding a copy of a message that needed to be recovered, a PFailBlock message will be sent accusing an already failed processes of failing. This action will cause the process to determine a new cut, and send a RecInfo message based on that cut. Once the RecInfo message is sent, the process waits to roll back the messages that are ready to be delivered to the point at which recovery started, and then transitions back to the *Gather Recovery Info* state.

The leader of the next view waits to receive a RTC messages from every process in the proposed membership while it is in the *Gather Ready To Commit Messages* state. Once it has received all of the RTC messages, it chooses a unique identifier for the next view, constructs a MC message with that identifier, sends it to the process group, and transitions to the *Wait for Membership Change Message* state. If the process, while in this state, receives a message that does not fit the proposed membership (for explanation, see Figure 5.6), it must roll back the messages that are ready to be delivered to the point at which recovery started. Once that task is finished, the process transitions back to the *Consensus* state.

A process in the *Wait for Membership Change Message* waits for a MC message from the leader of the next view. Once it receives this message, the process transitions to the *Deliver and Install* state. If the process, while in this state, suspects that the leader of the next view has failed or receives a message from the leader of the next view that is not consistent with the proposed membership, it must roll back the messages that are ready to be delivered to the point at which recovery started. Once that task is finished, the process transitions back to the *Consensus* state.

In the *Deliver and Install* state, the process delivers all of the messages ordered in the *Recovery* state to the application, followed by a UMC message based on the information in the MC message received from the new leader. The next sequence number that each process in the sender group of the new view delivers is set, according to the information in the MC message, if that value is not already larger. It cancels all retransmission requests for messages with a lower sequence number from that process. Finally, the process installs the new view, and resumes delivery of messages to the user and the sending of PLeave messages, before it exits the MRA.

5.5 Receiver Membership Repair Algorithm

The receiver membership repair algorithm (RMRA) is executed at a process that is not in the sender group of the current view, or that has shown no intention to be in the sender group of the next view. A process executes the RMRA while it is in the *Waiting for Membership Change* state of the process group membership protocol. We have discussed the entry conditions of the *Waiting for Membership Change* state previously. The *Waiting for Membership Change* state may be exited after a successful completion of the RMRA or if the process does not "believe" that the RMRA can be completed. If the RMRA is successful, the process transitions back to the state from which it entered the *Waiting for Membership Change* state. If the RMRA is unsuccessful, the process transitions to the *Bootstrap Receiver* state.

We discuss the RMRA in detail, because it allows the membership protocol to deal with failures of processes, partitioning of the process group, and merging of partitions. It differs from the MRA, because the process does not actively participate in the exchange of membership messages, other than reporting its local cut information to the control group. It can only observe messages and, thus, making the correct decision is even more important. For this reason, we have had to allow the process to leave the RMRA, deeming it unable to terminate.

5.5.1 Data Structures

The RMRA employs the following messages: PFailure, MC and UMC. All of these messages are described in Section 5.2.

5.5.2 The Protocol

The state machine for the RMRA is shown in Figure 5.7. The process enters the RMRA by entering the *Initial* state. This is where the RMRA is initialized. The delivery of messages to the user is halted.

Upon the completion of this task, the process determines the local cut information for its current view and sends it via the control hierarchy, and then transitions to the *Wait for Membership Change Message* state.

A process in the *Wait for Membership Change Message* waits for a MC message that contains the current view identifier in the *oldIDs* field. Once it receives this message, it enters the *Recovery* state.

The *Recovery* state is the state in which the process attempts to acquire all of the messages in its current view, before the cut for that view. Upon

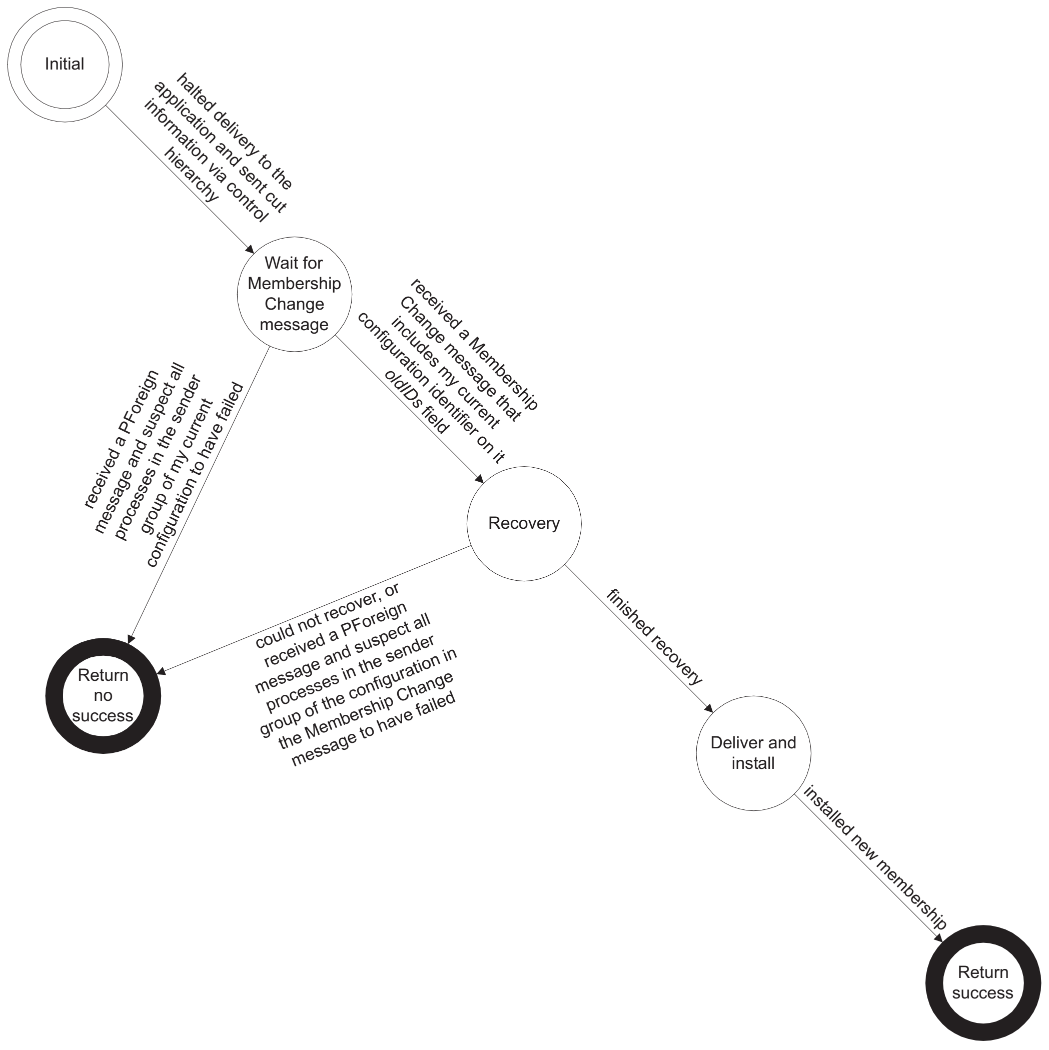
Initial
halted delivery to the application and sent cut information via control hierarchy
Wait for Membership Change message
received a Membership Change message that includes my current configuration identifier on it oldIDs field
received a PForeign message and suspect all processes in the sender group of my current configuration to have failed
Recovery
could not recover, or received a PForeign message and suspect all processes in the sender group of the configuration in the Membership Change message to have failed
finished recovery
Return no success
Deliver and install
installed new membership
Return success

Figure 5.7: The Receiver Membership Repair Algorithm.

entry to this state, the process calculates the cut for its current view, and the time at which the next view is to be installed, from the information in the MC message. If the process has delivered messages following the calculated cuts, it declares itself failed, informs the application of this fact and exits. Otherwise, it obtains and orders all of the messages necessary to install the next view.

When all of these messages are ready to be delivered, the process transitions to the *Deliver and Install* state.

In the *Deliver and Install* state, the protocol delivers all of the messages ordered in the *Recovery* state to the application, followed by a UMC message based on the information in the MC message received from the new leader. The next sequence number of the message to be delivered from each process in the sender group of the new view is set according to the information in the MC message, if that value is not already larger. All retransmission requests for messages with a lower sequence number from that process are canceled. Finally, the new view is installed, delivery of messages to the user is continued, and the sending of PLeave messages is allowed again, and the RMRA is deemed successful.

The RMRA is deemed unsuccessful if (1) all of the processes in the sender group of the current view are suspected of having failed before the MC message is received and a PForeign message is received, (2) all of the processes in the sender group of the view specified in the MC are suspected of having failed after the MC is received and a PForeign message is received, or (3) the process is not able to complete the recovery phase.

5.6 Process and Network Fault Detector

The process and network fault detector is responsible for determining process failures and the reachability of processes, and is based on timeouts. The process and network fault detector analyzes every message received from the process group. If it does not receive a message from a process for a predetermined amount of time it notifies the system of this via a PFailure message. In order for this approach to work, processes must send messages periodically even when they have no data to send. This requirement is satisfied because every process that is sending data also periodically sends Keep Alive messages.

Chapter 6

Implementation

The InterGroup protocols are implemented as three major components: the control hierarchy, the reliable multicast, and the ordering and delivery. Each of these components is implemented as described in the preceding chapters. Each component is built using a set of threaded modules that communicate via an asynchronous event/message model. Because different services require different modules, each component is designed so that the modules and message flows can be dynamically constructed and maintained. They are built around the InterGroupThreadedModule (IGTM) class, described in Section 6.1.

For testing purposes a statistics gathering infrastructure was added to the protocols. This allows unobtrusive collection of performance data from within the protocol stack, and has simplified the task of data collection in our wide-area test bed. The statistics gathering infrastructure and the wide-area test bed design are presented in Section 6.2.

6.1 The InterGroupThreadedModule

The InterGroup protocols implementation uses an asynchronous event/message model for communication between modules within a component. The base class for all modules in the InterGroup protocols implementation is the InterGroupThreadedModule (IGTM).

The IGTM implements two interfaces: Runnable and MessagePublisher (MP). It also contains a PriorityQueue class that implements the MessageSubscriber (MS) interface.

The Runnable interface is defined in the Java(TM) 2 Platform Standard Edition API[31]. It allows the IGTM to run as a thread.

The MP interface allows for the publishing of events/messages to classes that implement the MS interface.[1] Subscriptions are made via an **attach** method. The MP interface also provides a mechanism for subscribers to cancel their subscriptions using a **detach** method. The IGTM implementation of the MP interface uses a data structure that maps message types to message subscribers. Whenever a message is published via the **notify** method, the **update** method of each of the subscribers to the message type is called. This implementation allows subscribers to subscribe to and receive events/messages of different types and allows multiple subscribers to receive the same types of events/messages.

The PriorityQueue class implements the MS interface. It receives messages via the **update** method and buffers them until the IGTM is ready to process them. When the IGTM is ready to remove a message from the PriorityQueue, the message returned to the IGTM is chosen using a randomized priority queue algorithm based on the priority of the messages in the queue. The implementation of the PriorityQueue makes no guarantee that messages of different priorities will leave the queue in the order in which they arrived. However, for messages of the same priority, the PriorityQueue acts as a FIFO queue.

The properties of the IGTM class allow for easy setup of dynamic, configurable, asynchronous data flows within components.

6.2 The Statistics Gathering Infrastructure

In designing the statistics gathering infrastructure we had the following goals in mind: (1) to have control over the statistics reporting at the API level, (2) to allow application-specific statistics reports, and (3) to minimize the cost of the infrastructure when statistics reporting is turned off. We meet these goals by (1) using a singleton system-wide class that performs the actual reporting to the statistics server, (2) defining a StatisticsInterface that gives the programmer control over when the actual reporting to the statistics server is done, and (3) choosing a statistics message format that allows applications to define application-specific messages in an easy manner.

[1] An event/message may be an object of any class that implements the Message interface.

6.2.1 Client-side Implementation

The StatisticsInterface is implemented by the InterGroupObject (IGO) class in the InterGroup protocols. The IGO class uses a FIFO buffer to hold objects pending delivery to the statistics server. We use a buffer to hold statistical information to reduce the cost of statistics reporting. By using the *incStat(Object o)* method, a subclass of IGO can log statistics even in time-critical sections of the code. Thus, the actual reporting to the statistics server via the *sendStat()* method can be delayed, without major modifications to the code of the class. The *sendStat()* method calls the singleton system-wide class StatisticsModule that does the actual reporting to the statistics server. By using the StatisticsModule class in this way we prevent interleaving of concurrent reports at the statistics server.

6.2.2 Server-side Implementation

The StatisticsServer accepts connections from the reporting processes and launches a StatListen thread for each connection. Each StatListen thread, logs all the messages it receives in a file for later processing. The filename has the following format:

name of client: W.X.Y.Z

# of connections accepted by server before this one: n

filename = Z/Y/X/W/proto$n$.igs

Optionally, a real-time statistics processor and viewer may be added to the statistics server.

6.2.3 Statistics Messages

The format of statistics messages is relatively free-form, which allows an application to easily send statistics reports. A statistics message is defined as a list of Objects. The following are the restrictions on statistics messages:

- All objects in a statistics message must inherit the Serializable interface [31].

- The first object in a statistics message must be a String that represents the type of the message. The following types are reserved by the protocols, and should not be used by applications: "P_INIT," "COORD

MEMB," "LOCAL MEMB," "PLEAVE," "MRA," "NO MRA," "MC," "TO SENDER," "SPONSOR," "PADD," "RTR," "Sent RTR," "RTX."

6.2.4 The WAN Testbed

We have developed an application for testing the InterGroup protocols over a wide-area network. It uses the statistics gathering infrastructure to report on progress and other relevant statistics. This allows the application to run unsupervised, allowing us to distribute it to interested parties, without having to obtain accounts on their systems, and thus allows us to run more comprehensive tests of the protocols.

The application is split into two components: UDP reporting and InterGroup reporting. The UDP reporting uses IP multicast directly for the inter-application communication. It provides a baseline for the reachability and performance measurements in the wide-are environment. The UDP reporting component is based on the Beacon component of the Multicast Beacon [46] at the National Laboratory for Applied Network Research (NLANR). The processes communicate via multicast messages. Messages are periodically sent to the multicast group. Based on the received messages, each process creates a report that is sent out periodically to the InterGroup statistics server using the InterGroup statistics gathering infrastructure.. The report provides information on reachability, latency, jitter, losses, and out-of-order messages between this process and the other group members. The InterGroup component uses the InterGroup protocols for inter-application communication. Otherwise, the InterGroup component behaves in the same manner as the UDP component.

Chapter 7

Performance

To validate the behavior and to obtain performance numbers, we ran the Inter-Group Protocols in a local-area environment and in a wide-area environment. The local-area experiments provide us with information on the behavior of the protocols in a controlled setting. For these experiments, we ran the protocol on a computer cluster of the Computer Science Department at the University of California, Santa Barbara and on a LAN in the Computer Networks and Distributed Systems Laboratory of the Electrical and Computer Engineering Department at the University of California, Santa Barbara.

The cluster consists of 36 dual Pentium II 400MHz CPU nodes and 6 quad Pentium III 500MHz CPU nodes. Twenty-four of the dual nodes have 512MB of RAM, while the rest of the nodes have 1GB of RAM. Each node contains two 100 Mb/s Ethernet cards and all the nodes are connected through a Gigabit Switch. The nodes on the cluster run Linux version 2.2.15 (RedHat distribution) and Java(TM) VM version 1.2.2 (Classic VM build 1.2.2-L, green threads, javacomp).

In the Computer Networks and Distributed Systems Laboratory, 4 Pentium II xxxMHz nodes were used for the experiments. Each of the nodes has 128MB of RAM. Each node contains a 100 Mb/s Ethernet card and the nodes are connected via a 100 Mb Ethernet. The nodes run Linux version 2.2.x (Mandrake distribution) and Java(TM) version 1.3.0beta_refresh with the Java HotSpot(TM) Server VM (build 1.3.0beta-b07).

The wide-area experiments provide us with information on the behavior of the protocols in a more hostile environment. For these experiments, we ran the protocol on computers located at the Lawrence Berkeley National Laboratory

Number of processes	Message size (bytes)	Data Throughput (msgs/sec)	Data Throughput (bytes/sec)
1	1	96	96
2	1	154	154
4	1	248	248
1	1000	94	94000
2	1000	140	140000
4	1000	140	140000
1	10000	70	700000
2	10000	30	300000
4	10000	18	180000

Table 7.1: Data throughput on the CS cluster.

(LBNL) and Argonne National Laboratory (ANL). The computers used varied in processor architecture (i386, SPARC), operating system (Solaris 2.6-2.7, Linux 2.2.x) and virtual machine (classic Sun 1.2.x VM, HotSpot(TM) Client 1.3.x, HotSpot(TM) Server 1.3.x).

7.1 Local-area Environment

We investigated the latency and throughput of the InterGroup protocols in the local-area environment. We also observed the protocol overhead during these experiments. In all of the experiments, the processes requested the group timestamp ordered delivery service.

7.1.1 Throughput Measurements

To measure the throughput of the InterGroup protocols, we set up a system consisting of one process group. We varied the number of processes and the payload size of the messages sent. We sent messages as fast as the InterGroup protocols allowed. We used the dual Pentium II nodes of the cluster for our experiments. Table 7.1 shows the throughput we obtained in our experiments on the cluster.

The maximum message send rate at a process is 100 messages/sec. This is due to the interaction of our flow control algorithm and the Thread.sleep method implementation in the Java(TM) Virtual Machine (Java(TM) VM).

The flow control algorithm determines how long the protocols should wait before sending the next message. The waiting is accomplished by calling the Thread.sleep method. We have found that this method, on average adds 10ms to any call to it (*i.e.,* if an argument of 15ms is passed to the Thread.sleep method, it returns in 25ms, on average).[1] Thus, with this combination, the best send rate at a process, is 100 messages/sec.

The performance of the system suffers tremendously when the system attempts to send messages faster than 200kb/sec. A this point, the receivers begin losing large blocks of messages. This is due to the garbage collection mechanisms in the Java(TM) VM on these systems.

We ran throughput tests with the garbage collection mechanisms turned off. These tests showed that the system can process up to 900 messages/sec if there are no losses in the network. However, with garbage collection turned off, the system can run for only a very limited time.

The HotSpot(TM) 1.3.x Java(TM) VM from Sun offers an option for incremental garbage collection. We had problems running this implementation on the cluster because of a bug in the glibc version installed on the cluster.

Thus, we ran the same throughput tests, using the incremental garbage collection option, on machines in the Computer Networks and Distributed Systems Laboratory. Table 7.2 shows the throughput results we obtained. The maximum send rate at a process is now 50 messages/sec. This is due to the fact that, in this VM, on average 20ms is added to any call to the Thread.sleep method. [2]

With incremental garbage collection, we see that the throughput stays stable within the observed cases.

7.1.2 Latency Measurements

We measured the delivery latencies for a system consisting of one process group and varying the number of processes in the group. The experiments were performed on the dual Pentium II cluster nodes. We also varied the sending behavior of the application. In one case, we had the application send messages as fast as the InterGroup protocols would allow. In another case, the application sent messages every 100 milliseconds.

[1]The same behavior is observed for the Object.wait method.

[2]Again, we observed the same behavior with the Object.wait method. We also observed that the IBM 1.3 VM implementation shows the same behavior.

Number of processes	Message size (bytes)	Data Throughput (msgs/sec)	Data Throughput (bytes/sec)
1	1	49	49
2	1	98	98
4	1	197	197
1	1000	48	48000
2	1000	97	97000
4	1000	195	195000
1	10000	48	480000
2	10000	96	960000
4	10000	185	1850000

Table 7.2: Data Throughput on the Laboratory LAN.

Number of processes	Minimum Latency (ms)		Maximum Latency (ms)		Mean Latency (ms)		Median Latency (ms)	
	App	IG	App	IG	App	IG	App	IG
2	5	1	752	355	25	10	20	2
4	8	3	781	613	58	45	21	11

Table 7.3: Delivery latency at full throughput.

The delivery latencies presented are the delivery latencies to the group *i.e.,* the delivery latency was computed as the difference between the sending time of a message and the last time of delivery of that message. The application latencies are end-to-end latencies at the application, measured from the time the message is sent to the InterGroup protocols until the time the message is received from the protocols. The protocol latencies are measured from the time the message is sent to the outgoing socket until the time it is ready to be delivered to the application.

Latency at Full Throughput

In this latency experiment each process sent data messages as fast as the InterGroup protocols allowed. Table 7.3 shows the latency results.

No messages were lost in any of these runs.

A long tail in the latency distribution exists in the results that influences the mean and median values. The tail is due to the garbage collector. When the garbage collector runs, the protocol threads make very slow progress and the messages get delayed by the duration of the garbage collection process.

Figure 7.1 shows the latency frequencies for the two node experiment. In the figure we observe dominant peaks in the latencies. These are the delivery latencies that are not affected by the garbage collector. It can also be observed that the application latencies are shifted by 10ms (excluding the effects of the garbage collector). This is due to the flow control effect noted earlier.

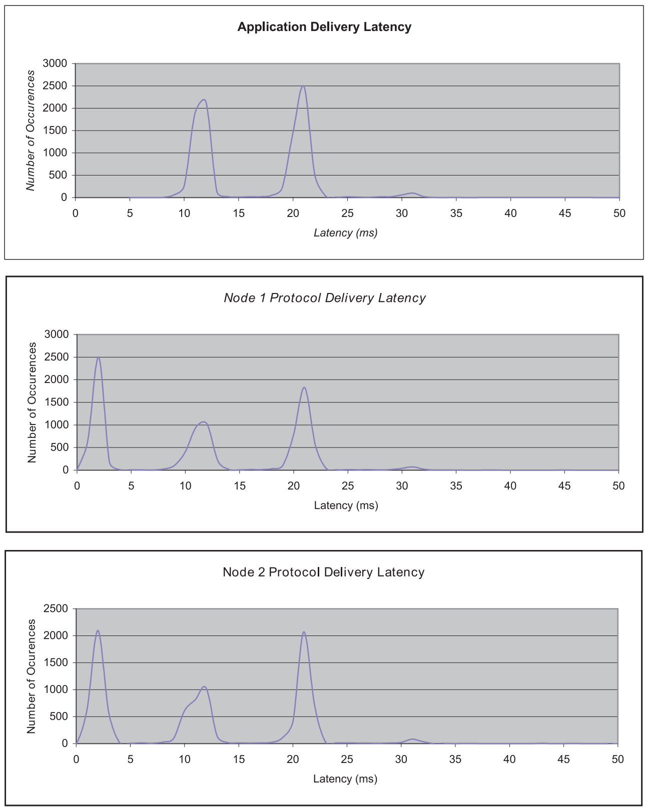

Figure 7.1: The delivery latency distribution at full throughput for four nodes.

Maximum Latency

In this latency experiment, each process sends a message every 100 milliseconds. Keep Alive messages were sent with a uniform distribution every 25-75 ms. Thus, these measurements reflect the latency to delivery of the protocols

Number of processes	Minimum Latency (ms)		Maximum Latency (ms)		Mean Latency (ms)		Median Latency (ms)	
	App	IG	App	IG	App	IG	App	IG
1	1	0	712	710	1	0	2	1
2	2	1	1253	982	44	37	37	27
4	11	3	1242	1187	80	57	57	44
8	18	9	3444	1977	760	747	741	733
16	41	38	4391	3630	1325	1315	1355	1346

Table 7.4: Delivery latency at 10 msgs/sec send rate at each sender.

when the Keep Alive timeout is the main factor in the delivery latency. The theoretical maximum delivery latency for timestamp group ordered delivery in the absence of errors is

Equation 7.1 $latency_{max} = distance_{max} + KEEP\_ALIVE\_TIMEOUT$

The maximum latency of the protocol in these experiments is thus expected to be 75 ms. Table 7.4 shows the results obtained from this experiment.

The results are again affected by the garbage collector, so we must look at the latency frequencies to obtain meaningful results. Figure 7.2 shows the latency frequencies for the application delivery latency in the four-node experiment. We observe dominant peaks in the 40-80 ms range. We also observe the effects of the garbage collector in the tail.

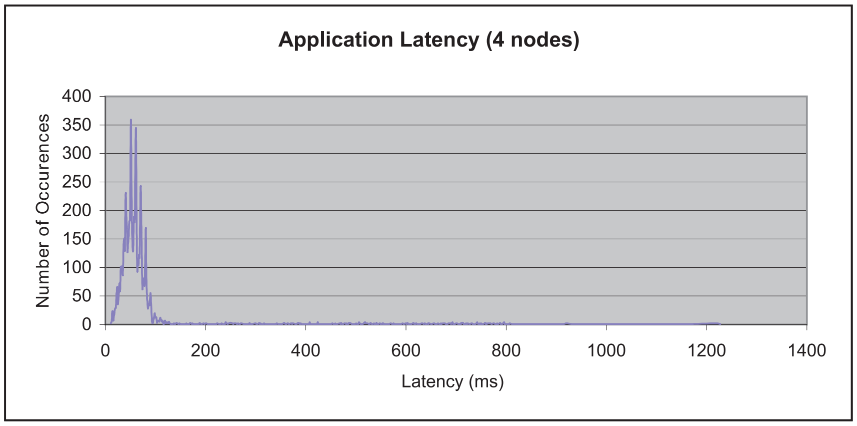

Figure 7.2: The application delivery latency distribution for four nodes.

Figure 7.3 shows the latency frequencies for the application delivery latency in the 16 node experiment. Here we can see that the garbage collector effects have become the dominant feature, although a peak around 70ms still exists. This shows that in the absence of the garbage collector the latencies would be as predicted.

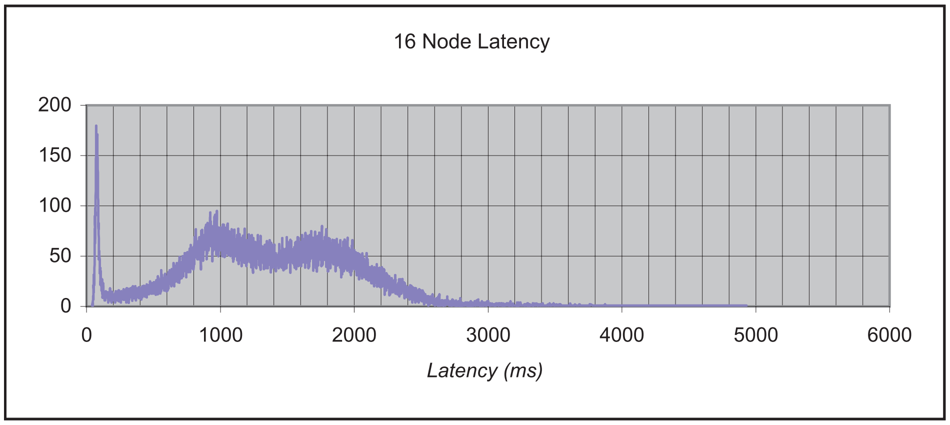

Figure 7.3: The application delivery latency distribution for sixteen nodes.

7.1.3 Protocol Overhead

During the local-area measurements, we also measured the overhead of the additional protocol messages and headers. The experimental results corroborate the theoretical expectations.

The flow control used for control messages is RTCP-based with a minimum time between messages of 1 second. Thus, if the bandwidth consumed by the control messages is less than the allowed maximum control bandwidth of 390 kB/sec, each process should, on average, send one control message every second. The control bandwidth in our experiments was much lower than 390 kB/sec, and the average number of control messages received per second was equal to the number of processes participating in the experiment.

Overhead messages, other than the control messages, were mainly due to retransmission requests and repairs. This overhead depends on a variety of factors and cannot be easily predicted. On average, we observed only slightly more than one request and repair message per message loss.

Application message size (bytes)	Percent overhead
1	98.54%
1000	6.55%
10000	0.71%

Table 7.5: Protocol overhead.

The only overhead messages that occur in the error-free case are the Keep Alive messages. These messages are sent only when there are no application messages to be sent or when the flow control algorithms block the sending of application messages. Thus, Keep Alive messages were mainly observed in the latter latency experiment.

Thus, in terms of message count, the protocol overhead becomes a significant factor only when the application is sending data at a rate less than the inverse of the average Keep Alive timeout. This was the case in the latter latency experiment. In that experiment, slightly more than half of the messages produced by the protocol were non-application data messages.

The protocol header for application messages comprises 72 bytes, 32 of which are due to IP and Java(TM) headers. Thus, in terms of byte counts the protocol overhead becomes a significant factor when the messages being sent by the application are small. In Table 7.5 we show the observed percent overhead, in terms of bytes, for the message sizes used in our experiments.

7.2 Wide-area Environment

We investigated the application latency to delivery of the InterGroup protocols in the wide-area environment for the unreliable unordered delivery service and the reliable group timestamp ordered delivery service. The WAN testbed described in Section 6.2.4 was used to gather the data. We present here the results obtained from an experiment conducted from 9pm on September 20, 2000 through 9am September 21, 2000. Each node sent an application message every second and Keep Alive messages were sent every 50ms. Reports on the delivery latencies were gathered by the nodes and sent to a centralized server every 30 seconds. The experiment was run on 6 nodes, 3 at LBNL and 3 at ANL.

Figure 7.4 shows the latency to delivery of messages, sent by a node at

ANL and delivered at a node at LBNL. The unreliable unordered delivery curve shows only the maximum latencies to this node for each report period.

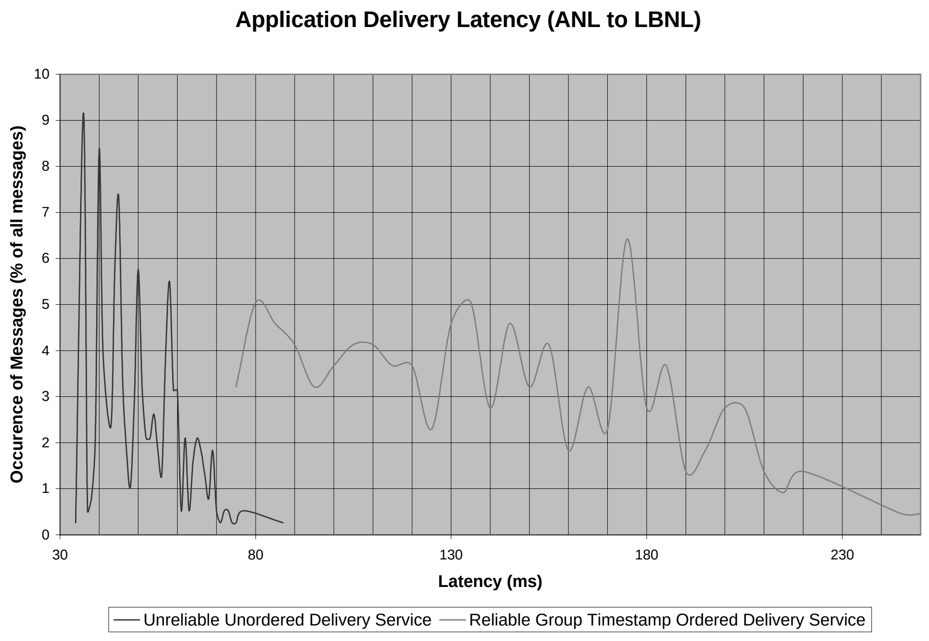

Figure 7.4: The application delivery latency between ANL and LBNL.

The theoretical latency to delivery for timestamp group ordered delivery, of message from other nodes, in the absence of errors is

Equation 7.2 $dLat_{rgto} \leq dLat_{uu} + KEEP\_ALIVE\_TIMEOUT$

where $dLat_{rgto}$ is the latency to delivery for the reliable group timestamp ordered service and $dLat_{uu}$ is the latency to delivery for the unreliable unordered service. This latency is obtained from Equation 7.1 and the fact that the delivery latency for the unreliable unordered service will always be greater than or equal to the distance (in terms of time) between any two nodes.

Thus, by Equation 7.2 and the results presented in Figure 7.4 the latency to delivery for the reliable group timestamped ordered service should not be greater than 140ms in the absence of errors.

A message loss delays the delivery of all subsequent messages until the lost message is retrieved. The link between ANL and LBNL experienced message losses, and thus the theoretical latency to delivery must account for the time to recover a lost message. In this experiment we have determined that the time to recover a missing message was in the range 40ms-180ms. Thus the maximum theoretical latency becomes 250ms.

We have shown that the latency to delivery for the reliable group timestamp ordered service in the wide-area matches our theoretical assumptions. The garbage collection effects we observed in the local-area experiments did not appear in our wide-area experiments since these experiments were not as network (bandwidth) intensive.

Chapter 8

Conclusion and Future Work

To achieve scalability in a group communication system, we have had to change some of the basic ideas of group communication systems and have had to devise novel mechanisms for such systems. Traditional group communication systems have strict safety properties and, to achieve our goals, we have had to modify some of these properties.

To this end we created an asymmetry in the role of the processes in the system based on their sending activity. This asymmetry has allowed us to decrease the number of processes participating in most consensus decisions. Thus, we have focused on the membership of the sender group and, consequently, we have slightly weaker delivery guarantees.

The InterGroup protocols employ a hierarchical structure for gathering and distributing control information. This structure allows acknowledgments and consensus information to be sent to the processes without causing message implosion. Using this hierarchy for acknowledgments adds an additional delay for messages to reach stability. This delay affects the clearing of message buffers and flow control mechanisms, if message stability is required.

The InterGroup system introduces a receiver-oriented choice of delivery services to group communication systems. This choice of delivery services allows individual processes to leverage their own ordering, reliability and real-time constraints independently of the other processes in the system. It also sets the InterGroup protocols apart from traditional group communication systems. The system thus becomes a hybrid of traditional group communication systems and asymmetric systems, where a small number of processes transmit data to a large number of receivers.

We have implemented a prototype of the InterGroup protocols in Java(TM) and have tested the system performance in both local-area and wide-area networks. The performance of the system under a constant load was hampered by the garbage collection mechanisms of the Java(TM) Virtual Machines. Otherwise, the behavior of our prototype implementation matched our expectations.

In the future, we will attempt to improve the performance of the protocols by fine-tuning the implementation. This will include better memory management inside the InterGroup system, so that the effects of the garbage collector are minimized. Also, we will consider rewriting the code in C++.

Future work also includes adding other delivery services to the existing services. The services of interest include unreliable source ordering, unreliable timestamp ordering, and an ALF-enabled service. We also plan to build a protocol layer on top of the existing InterGroup protocols that allows processes to perform ordering across process groups.

Other future work includes adding a proxy service, so that a process may send messages to the group without having to join the sender group. We are also interested in adding a service for approximating the receiver membership. We have implemented a prototype of this service using RTCP.

Bibliography

[1] H. Abdel-Wahab, K. Maly, A. Youssef, E. Stoica, C. M. Overstreet, C. Wild, and A. Gupta. The software architecture and interprocess communications of IRI: an internet-based interactive distance learning system. In *Proceedings of IEEE Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises (WETICE'96)*, Stanford, CA, June 1996.

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

[3] E. S. Al-Shaer, H. Abdel-Wahab, and K. Maly. HiFi: A new monitoring architecture for distributed system management. In *Proceedings of the 19th IEEE International Conference on Distributed Computing Systems*, pages 171–178, Austin, TX, June 1999.

[4] Y. Amir, D. Breitgand, G. Chockler, and D. Dolev. Group communication as an infrastructure for distributed system management. In *Proceedings of the 3rd International Workshop on Services in Distributed and Networked Environments (SDNE)*, pages 84–91, Macau, China, June 1996.

[5] Y. Amir, C. Danilov, and J. 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*, New York, NY, June 2000.

[6] Y. Amir, D. Dolev, S. Kramer, and D. Malki. Membership algorithms for multicast communication groups. In *Proceedings of Distributed Algorithms. 6th International Workshop, WDAG '92*, pages 292–312, Berlin, Germany, November 1992.

[7] Y. Amir, D. Dolev, S. Kramer, and D. Malki. Transis: A communication subsystem for high availability. In *Proceedings of the 22nd IEEE International Symposium on Fault-Tolerant Computing*, pages 76–84, New York, NY, July 1992.

[8] O. Babaoglu, A. Bartoli, and G. Dini. Enriched view synchrony: A programming paradigm for partitionable asynchronous distributed systems. *IEEE Transactions on Computers*, 46(6):642–658, June 1997.

[9] O. Babaoglu, R. Davoli, and A. Montresor. Partitionable group membership: specifications and algorithms. Technical Report TR UBLCS97-1, Department of Computer Science, University of Bologna, January 1997.

[10] K. Berket, R. Koch, L. E. Moser, and P. M. Melliar-Smith. Timestamp acknowledgments for determining message stability. In *Proceedings of the 2nd International Conference on Parallel and Distributed Computing and Networks*, Brisbane, Australia, December 1998.

[11] K. Birman, R. Friedman, M. Hayden, and I. Rhee. Middleware support for distributed multimedia and collaborative computing. In *Proceedings of the SPIE/ACM Conference on Multimedia Computing and Networking (MMCN'98)*, San Jose, CA, January 1998.

[12] K. Birman and T. Joseph. Exploiting virtual synchrony in distributed systems. In *Proceedings of the Eleventh ACM Symposium on Operating Systems Principles*, pages 123–138, Austin, TX, November 1987.

[13] K. P. Birman and R. Van Renesse, editors. *Reliable Distributed Computing with the Isis Toolkit.* IEEE Computer Society Press, 1994.

[14] C. Bormann, J. Ott, H.-C. Gehrcke, T. Kerschat, and N. Seifert. MTP-2: Towards achieving the S.E.R.O. properties for multicast transport. In *International Conference on Computer Communications and Networks (ICCCN 94)*, San Francisco, CA, USA, September 1994.

[15] V. G. Cerf and R. E. Kahn. A protocol for packet network intercommunication. *IEEE Transactions on Communications*, 22(5):647–648, May 1974.

[16] S. Chodrow, M. Hircsh, I Rhee, and S. Y. Cheung. Design and implementation of a multicast audio conferencing tool for a collaborative computing framework. In *Proceedings of 3rd Joint Conference on Information Sciences (JCIS '97)*, Research Triangle Park, NC, March 1997.

[17] D. Clark and D. Tennenhouse. Architectural considerations for a new generation of protocols. In *Computer Communications Review, vol. 20, no. 4, ACM SIGCOMM '90 Symposium. Communications Architectures and Protocols*, pages 201–208, Philadelphia, PA, USA, September 1990.

[18] J. Cooperstock and S. Kotsopoulos. Why use a fishing line when you have a net? an adaptive multicast data distribution protocol. In *Proceedings of USENIX 1996 Annual Technical Conference*, pages 343–352, San Diego, CA, USA, January 1996.

[19] S. Deering. Host extensions for IP multicasting. Network Working Group Request for Comments Internet RFC-1112, 1989.

[20] Moser L. E., P. M. Melliar-Smith, and P. Narasimhan. Consistent object replication in the eternal system. *Theory and Practice of Object Systems*, 4(2):81–92, 1998.

[21] P. Ezhilchelvan, R. Macedo, and S. Shrivastava. Newtop: A fault-tolerant group communication protocol. In *Proceedings of the 15th IEEE International Conference on Distributed Computing Systems*, pages 296–306, Vancouver, Canada, May 1995.

[22] A. Fekete, N. Lynch, and A. Shvartsman. Specifying and using a partitionable group communication service. In *Proceedings of the 16th ACM Symposium on Principles of Distributed Computing (PODC)*, pages 53–62, Santa Barbara, CA, August 1997.

[23] S. Floyd, V. Jacobson, C.-G. Liu, S. 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.

[24] R. Friedman and R. van Renesse. Strong and weak virtual synchrony in Horus. Technical Report TR 95-1537, Department of Computer Science, Cornell University, August 1995.

[25] R. Friedman and A. Vaysburg. Fast replicated state machines over partitionable networks. In *Proceedings of the IEEE 16th International Symposium on Reliable Distributed Systems*, Durham, NC, October 1997.

[26] D. Gang, G. Chockler, T. Anker, A. Kremer, and T. Winkler. Conducting midi sessions over the network using the transis group communication system. In *Proceedings of the International Computer Music Conference (ICMC 97)*, Thessaloniki, Greece, September 1997.

[27] J. Hickey, N Lynch, and R van Renesse. Specifications and proofs for ensemble layers. In *Proceedings of the Fifth International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS '99)*, Lecture Notes in Computer Science, Amsterdam, Netherlands, March 1999. Springer-Verlag.

[28] M. Hofmann. Adding scalability to transport level multicast. In *Proceedings of Third International COST 237 Workshop*, pages 41–55, Barcelona, Spain, November 1996.

[29] H. W. Holbrook, S. K. Singhal, and D. R. Cheriton. Log-based receiver-reliable multicast for distributed interactive simulation. *Computer Communication Review*, 21(4):328–341, October 1995.

[30] IETF reliable multicast transport (rmt) working group home page. http://www.ietf.org/html.charters/rmt-charter.html.

[31] Java(tm) 2 platform, standard edition, v1.2.2 API specification. http://java.sun.com/products/jdk/1.2/docs/api/index.html.

[32] I. Keidar and D. Dolev. Effiecient message ordering in dynamic networks. In *Proceedings of the 15th ACM Symposium on Principles of Distributed Computing (PODC)*, pages 68–76, Philadelphia, PA, May 1996.

[33] B. Kemme and Alonso G. A suite of database replication protocols based on group communication primitives. In *Proceedings of the 18th IEEE International Conference on Distributed Computing Systems*, pages 156–163, Amsterdam, Netherlands, May 1998.

[34] R. Khazan, A. Fekete, and N. Lynch. Multicast group communication as a base for a load-balancing replicated data service. In *Proceedings of the*

12th International Symposium on Distributed Comupting (DISC), pages 258–272, Andros, Greece, September 1998.

[35] R. R. Koch. *The Atomic Group protocols: reliable ordered message delivery for ATM networks.* PhD thesis, University of California, Santa Barbara, August 2000.

[36] A. Krantz, S. Chodrow, and M. Hircsh. Design and implementation of a distributed x multiplexor. In *Proceedings of the 18th IEEE International Conference on Distributed Computing Systems*, Amsterdam, Netherlands, May 1998.

[37] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–568, July 1978.

[38] C. Malloth and A. Schiper. View synchronous communication in large scale networks. Technical Report BROADCAST TR No. 92, Départment d'Informatique, Ecole Polytechnique Fédérale de Lausanne, 1995.

[39] P. M. Melliar-Smith, L. E. Moser, and V. Agrawala. Broadcast protocols for distributed systems. *IEEE Transactions on Parallel and Distributed Systems*, 1(1):17–25, January 1990.

[40] D. L. Mills. Network time protocol (version 3) specification, implementation and analysis. IETF Request for Comments: 1305, March 1992.

[41] S. Mishra and G. Pang. Design and implementation of an availability management service. In *Proceedings of the 19th IEEE International Conference on Distributed Computing Systems Workshop on Middleware*, pages 128 – 133, Austin, TX, June 1999.

[42] L. E. Moser, Y. Amir, P. M. Melliar-Smith, and D. A. Agarwal. Extended virtual synchrony. In *Proceedings of the 14th IEEE International Conference on Distributed Computing Systems*, pages 56–65, Poznan, Poland, June 1994.

[43] L. E. Moser, P. M. Melliar-Smith, and V. Agrawala. Processor membership in asynchronous distributed systems. *IEEE Transactions on Parallel and Distributed Systems*, 5(5):459–473, May 1994.

[44] L. E. Moser, P. M. Melliar-Smith, R. K. Budhia D. A. Agarwal, and C. A. Lingley-Papadopoulos. Totem: A fault-tolerant multicast group communication system. *Communications of the ACM*, 39(4):54–63, April 1996.

[45] P. Narasimhan, Moser L. E., and P. M. Melliar-Smith. Replica consistency of corba objects in partitionable distributed systems. *Distributed Systems Engineering*, 4(3):139–150, September 1997.

[46] NLANR - multicast beacon home page. http://dast.nlanr.net/Projects/Beacon/.

[47] J. Nonnenmacher, E. W. Biersack, and Towsley D. Parity-based loss recovery for reliable multicast. *IEEE/ACM Transactions on Networking*, 6(4):349–361, August 1998.

[48] S. Paul, K. K. Sabnani, J. C. Lin, and S. Bhattacharyya. Reliable multicast transport protocol (RMTP). *IEEE Journal on Selected Areas in Communications*, 15(3):407–421, April 1997.

[49] S. Raman and S. McCanne. Scalable data naming for application level framing in reliable multicast. In *Proceedings of ACM Multimedia '98*, Bristol, United Kingdom, September 1998.

[50] I. Rhee, S. Cheung, P. Hutto, and Sunderam V. Group communication support for distributed multimedia and CSCW systems. In *Proceedings of the 17th IEEE International Conference on Distributed Computing Systems*, Baltimore, MD, May 1997.

[51] D. Rubenstein, J. Kurose, and D. Towsley. Real-time reliable multicast using proactive forward error correction. In *Proceedings of NOSSDAV 1998*, Cambridge, England, July 1998.

[52] A. Schiper and M. Raynal. From group communication to transactions in distributed systems. *Communications of the ACM*, 39(4):84–87, April 1996.

[53] H. Schulzrinne, S. Casner, R. Frederick, and V. Jacobson. RTP: A transport protocol for real-time applications. IETF Request for Comments: 1889, January 1996.

[54] P. Sharma, D. Estrin, S. Floyd, and L. Zhang. Scalable session messages in srm using self-configuration. Technical Report 98-670, USC, February 1998.

[55] T. Shiroshita, T. Sano, O. Takahashi, M. Yamashita, N. Yamanouchi, and T. Kushida. Performance evaluation of reliable multicast transport protocol for large-scale delivery. In *Proceedings of Fifth International Workshop on Protocols for High-Speed Networks*, pages 149–164, Sophia Antipolis, France, October 1996. Chapman and Hall.

[56] W.T. Strayer, B.J. Dempsey, and A.C. Weaver. *XTP: The Xpress Transfer Protocol.* Addison-Wesley, 1992.

[57] R. Talpade and M. H. Ammar. Single connection emulation (SCE): An architecture for providing a reliable multicast transport service. In *Proceedings of the IEEE International Conference on Distributed Computing Systems*, pages 144–151, Vancouver, BC, Canada, May 1995. IEEE Computer Society Press.

[58] R. van Renesse, K. Birman, M. Hayden, A. Vaysburd, and D. Karr. Building adaptive systems using ensemble. Technical Report TR97-1638, Department of Computer Science, Cornell University, July 1997.

[59] R. van Renesse, K. P. Birman, and S. Maffeis. Horus: A flexible group communication system. *Communications of the ACM*, 39(4):76–83, April 1996.

[60] L. Vicisano and J. Crowcroft. One to any reliable bulk-data transfer on the mbone. In *Proceedings of the Third International Workshop on High Performence Protocol Architectures HIPPARCH '97*, Uppsala, Sweden, June 1997.

[61] R. Vitenberg, I. Keidar, G. V. Chockler, and D. Dolev. Group communication specifications: A comprehensive study. Technical Report CS99-31, Computer Science Institute, The Hebrew University of Jerusalem, September 1999. Also MIT Technical Report MIT-LCS-TR-790.

[62] B. Whetten, T. Montgomery, and S. Kaplan. A high performance totally ordered protocol. In *Proceedings of the International Workshop on Theory and Practice in Distributed Systems*, pages 33–57, Dagstuhl Castle, Germany, September 1994. Springer-Verlag.

[63] R. Yavatkar, J. Griffioen, and M. Sudan. A reliable dissemination protocol for interactive collaborative applications. In *Proceedings of ACM Multimedia*, pages 333–44, November 1995.