Fall 1995

Atomic Broadcast in Heterogeneous Distributed Systems

Osman ZeinElDine
Old Dominion University

ATOMIC BROADCAST IN HETEROGENEOUS DISTRIBUTED SYSTEMS

by

Osman ZeinElDine

B.S. June 86, Alexandria University, Alexandria, Egypt
M.S. July 89, Alexandria University, Alexandria, Egypt

A Dissertation Submitted to the Faculty of Old Dominion University in Partial Fulfillment of the Requirements for the Degree of

Doctor of Philosophy
Computer Science
Old Dominion University

Approved by:

Dr. Hussein Abdel–Wahab, Advisor

Dr. Ravi Mukkamala

Dr. Stewart Shen

Dr. Shunichi Toida

Dr. Hani Elsayed–Ali

To My Parents

Abstract

Communication services have long been recognized as possessing a dominant effect on both performance and robustness of distributed systems. Distributed applications rely on a multitude of protocols for the support of these services. Of crucial importance are multicast protocols. Reliable multicast protocols enhance the efficiency and robustness of distributed systems. Numerous *reliable multicast* protocols have been proposed, each differing in the set of assumptions adopted, especially for the communication network. These assumptions make each protocol suitable for a specific environment. The presence of different distributed applications that run on different LANs and single distributed applications that span different LANs mandate interaction between protocols on these LANs. This interaction is driven by the necessity of cooperation between individual applications. The state of the art in reliable multicast protocols renders itself inadequate for multicasting in interconnected LANs. The progress in development methodology for efficient and robust LAN software has not been matched by similar advances for WANs. A high-latency, a lower bandwidth, a higher probability of partitions, and a frequent loss of messages are the main restrictive barriers. In our work, we propose a global standard protocol that orchestrates cooperation between the different reliable broadcast protocols that run on different LANs. Our objective is to support a reliable ordered delivery service for inter-LAN messages and achieve the utmost utilization of the underlying local communication services. Our protocol suite accommodates the existence of LANs managed by autonomous authorities. To uphold this autonomy (as a defacto condition), LANs under different authorities must be able to adopt different ordering criteria for group multicasting. The developed suite assumes an environment in which multicasting groups can have members that belong to different LANs; each group can adopt either total or causal order for message delivery to its members.

We also recognize the need for interaction between different reliable multicasting protocols. This interaction is a necessity in an autonomous environment in which each local authority selects a protocol that is suitable to its individual needs. Our protocols are capable of interacting with any reliable protocol that achieves a causal order as well as with all timestamp-based total-order protocols. Our protocols can also be used as a medium for interaction between existing reliable multicasting protocols. This feature opens new avenues in interactability between reliable multicasting protocols. Finally, our protocol suite enjoys a communication structure that can be aligned with the actual routing topology, which largely minimizes the necessary protocol messages.

Acknowledgments

I am extremely fortunate to have Dr. Hussein Abdel-Wahab as my advisor. His guidance and knowledge made the completion of this work possible; I have always been able to count on his encouragement and support whenever I need them. His kindness and understanding have made my life easier. His patience in guiding my educational progress has been remarkable, I have truly enjoyed our relationship and fruitful discussions.

I would like to thank my dissertation committee members, Drs. Ravi Mukkamala, Stewart Shen, and Shunichi Toida, for their individual guidance and support. Dr. Toida helped me to learn clustering. His support, advise, valuable comments, and encouragement was of great help. I still remember my first database class at ODU with Dr. Shen, and I will never forget the cooperative environment provided by Dr. Mukkamala both in research and classes. A word of appreciation also goes to Dr. Hani Elsayed-Ali for his participation in the committee.

I am indebted to Dr. Ken Birman for his valuable input and constructive criticism of different aspects of this work. I would also like to thank Dr. Sam Toueg for providing me with both the initial idea and the resources that made Chapter 9 possible. His assistance in revising some of my work is greatly appreciated. I would also like to extend my thanks to Pat Stephenson and Anne Spauster for their help and advice.

Special thanks go to my dear friend and housemate, Ashraf Wadaa, for his help, support, and fruitful ideas that positively contributed to this work. Our long discussions and his cooking skills made my life much more enjoyable. Thanks are also due to Mohamed Eltoweissy for his comments on different parts of this work, and to Jonay Campbell for editing this dissertation. My thanks also are extended to my friends Nahil Sobh, Alaa Elmiligui, Rafat Shaheen, Hussein Moustafa, and Ashraf Morsi for the good times we had together. Also, a word of gratitude goes

to my professors at Alexandria University for their support and encouragement.

Finally, words do not begin to express my deepest gratitude to my family, in particular, my parents who have overwhelmed me with their love, warmth, care, support, and encouragement. Their sacrifice, endurance, and inspiration are beyond description. Manal, Walid, and Marwa, being your brother is really marvelous. In addition, I will never forget the encouragement, care, and support of Drs. Mohamed Fahmy and Aly Fahmy, my cousins. They will never believe how much difference they have made in my life and career. I am truly in debt to both of them.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Computation and communication interplay; hence, you compute distributedly. Computation affects the state of resources. Communication publicizes these effects; hence, a distributed behavior results.

To compute is to effect an orderly change upon the state of computing resources up to some specification. If computing resources are distributed and are usable in their distributed fashion, a form of "distributed computing" is being exercised. Upon numerous practical grounds, distributed computing is appealing as a general-purpose computing paradigm. The case for distributed computing is amply presented in the literature, as can be found in references [72, 42, 24, 23, 60].

The hallmark of the distributed computing paradigm is the notion of distribution. The distribution of resources and, hence, of computation results in applications that are distributed. A "distributed application" describes a situation in which several concurrent processes, customarily referred to as application processes [24], affect the state of disjoint subsets of resources to realize a logical specification. In a distributed application, computation is distributed across a number of application processes. Thus, application processes must act in concert to ensure that their behavior indeed realizes the goal specification. This concerted action

is possible only if the state of resources in accordance with one process can be shared by other processes, which allows them to adjust their future actions accordingly. If a platform of independent and network interconnected computers is used for distributed computing [60], then communication via the network becomes the only vehicle for state sharing. Hence, the distributed computing paradigm can be considered as an interplay of computation and communication.

Collectively, the set of communication functions offered in a distributed computing platform are incorporated into communication services [13]. Different services generally possess different properties and, hence, are suitable for different classes of distributed applications. A principal concern in designing communication services for distributed computing is to ensure that the communication needs of different application classes in a given domain are satisfactorily accommodated by the underlying communication services. Customarily, application classes would impose conditions on the reliability, performance, and other properties of communication services [69, 9]. Unfortunately, efforts in designing efficient general-purpose communication services for distributed computing have encountered major difficulties. Different application classes need different types of services. Also, for a given type of service, the conditions prescribed by different classes of applications can vary considerably. Furthermore, the accurate identification of the needs of future applications a priori is often not feasible. As a result, research efforts in this area have been driven to present solutions that establish specific services for narrow classes of applications.

A communication service of major importance to distributed applications is multicasting [38, 51]. In multicasting, an application process sends a message via the network to a subset of other application processes, called recipients. The multicasted message is guaranteed to be received by either all or none of the recipients. A multicast service that offers this guarantee is commonly referred to as an atomic

broadcast [19, 12]. If a multicasted message is guaranteed to be ultimately received by all correct recipients, then one has a reliable broadcast service. Atomicity and reliability are highly desirable properties because atomic reliable broadcasts simplify the design of distributed applications. In many practical situations, messages need to be delivered according to a specific ordering criteria, as well as in a reliable atomic manner. The later condition is met by ordered atomic reliable broadcast services [12, 56, 53, 41]. Message delivery order is critical in a large class of practical applications. For example in multicasting voice data, successive packets can be incorporated into messages that are eventually multicasted. The delivery of both the transmitted packets and the corresponding messages in their chronological order is considered a correctness criterion for such an application.

Numerous protocols for ordered reliable atomic multicasting have been proposed in literature [19, 12, 63]. The majority of these protocols assume an environment of a single LAN that has multicasting capabilities [19, 63]. Unfortunately, almost all of the proposed protocols can enforce only a single ordering criterion for message delivery across all active distributed applications. Birman and Joseph [12] have proposed a multicasting protocol that can handle multiple message streams, each associated with a single ordering criterion. Messages from the same stream are ordered for delivery according to this criterion independent of the recipient. Effectively, this deprives the recipients of their autonomy in determining their own criteria for ordering delivery of incoming messages.

Multicasting efforts thus far have largely failed to address the situation of an interconnected group of networks. As a result of the interconnected network architecture, a multitude of issues that have not been addressed before must now be handled. Certain members of the interconnected networks possess no multicasting capabilities, contrary to the assumption made thus far in the majority of multicasting protocols.

In the development of new multicasting protocols, the maintenance of autonomy in managing local traffic in each network and the consequent heterogeneity in communication services across different networks are principal concerns [40]. In practice, neither the enforcement of a single ordering criterion nor the deprivation of recipient control over their own ordering criterion is acceptable. Performance tradeoffs in a single LAN can be invalidated in interconnected networks because networks of different speeds can be part of the same interconnection. Failure models for interconnected networks are different from that of a single LAN because the network elements differ in size and type. This variety has direct consequences on the reliability problem and its solutions.

Upon numerous grounds, interconnected networks are envisioned to be the underlying architecture for distributed computing in the next decade.

Supporting arguments can be found in references [4, 23, 31, 47]. The research community has taken up the challenge of studying issues particular to such computing architectures and the subsequent formulation of an appropriate infrastructure of communication services for these platforms. Our dissertation is an effort to promote and design a core for an advanced multicasting infrastructure for interconnected networks.

1.1 Motivation

Urged by both economic and political forces, businesses, government, and other entities are experiencing a massive drive for interconnection. The affordability of relatively mature network computing resources has resulted in a proliferation of networking on both a small and large scale in the past few years. Today's economic landscape is witnessing intercorporation interaction as never before. For example, mergers to facilitate cost consolidation and to gain a market share are

commonplace. Business dynamics have forced major corporations to pool expertise in order to cut down on the cost and turn-around time of products. These practices have magnified the problem of computing-resource interconnection and, hence, the interoperability of autonomous networks.

Undoubtedly, the advent of the information superhighway as envisioned has introduced new aspects to the problem. These interconnections between networks become the means by which indispensable resources can be provided and accessed. The dense interconnection and economy will continues to encourage a shift toward utilization of the superhighway. These activities mark no less than a revolution in the notion of distributed applications. The current state of the art in communication services in general and multicasting services in particular does not suffice as an infrastructure for interconnected autonomous networks. Below is a discussion of the reasons behind our conjecture.

1.1.1 Autonomy

Because processor cycles, storage bytes, and high-speed network bandwidths are becoming increasingly affordable, the surge toward distributed computing is gaining new momentum. Computing platforms that contain interconnected networks are emerging as new intracorporation and intercorporation distributed applications are introduced. Because these platforms connect networks that have been established and managed by autonomous entities, a principal concern is autonomy [40]. Autonomy is relevant in two ways to the multicasting problem. First, the autonomy of different applications in determining the properties of the communication services they invoke should be upheld. For multicasting, each application should have the autonomy to determine the delivery order of messages to application processes. In a large-scale interconnected network (e.g., the internet), marked diversity is evident in the application space, which renders this autonomy a must.

Second, the autonomy of each connected network to exercise control over communication activity that does not cross the network boundary (i.e., local activity) must be preserved. For multicasting, one consequence is that each network must be allowed the autonomy to use its own multicasting service protocol(s) in handling local activity. Another consequence is the ability of each network to upgrade or change local multicasting protocols at their own discretion.

> *Autonomy, as manifested by the message delivery order and the use of local protocols to handle local activity in each network, is characteristic of multicasting in interconnected networks.*

1.1.2 Heterogeneity

The interconnected networks are largely made up of networks that were established and managed in the past by autonomous entities. Heterogeneities in network services should be accepted as commonplace. Heterogeneity across different networks is be manifested, in part, by differences in the types of communication services and in the properties of a given type of service. For multicasting, heterogeneity in the local protocol(s) poses a serious problem for distributed applications across network boundaries. Processes of one application may belong to different networks; hence, multicast messages must be effectively "handed" to local multicast protocols for delivery to uphold network autonomy.

> *The interoperability of interconnected network multicasting protocols and local multicasting protocols in each network constitutes a major concern in interconnected network multicasting.*

1.1.3 Performance

Performance of communication services in a single LAN and in interconnected networks are largely different problems. The multiplicity of networks in an interconnected environment introduces several issues (i.e., the speed of each network, the performance of each local multicasting service (if any), and the performance of different routers and gateways). Each of these factors can exhibit large variation. For example, a slow router or gateway can become faster with upgrades or load fluctuations. An efficient multicasting protocol can experience performance degradation if additional hosts and users are connected to the local network. Up to the autonomy condition discussed above, local multicasting of messages in each network is probably handled by local protocol(s). Therefore, in this case the performance of multicasting activity is a function, in part, of the performance of local multicasting in the different networks. Performance tradeoffs are difficult to define, much less to take into account in designing multicasting protocols.

> *Multicasting protocols in interconnected networks should maintain acceptable performance across constant changes in the performance of different local networks and their services.*

1.1.4 Resiliency

Interconnected networks introduce a larger number of hardware and software elements that are susceptible to failures. The probability of a single element failure is largely increased. Several types of failures in computer networks defined in the literature undermine the resiliency of network service protocols [44]. Specifically, connection failures due to the failure of connections, routers, and gateways can lead to message losses. The fact that these failures are expected to occur at a higher rate in interconnected networks adversely affects the message-loss problem. Net-

works interconnected by gateways results in a faster network partitioning [43, 66] (i.e., one network becomes disconnected from the others because of a gateway failure). Two partitions are formed; one contains the disconnected network and the other contains an interconnection of all other networks. In a single LAN, network partitioning is not relevant and is not addressed by multicasting protocols for this environment. For example, a connection failure in a token-ring network breaks the ring and may potentially bring down the entire network rather than a part of it. Several issues in regard to network partitioning have yet to be addressed, such as whether one or both partitions should continue to function or whether one partition should function in a restricted manner [67]. Another issue is the management of multiple partitions.

> *Contrary to a single LAN environment, network partitions are common enough in an interconnected environment to seriously impact reliability. Thus, this problem must be addressed by any practical multicasting protocol for interconnected networks.*

In summary, distributed computing on interconnected networks is expected to be defacto in the near future. Because of economic and political forces, various autonomous entities are involved in interconnection efforts in answer to a growing need for cooperation. Affordable, mature computing and networking resources are essential to the economic feasibility of these efforts. Unfortunately, a communication infrastructure suitable for interconnected networks must address a host of problems that are not addressed by current solutions for single LANs. Our dissertation presents a multicasting infrastructure for interconnected networks. The maintenance of intranetwork autonomy in managing communication in the local network boundary is a major concern of such an infrastructure. Another concern is the accommodation of the heterogeneity of local communication services by allowing for the interoperability of interconnected network multicasting protocol(s)

and local network multicasting protocols (if any). Such interoperability causes the performance of multicasting in interconnected LANs to be dependent on the performance of the local communication services in each network. Multicasting service performance must be addressed in a framework that incorporates the effects of all local multicasting protocols. The multicasting infrastructure must have resiliency against both message loss and network partitioning built into the design. The characteristics discussed above stand in sharp contrast to the assumptions that drove efforts toward the current state of the art in multicasting services for distributed computing. Communication services have always had a far-reaching effect upon the success of distributed applications. For interconnected autonomous networks, a multicasting infrastructure is no less than a cornerstone that will enable technological advance; hence, our motivation is clear.

1.2 Objectives

Our main objective is not only to design a global standard protocol that will support a reliable ordered multicast service in an interconnected group of LANs but also to fully utilize the underlying communication network capabilities. Furthermore, we search for a method of orchestrating the interaction between different ordered reliable multicasting protocols. This method, if realized, will introduce a greater opportunity for cooperation between all distributed applications that use different ordered reliable multicasting protocols. The multiprotocol interaction will allow local administrations to have a higher level of autonomy in selecting their local protocols. We also introduce a new approach that not only allows greater interaction between different groups in distributed applications but allows local sites and groups to have more freedom in selecting their ordering criteria without preventing interaction with groups that have different ordering criteria. Further-

more, if multiple ordering criteria are able to be used in the same environment, then groups will not be forced to impose a costly ordering condition because they must interact with another group that requires this ordering condition.

1.3 Contribution

To meet our objective, we investigated the ordering requirements of multicasting groups. As a result of this investigation, we have defined the communication environment from both a physical and logical perspective. We then analyzed the existing multicasting protocols to identify a link that would allow our protocol to interact with other protocols. The complexity of this problem urged us to pursue a multistep approach to the solution. This type of approach allows us to better understand both the interaction between the reliable multicast protocols and our *INTER LAN multicast protocol* and the expected effects of our protocol on the total performance of the system. We then developed a set of protocols that achieves the objectives. Our research includes the following outcomes:

- We characterize the multicasting requirement of interconnected LAN environments.
- We introduce the hierarchical communication structure adopted by the protocols to achieve a lower delivery delay by using the inherent hierarchy of the internet and recommend a heuristic algorithm to build this structure.
- We present a set of protocols that uses the presented communication structure to achieve total or causal ordering between multicasted messages. This set of protocols includes the Bottom-Up Stamping (BUS) protocol; the Bottom-Up Stamping (BUS-TO) protocol, the Total-Order version; and the Top

Down Stamping (TDS) protocol. These protocols allow multicasting to be performed over a set of interconnected LANs.

- We introduce the idea of multiorder delivery in a multigroup environment and the development of MLMO: the Multi-LAN Multi-Order protocol that allows the enforcement of different ordering criteria over multicasted messages.

- We tackle the multiple protocol interaction between our protocol and existing protocols and between the existing protocols with one another. The study of this problem has resulted in the development of INTER: a multiprotocol interface that provides a vehicle of interoperability between these protocols.

- We examine the effectiveness of our approach in using the hierarchical communication structure and determine its effects on the general behavior of the protocol. Our results indicate a shorter delivery time and a decrease in the number of protocol messages; these results show that our protocols meet the established performance criteria. We also present several failure-handling protocols that can be incorporated within our protocols. Our protocols are resilient to send-omissions, receive-omissions, and network-omissions, and can handle process failures and network partitions.

1.4 Outline of Dissertation

The remainder of this dissertation consists of the following:

Chapter 2, Background. Introduces distributed systems. The different characteristics of a distributed system are discussed and the key issues in heterogeneous systems are identified. The chapter introduces the ordered multicasting protocols as a way to solve some of the problems in distributed systems. It also presents a survey of different ordered reliable multicasting protocols relevant to our study,

along with a comparison between the different costs associated with each of them.

Chapter 3, Multicasting in Interconnected Networks. Discusses the different problems encountered in multicasting in interconnected networks in general and in heterogeneous distributed systems in particular. The chapter provides examples of some of the problems, and details our approach to the solution. The chapter proceeds by introducing the communication structure that will be used by our multicasting protocol suite in message delivery. It defines the set of rules and terms that will be used to build this structure.

Chapter 4, BUS: The Bottom-Up Stamping Protocol. Presents a new ordered multicasting protocol that achieves a causal order among multicasted messages. This protocol uses the communication structure presented in chapter 3. The chapter also validates the correctness of the protocol.

Chapter 5, BUS-TO: Bottom-Up Stamping Protocol (The Total-Order Version). Defines another ordered multicasting protocol that achieves a total order among messages. The chapter provides a layout of the protocol and ends with a validation of its correctness.

Chapter 6, MLMO: Multi-LAN Multi-Order Protocol. Introduces a new multicasting protocol that can achieve multiorder delivery of multicasted messages in a multigroup environment. The protocol allows group members to span different LANs and enables each group to adopt its own ordering criteria for message delivery.

Chapter 7, INTER: A Multi-Protocol Interface. Discusses the interaction of different ordered multicasting protocols with one other to achieve a consensus order in regard to shared messages. The chapter describes the protocol and introduces the layout of the different modules.

Chapter 8, Performance Issues. Describes several performance issues and provides a simple basis for comparison with other existing protocols.

Chapter 9, Reliability and Fault Tolerance. Presents the failure assumptions handled by our protocol suite and the different procedures that ensure the reliability of these protocols.

Chapter 10, Conclusion and Future Work. Presents a final assessment of the work, the significance of the work thus far, and the future direction of our research.

Chapter 2

Background

2.1 Overview of Distributed Systems

A distributed system is a system with many processing elements and many storage devices that are connected together by an underlying communication system. This feature makes a distributed system potentially more powerful than a conventional centralized system in two ways. First, it is more reliable because functions may be replicated. For example, when one processor fails, another can take over the work. Each file can be on several disks, so a disk crash does not destroy any information beyond recovery. Second, a distributed system can do more work in the same amount of time because many computations can be carried out in parallel.

These two properties, *fault tolerance* and *parallelism*, make a distributed system much more powerful than a traditional centralized system. Although these two properties are characteristics of any distributed system, an exact definition of a distributed system is difficult to determine. Birrell et al. [14] defines a distributed system with a set of *symptoms*. They states that if a system has all of the symptoms listed below, it is probably a distributed system. If it does not exhibit one or more of these symptoms, it is probably not a distributed system. These symptoms can

be are summarized as follows.

- *Multiple processing elements* that run independently. Each processing element, or node, must contain at least a CPU and memory with communications between the processing elements.
- *Interconnection hardware,* which allows parallel processes to communicate and synchronize.
- *Independent failure of processing elements,* to prevent the simultaneous failure of all nodes. A distributed system cannot be fault tolerant if all nodes fail simultaneously.
- *Shared state* that allows recovery from failure. If recovery were not possible, a node failure would cause some part of the system's state to be lost.

2.1.1 The Need for Distributed Systems

Several features of distributed systems and current technology have urged people to move from old centralized systems toward distributed systems. Among the most important features that encourage this migration are the following:

- *Distribution:* Information generated in one place is often needed in another. The workstations and personal computers are connected together because of a desire to communicate and to share information and resources.
- *Expandability:* Distributed systems are capable of incremental growth. To increase the storage or processing capacity of a distributed system, one can add file servers or processors at any time.
- *Availability:* Because distributed systems replicate data and have built-in redundancy for resources that can fail, distributed systems have the potential to be available when failures occur.

- *Scalability:* The capacity of any component of a centralized system imposes a limit on the system's maximum size. Distributed systems have no centralized components; therefore, the maximum size of the system is not restricted.

- *Reliability:* Availability is one aspect of reliability. A reliable system must not only be available, but it must do what it claims to do correctly even when failures occur. The protocols used in a distributed system must not only behave correctly when the functions of the underlying virtual machine are correct but should be capable of recovering from failures of the underlying virtual machine environment as well.

2.1.2 Problems in Distributed Systems

Distributed systems are among the most complicated systems to design and maintain. To quote Mullender [60] "Distributed computer systems have only been around for a decade or so, but they are every bit complicated to design and will take many generations of distributed systems before we can hope to understand how to build one properly."

The basic source in a complexity of distributed systems is that an interconnection of well-understood components can generate new problems not apparent in the components. To clarify this matter, we present some of the problems given in reference [60].

- *Interconnection:* A large number of system problems come about when components that have previously operated independently are interconnected. This was a common type of problem when various computer networks for electronic mail were interconnected.

- *Interference:* Two components in a system, each with reasonable behavior when viewed in isolation, may exhibit unwanted behavior when combined.

- *Propagation of effect:* Failure in one component can bring down a whole network when system designers aren't careful enough.

- *Effects of scale:* A system that works well with ten nodes may fail miserably when it grows to a hundred nodes. This problem is usually caused by some resource that doesn't scale up with the rest of the system and become a bottleneck, or by the use of protocols that do not scale up.

- *Partial failure:* The fundamental difference between traditional, centralized systems and distributed systems is that in a distributed system a component may fail, while the rest of the system continues to work. In order to exploit the potential fault tolerance of a distributed system, distributed applications must be prepared to deal with partial failures.

These problems exist in all computer systems, but they are much more apparent in distributed systems. The distributed system comprises more pieces; hence, the potential exists for more interference, more interconnections, more opportunities for propagation of effect, and more kinds of partial failure.

2.2 Synchronous and Asynchronous Networks

Hadzilacos and Toueg[44] characterize a distributed system as *synchronous* if it has the following properties:

1. A known upper bound exists on the time required by any process to execute a step.

2. Every process has a local clock with a known bounded rate of drift with respect to real time.

3. A known upper bound exists on message delay; this consists of the time it takes to send, transport, or receive a message over any link.

All of the above properties are necessary for the use of timeouts to detect crash failures. If any of the three properties is violated, and a process p timeout on a message expected from a process q, then p still cannot conclude that q has crashed. The message delay could have been longer than expected, the clock used by p to measure the timeout could have been running too fast, or q could be executing steps slower than expected.

A distributed system is *asynchronous* if *no* timing assumptions are made whatsoever. In particular, no assumptions can be made on the maximum message delay, clock drift, or the time needed to execute a step. An asynchronous system is easier to port than those that incorporate specific timing assumptions; in practice, variable or unexpected workloads, network traffic, and other dynamic components that affect performance are sources of asynchrony. Thus, synchrony assumptions are, at best, probabilistic.

Synchronous and asynchronous systems are the two extremes of a spectrum of possible models. Many intermediate models of *partial synchrony* have also been studied [34, 35, 21, 36]. For example, known bounds may exist on clock drift and step execution time, but message delays could be unbounded. Or bounds may exist on clock drift, step execution time, *and* message delay, but these bounds may be unknown.

2.2.1 Autonomy

Node autonomy is one of the keywords in distributed systems, especially in the context of heterogeneous and federated systems. A *heterogeneous database systems* (HDBS), for example, is a distributed database system that includes *heterogeneous database* (HDB) components; heterogeneity means different components

at the database level such as data model, query language, and schema. A *federated database system* (FDBS) is a collection of cooperating database systems that are autonomous and possibly heterogeneous [70]. Webster's defines the word autonomy as "the quality or state of being independent, free, and self-directing." The issue of autonomy in distributed systems is only meaningful in the context of cooperation between nodes. Several reasons make node autonomy desirable in distributed systems:

- *Organizational issues:* In a large organization, distributed-computer-system node autonomy is a natural extension to departmental autonomy.
- *Diversity of local needs:* Different parts of the system can be more easily tailored to the needs of local users.
- *Data security:* For those distributed systems that are sensitive to unauthorized data access, nodes are often responsible for the security of the data they store. In such an environment, node autonomy is absolutely essential to enforce security procedures.
- *Failure/Bug containment:* This help to limit the spread of the effects of local failure at a given node throughout the system. A degree of independence implies that healthy nodes would continue to function in spite of such a failure.
- *Lower costs:* Autonomy can be viewed as cooperation without constant coordination. By reducing the number of messages exchanged among tasks that are executing at different nodes, costs can be decreased.

Types of autonomy

Different types of autonomy are defined depending on the particular way each node in the system exercises its freedom of choice. The degree to which each node in the system exercises these different types of autonomy is difficult to evaluate [40]. Garcia-Molina has presented the following types of autonomy:

- *Heterogeneity:* Each node has the flexibility to select its local resources, mechanisms, and representations.

- *Naming autonomy:* A node may have different degrees of independence in creating and translating names. In particular, *name creation autonomy* is determined by whether a node must secure the consent of any other node to create a name. *Translation autonomy* is the capability of a node to independently translate a name to the corresponding physical address.

- *Communication autonomy:* Each component in the system has the ability to decide whether to communicate with other components. A component with communication autonomy is able to decide when and how it responds to a request from another component in the system.

- *Execution autonomy:* Each node has the right to execute transactions or honor requests at any time. An example of this type of autonomy is the case of database transactions, especially when replication exists.

- *Setting priorities:* Autonomous nodes should be able to unilaterally decide whether to honor foreign requests, taking into account primarily their own interests. Also, *data sharing* is part of our autonomous node. Data security dictates that a node have complete freedom in deciding whether to grant the requested access. A problem occurs here in regard to a read or a write request to a nonreplicated data item. In this case, the node may have some

obligation to not refuse the request. However, even in accepting the request, the node may have the freedom to set the local execution priority of the request.

- *Abort autonomy:* Each node can unilaterally abort a distributed transaction, perhaps even after a decision to commit has been made. This feature can be a problem in that it violates commit protocols; however, it can be beneficial in cases like optimistic protocols for partitioned networks [30], where a node can have execution autonomy during partitions. After the partitions are repaired, some of the transactions may need to be aborted due to conflict.

Cost of Autonomy

Node autonomy forces an overhead on the whole distributed system. Although autonomy is generally viewed as a desirable characteristic for each node, it can impose several restrictions on the behavior of the whole system. However, we see that it has some advantageous properties, such as the possibility of maintaining system functionality in case of partial failure. Among the most costly effects of autonomy are:

- *Correctness:* The introduction of high levels of transaction-control autonomy raises the important issue of execution correctness. For example, one way to maintain correctness in distributed systems is through the use of lock-based distributed-concurrency control mechanisms. When a node has lock autonomy, it can release a lock acquired by a nonlocal transaction and, in doing so, violate an established locking protocol, which in turn may imply a breach of correctness criteria.

- *Timeliness:* With autonomous setting of priorities for foreign requests, no guarantees can be made on how expediently such requests are executed.

Thus, timeliness of execution of foreign requests can be negatively affected by a high degree of node autonomy. In addition, timeliness of update propagation can also suffer because of autonomy.

- *Level of cooperation:* Cooperation involves data and load sharing among nodes. When relatively high levels of cooperation are mandatory, node autonomy is more difficult to maintain. However, autonomy does not necessarily imply refusal to cooperate. In an ideal situation, nodes would be willing to support the highest level of cooperation that would not compromise their individual interests (e.g., data security, and good response time for local jobs).

- *Degree of data replication:* High autonomy in some cases requires data replication, which is yet another price that may have to be paid for autonomy. In particular, replicated data of one type or another makes it easier to provide scheduling, name translation, and execution autonomy.

2.2.2 Heterogeneity

Heterogeneity can be divided into two main categories: *hardware* and *software*. In *hardware heterogeneity,* the nodes have different hardware configurations; in *software heterogeneity,* the nodes have different software running, including the operating system and all implemented algorithms and protocols. For example, a heterogeneous database has individual nodes that are free to choose their local schemata, concurrency control, etc.

Is homogeneity more beneficial than heterogeneity? To assume a homogeneous system is an imposed assumption, so it is more realistic to assume a *heterogeneous system.* Hardware heterogeneous systems have different computer capabilities (resources, speed, or manufacturer) and/or different networks (Ethernet, Token-

ring, or point-to-point connections); software heterogeneous systems have different operating systems, applications, and protocols that run at each site.

Realistically, the homogeneity assumptions cannot be imposed, and the interaction and flexibility of each site in choosing its own software and hardware while being part of an integrated distributed system cannot be restricted. We believe that this is not the only issue because, although some algorithms function well in certain environments, they function much worse in other environments. For this reason, the use of different algorithms is justified because the expected behavior and performance of an algorithm depends mainly on the environment in which it is running. Several criteria are involved in selecting the algorithm; because several selections can be made, when a local algorithm is necessary in a distributed system, the node requires the freedom to select the appropriate algorithm, depending on its local environment. The possibility exists that the heterogeneity of the algorithm over the nodes would enable better behavior across the whole system.

2.3 Ordered Reliable Multicast

At the heart of any distributed system is the problem of *transferring* information between cooperating processes. Broadly speaking, this can be done in one of two ways: by permitting the processes to interact with some common but passive resource or memory, or by supporting message exchange between them. Advantages and disadvantages are associated with each approach; hence, the appropriate approach to information transfer for a particular problem must be determined by an analysis of the characteristics specific to that problem.

In its simplest form, multicasting causes a copy of a message to be sent to each one of several destination processes. The multicast operation must take care of the possibility of failure of one or more of the participating processes. It must also

handle the problem of lost messages. A multicast that provides such guarantees is called a *reliable multicast.* Reliable multicasts are implemented with special protocols that detect failures and/or take compensating actions. However, under certain failure patterns, no protocol can guarantee the delivery of a multicast message to all operational destinations. For example, the sender could crash before it actually sends out any messages. Even if it manages to communicate with some other processor before it crashes, the other processor could experience a failure before it communicates with any other process. In general, a set of failures in an early stage of a multicast protocol could wipe out all knowledge of the message. Reliable message delivery must be an all-or-nothing operation. More precisely,

> If processor p sends a message m to a set D of destination sites, then the system will eventually reach one of the following two states:
>
> 1. For all $q \in D$: q has received m or q has crashed.
> 2. Processor p has crashed, and for all $q \in D$: q has crashed or q will never receive m.

In addition to atomicity, reliable multicast guarantees a particular ordering of messages. This enforcement of order increases the latency, results in additional communication, and requires that the messages be stored for some time during the execution of the protocol. Control messages associated with a multicast protocol represent additional overhead. This overhead depends on the degree of fault-tolerance achieved and the type of order enforced.

Here, no shared memory exists between sites; therefore, the only form of communication between them is through the network, which enables messages to be transmitted from any processor to any other processor in the system. Message transmission is asynchronous in the sense that sending and receiving operations do not have to wait for one another for communication to occur, and message

transmission times are variable.

The multicast protocols are located above the transport layer; the protocols enables the site to send a message from one process to a set of processes. A process that wants to perform a multicast presents the multicast layer with a message and a list of destination processes for that message. The multicast layer uses the destination list to compute a set of sites that must receive this message, and uses the transport layer to send a copy of the multicast message to each of these sites.

Many *reliable atomic multicast* protocols have been proposed [19, 28, 12, 26, 56, 53]. These protocols differ in the way they achieve the order and the reliability of message delivery. Also, they differ in the assumptions adopted, especially for the communication network. These differences make specific protocols appropriate for different environments.

By imposing a consensus total order on multicast messages, one of the traditional problems in the design of distributed systems can be eliminated; the lack of a global system state. Without a global system state, complex reasoning is necessary about what information is known to each processor. The agreed total order of multicast messages enforces a common history and, thus, a common system state. Each processor maintains as much of the system state as necessary for its functioning. This simplifies the design process of a distributed system.

The existence of several distributed applications running on different LANs and distributed applications that span different LANs force interaction between the protocols on these LANs. This interaction is forced by the need for the applications to cooperate. The problem here is the heterogeneity that would normally occur due to the interaction between different autonomous systems. This problem is similar to that encountered with *heterogeneous database systems*, in which different *DBMS* used in different sites cooperate in spite of the heterogeneity of the system.

The cooperation among different groups with different *reliable atomic multicast* protocols is a real problem. However, this problem has never been tackled in record research performed in multicasting [20]. A solution to this problem with a reasonable cost protocol would revolutionize multicasting, similar to the achievement of allowing heterogeneous databases to interact [10].

We foresee the possibility of several LANs that each run different reliable multicasting protocol with different LAN protocols such as ethernet, token-ring, or just point-to-point links. Our purpose here is to develop a protocol that will orchestrate the cooperation between these protocols to achieve a reliable ordered delivery service for InterLAN messages. Our main objective here is not only to design a global standard protocol that can support a reliable atomic multicast service in an interconnected group of LANs, but also to achieve full utilization of the underlying communication network capabilities.

2.3.1 Aspects of Reliable Atomic Multicast

Ordering

One of the important properties available in most of the reliable multicast protocols is message ordering. An order, enforced on the messages delivered to the application layer, helps to decrease the complexity of the protocols that run over the multicasting protocol (see section 2.3.2 for the importance of the ordering property).

The multicast protocol must guarantee the order in which messages are delivered to the destination processes. The following ordering criteria are common.

- *Single-source ordering:* If messages a and b are sent from the same site such that a is sent before b, denoted $a < b$, then all destination sites that receive both a and b will deliver them in the same order.

- *Multiple-source ordering:* If messages a and b are sent from two different sites, then all the destination sites within the same group will receive them in the same relative order.

- *Multiple-group ordering:* If messages a and b are sent from two different sites, then all the destination sites, whether in the same or two different groups, will get them in the same relative order.

In certain applications, the receipt of messages in different order will lead to inconsistency or deadlock problems. For example, consider a bank with two main computers. Each computer has a copy of the entire banking database and can process all transactions that arrive from the branch offices(the second computer is necessary for disaster recovery). These two main computers constitute a multicast group, and each branch office is a potential source site. Transactions should be executed in the same order at both computers, otherwise the database state on one machine will differ from that on the other. For instance, consider a deposit and a withdrawal to the same account. Assume that if the withdrawal is done first, then an overdraft may occur and a penalty is charged. However, if withdrawal follows the deposit, then no penalty is incurred and the resulting account balance is different.

If we observe the locking procedure in a distributed database, we get some idea of the importance of order. Assume that we have two transactions $T1$ and $T2$ initiated from two sites A and B, respectively. Transaction T1 requests a write lock on data item X and Y; T2 requests a write lock on X and Z. Assume that item X is replicated on a set of sites and we are using the write-all-read-one algorithm. Site A will multicast the write lock request of T1 for X on all sites of the replica set of data item X. Similarly, site B multicasts the write lock of T2. If the order of arrival of the lock requests at the replica sites of X are not the same, then a lock on the data item for T1 would result at some sites and for T2 at the others.

The lock request will not be granted to either, and as a result, both transactions will be aborted. If the order is enforced, then one of the two transactions would be allowed to own the lock, which decreases the number of unnecessary aborts.

For some applications, not only must multicasts be received in the same order at the different destinations, but this order must also be the same as some predetermined order (called *causal order*). For example, consider a computation that first sets copies of a replicated variable to zero and later increments that variable. In this case the two operations must be carried out in the same order at all copies, and the increment must *always* occur second. The *potential causality* in an asynchronous distributed system, in which information is exchanged only by transmitting messages, is studied by Lamport [50]. In such a system, a multicast B is said to be potentially causally related to multicast B' only if:

- *Rule 1.* They are sent by the same process and B'occurs after B; or
- *Rule 2.* If B is delivered at the sender of B' before B' is sent, or if B is delivered at the sender of B'' before B'' is sent and B'' is delivered to sender of B' before B' is sent (and so forth with any similar dependency).

In an asynchronous system, any protocol that guarantees ordering properties requires every message to take at least *two* hops before it is delivered. Consider, for example, a system with two processors *p1* and *p2*. Process *p1* multicasts a message *a*; at the same time, *p2* multicasts *b*. Both messages are addressed to both processors. We claim that either message *a* also needs at least two hops (to *p2* and back to *p*1) before it can be delivered at *p*1, or message *b* needs two hops. Assume that the protocol delivers *a* to *p*1 in one hop. This means that *p*1 sends *a* to *p*2 but delivers the message locally without waiting for a reply from *p*2 (See Figure 2.1). At the time of this local delivery, *p*1 may not yet know that *p*2 has sent a multicast. If the message *b* from *p*2 to *p*1 is delayed long enough, the protocol

will deliver a before b at $p1$. Similarly, the possibility exists that at $p2$, b will be delivered before a, which would violate some relative order constraint [68].

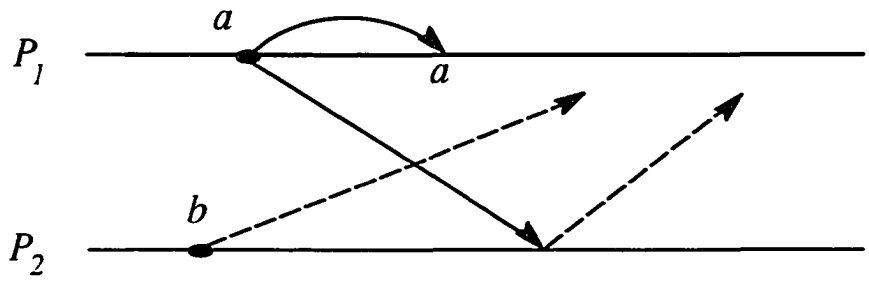


Figure 2.1: Process $p1$ delivers a locally without waiting for any messages from $p2$.

Reliability

Reliability is concerned with the behavior of the system in case of a failure. The types of failures that may be encountered are described below.

- *Transient failure:* This failure type causes some messages to be lost, possibly due to buffer overflow.
- *Persistent failure:* This failure type causes network partitions, in which some group of hosts is disconnected from the other multicast groups. This results in a total loss of messages multicasted by sites from the other partitions.

Recovering from a persistent failure is more costly than recovering from a transient one because it essentially requires remulticasting of multiple messages to multiple destinations [39]. No protocol exists that is resilient to network partitioning when messages are lost (i.e., the possibility always exists that some sites block when networks become partitioned [71].

Failure properties of the described network are characterized by the following set of assumptions. Sites can exhibit omission failures (i.e., they can fail to send a message when prescribed to do so). Note that omission failures cover the case of site crashes. We assume, however, that no malicious failures occur (i.e., messages are not altered or generated when they are not supposed to in order to disrupt the correct functioning of the system). Detection of a failure is a complicated issue because many problems mimic failures. For example:

1. A series of message losses can mimic a failure.
2. Failure detection by timeout is not reliable. For example, slow computers or heavily loaded networks can trigger the timeout when a loss or failure did not actually occur.
3. The order in which failures are perceived to have occurred may vary from process to process.

Atomicity

Because we are multicasting the messages to a set of members, the issue of guaranteed delivery is important. Are we ensuring a delivery of the messages to all operational members of the group? The *atomicity* properties ensure that a message multicasted by a member of a group will either be received by all or none of the operational sites of the group. Actually, the atomicity property is important because without this property assumptions in regard to the state of the other members would be difficult. The atomic multicast problem can be viewed as a multicopy update problem, where the data copies are the total orders of the multicast messages maintained by participating processors. In principle, existing protocols for consistent updating of multicopy databases [73, 7, 1, 2] could be used for atomic multicast. However, these protocols would be inappropriate in practice because

their generality would introduce unnecessary complexity and latency for atomic multicast applications.

2.3.2 Importance of Reliable Atomic Multicast

When processes cooperate to implement some distributed behavior, an important issue is to ensure that their actions will be *mutually consistent.* Not surprisingly, the precise meaning that one attaches to consistency has important implications throughout a distributed systems that presents coordinated behavior. Transactional *serializability* is a widely accepted form of consistency [61]. This leads to a natural question: should all types of distributed consistency be viewed as a variant form of transactional consistency, or are there problems that can only be addressed with other methods[13]? The issue here may concern the *isolation* properties enforced over the transactional model. The isolation properties result in non interference between processes. Not all distributed transactions conform to a similar notion of consistency. This leaves us with two choices for solving the problem of consistency criterion:

- Extending the transactional model to cover the requirements of distributed applications. Some work has been done in this area [45, 54, 52]. The trouble with these models is the extra complexity introduced.

- Developing a different notion of consistency for distributed computation that would fit the problem in a better way. The main approach here is to enable programs to reason about each other's states and actions [13].

Any notion of distributed consistency will be incomplete unless it takes into account the *asynchronous* nature of the systems in question. This notion would require a special protocol to allow each process involved in a distributed computation to have a view of each participant state. This allow each process to get to

a common decision due to the state information available at each process. The agreed total order on multicast messages enforces a common history and, thus, a common system state; each processor maintains as much of the system state as necessary for its functioning. Consequently, distributed systems need not be more difficult to design than asynchronous centralized systems [56].

One of the major advantages of reliable atomic multicast is that it greatly simplifies the implementation of distributed applications. In most general terms, a distributed implementation of a service runs like this:

- A client at processor i invokes an operation a.
- Processor i starts an *agreement protocol* among all processors to decide on the effect of operation a and its return value.
- When the protocol terminates, the result is returned to the client.

Schmuck [68] showed that in order to obtain an implementation to any special problem, it is sufficient to have the agreement protocol establish a *global order* on all operations invoked by different clients in the system. Such implementation gives a correct solution for any specification. Therefore, the execution under reliable multicast protocol will be greatly simplified, because no agreement protocol need to be managed by the distributed application. Hence, a distributed implementation of a service runs as follows:

- A client at processor i invoke an operation a.
- Processor i puts operation a (including its parameters) into a message and multicasts it to all sites in the system (including itself).
- Other processors that receive this message update their local state.
- When site i receives its own message, it also updates its state and at that time computes the result to be returned to the client.

Aside from some strong "impossibility results" [37, 34, 58], existing asynchronous agreement protocols are very expensive, and require a large number of messages to reach agreement in the absence of failures and many more messages in the presence of processor failures or communication errors [62]. Thus, all activities that need an agreement phase, which is essential to distributed systems, are rather expensive. Multicasting protocols can potentially eliminate the need for an agreement protocol, which reduces the total cost of reaching an agreement in a distributed application.

Which a multicast protocol, the agreement process is highly efficient. For example, locking records in a distributed replicated database typically requires only a single multicast message to claim a lock and a single multicast message to release it. Based on this strategy, a simple and efficient, yet very robust, distributed systems can be designed, such as distributed operating systems and distributed transaction processing systems [56].

Guaranteed delivery relieves application processes from implementing special protocols for message delivery. Atomicity ensures that a multicast message will be delivered to every operational destination or none. A delivery order of an application's messages from any single site is often important and, therefore, should be preserved to ensure the correctness of the application. For example, the file lock and unlock messages that originate from a given site in the distributed two-phase locking scheme illustrates the need for delivery in the order that messages are produced. In contrast, the ordering of the multicast messages that originate from different applications need not be constrained. Nevertheless, all messages sent by different sites are still delivered in the same arbitrary order at all sites.

On the other hand, atomic multicast makes the design of fault-tolerant distributed applications much easier because it reduces the uncertainty about the system state caused by message delays and failures in the system.

Multicasting can provide large performance improvements for distributed fault-tolerant systems when appropriate protocols are used. The use of multicast communications will make the development of high-performance transaction processing systems, feasible with fault-tolerant distributed architectures rather than the centralized architectures that are currently used.

2.4 Multicast Protocols

2.4.1 Chang and Maxemchuck: (Token passing approach)

Chang and Maxemchuck [19] describe a family of protocols that achieve ordered reliable multicasts. The main idea behind their protocols is to make a general system appear to be a combination of two simple systems, one with a single receiver and the other with a single transmitter. A system with many transmitters can be made to look like a system with a single transmitter by passing all the messages through a primary receiver called the *token site.* The *token site* then retransmits the messages to the receivers. The system operates as a positive acknowledgment system between the sources and the token site and as a negative acknowledgment system between the token site and the remaining receivers.

Their protocols do not require that the transport layer provide reliable point-to-point transmission; unreliable datagrams suffice because the retransmission of lost messages is built into their protocols. In these protocols,

- One member of each group of processes is assigned a token and is called the *token site*;
- the token site assigns a timestamp for each multicast, and multicasts are delivered at all destinations in the order of their timestamps, which ensures that all multicasts to a group are delivered in the same order to all members

of the group;

- The protocols require that the token be periodically transferred from site to site; the list of possible token sites, called the *token list* is maintained at each of the token sites, and a token site passes the token to the next site in this list; the protocol operates correctly as long as the number of failures that occur is less than the size of the token list;

- The sites go through a *reformation phase* whenever the token list has to be changed, either because of a failure or because a new site is to be added to the list.

Each protocol in this family of protocols has different rules in passing the token to the next site in the token list. These rules determine the various costs for the protocols which will be described shortly.

In the protocol by Chang and Maxemchuck, a message may be committed and its memory discarded only when the token has been passed twice around the sites in the token list. At the end of the first round the message has been received everywhere; at this point copies can be safely delivered. At the end of the second round the message has been committed (delivered) everywhere; the processes can safely discard any status information needed during the protocol. Thus, the rate at which the token is passed from site to site and the size of the token list determine the latency as well as the storage cost (because information about the messages must be stored until it is committed). If the token is passed rapidly, then the latency and storage costs are minimized; however, unless special hardware can be exploited, such as Ethernet multicast, the communication costs will go up (control message overhead will be N or higher). The communication costs may be reduced by passing the token infrequently, but this would increase the latency and storage costs. In the limit, if the token is never passed, the additional communication goes

down to one acknowledgment message per multicast, but the latency and storage costs go up to infinity and fault-tolerance is lost.

The drawbacks of these protocols emerge from the fact that small latency and high resiliency contradict one another. Also, the protocol enters a reformation phase every time a site failure or recovery is detected. Site autonomy represents yet another problem, for example, the decision to go off-line will be costly because it requires the initiation of a reformation phase, which forces the normal operation of the multicast to be delayed until the reformation succeeds. This cost is unavoidable in ring-based multicast protocols because such protocols require global consensus on site membership in the system following site failure and recovery.

2.4.2 Birman and Joseph: (ISIS)

The ISIS system adopts an approach that is different from Chang and Maxemchuck [19]; it is based on synchronous execution, whereby every process sees the same events in the same order [12]. The problem with synchronous execution is its cost. The ISIS system provides an illusion of synchronous execution, called *virtual synchrony* [11], in much the same sense that transactional serializability provides the illusion of a sequential transaction execution.

The ISIS broadcasting protocol avoids some of the problems that occur with the protocol of Chang and Maxemchuck [19] presented earlier. The protocol does not multicast the messages to all sites of the distributed system and provides different primitives that help to relax the total order.

The ISIS system provides the following group of primitives to help perform the multicasting operations:

- **ABCAST**: This primitive provides an *atomic* broadcast for data, where the order in which data are received at a destination must be the same as the order at other destinations, even though this order is not determined in

advance. The set of processes to which the message must be delivered will receive it in the same order relative to another ABCAST message that has some overlapping destination.

ABCAST operates by assigning a timestamp to each broadcast and delivering messages in the order of the assigned timestamps.

The *timestamp* assignment costs at least two round of messages. Also, a message must wait for all messages with smaller timestamps to be delivered. Extra storage is necessary for queues and for the copies kept for retransmission requests.

ABCAST requires $2N$ protocol messages per broadcast received under normal conditions, where N is the number of sites in the broadcast group. Messages received by a site cannot be delivered to the receiving process when the network is partitioned.

- **CBCAST**: This primitive, called the causal broadcast primitive, like ABCAST provide an *atomic* broadcast for data but differs because it allows a certain predetermined order to be enforced.

  CBCAST is used to enforce a delivery order, but with minimal synchronization. The CBCAST message specifies the parameters over which the order will occur. The *potential causality* [50] is the main criteria for ordering messages using the CBCAST protocol. The issue of enforcing causal order between all messages may not be required by all applications; as a result, CBCAST only enforces order depending on the parameters specified by the application. This allows more flexibility for the applications and decreases the overhead required to enforce a total order.

- **GBCAST**:

The GBCAST primitive is used to inform operational group members when another member fails, recovers, joins, withdraws, or experiences any other change. It is used to update the group view that represents the group site state.

A GBCAST message must be ordered relative to other GBCAST messages sent to the site, as well as relative to the ABCAST and CBCAST messages. In addition, the GBCAST messages must be delivered after every message from the failed process has been delivered.

Both GBCAST and ABCAST are normally invoked synchronously to implement remote procedure calls by one member on all members of its process group. The CBCAST primitive is almost invoked asynchronously, which represents the main source of concurrency in the ISIS system.

The main features of the ISIS primitives follow:

- They allow the join, withdraw, and recover procedures to be less costly. This cost reduction helps in the implemention of the dynamic group.
- They provide group addressing.
- The system does not assume a LAN with special broadcasting capabilities.
- In the case of partitions, the operations are resumed in the partition with the majority of sites.
- Several ordering primitives allow more ordering precedence to the applications.

The disadvantage of the ISIS broadcast primitives is that they do not use any of the broadcasting capabilities that can exist on the underlying communication network. The ISIS does not survive network partitioning.

2.4.3 Melliar-Smith et al.:(Trans-Total protocol)

Melliar-Smith at al. [56] presented a broadcast mechanism that allows both reliability and order to be maintained between broadcasted messages. They presented two protocols that interact together to ensure reliability and ordering:

- *Trans*: an efficient broadcast protocol that ensures that every message received by any operational processor is also received by every operational processor, and
- *Total*: responsible for enforcing a total order on broadcast messages and for ensuring that even in the presence of failure all operational processors agree on the same sequence of broadcast messages.

A fundamental assumption is that that the underlying communication network possess some broadcasting capabilities. The model also assumes that processors are subject to fail-stop, omission, and timing faults, but not to malicious faults. In order for this algorithm to functions efficiently, they assume that the broadcasted message is received immediately or not at all. This protocol is able to accommodate to network partitioning faults; the partitions with at least $2N/3$ processors can resume operation, where N is the number of sites.

The idea behind the Trans protocol is that acknowledgments for broadcast messages are *piggybacked* on messages that are themselves broadcasted and typically seen by all other processors.

The Trans protocol provides a partial order of messages at all the sites, and to achieve a total order they use the Total protocol to transform the partial order to a total order. Their protocols require one broadcast message per agreement, and they reach this agreement after $\lceil (N+2)/2 \rceil$ broadcast messages from distinct processors.

The problem with this approach is that both the Trans and Total protocols assume that the communication interface will include a special interface processor and extra buffer space sufficient to receive, buffer, process, and acknowledge every message delivered by the communication medium.

2.4.4 Luan and Gligor: (The consensus protocol)

Luan and Gligor [53] presented a broadcast protocol that allows toleration of the loss, duplication, reordering, and delay of messages, and network partitioning in an arbitrary network of fail-stop sites. Their protocol is based on majority-consensus decisions to commit on total ordering of received broadcast messages. Under normal operating conditions, the protocol requires three phases to complete and approximately $4N$ protocol messages, where N is the number of sites. The protocol-message overhead can be reduced if distributed among multiple-broadcast messages; thus, the heavier the broadcast traffic, the lower the overhead per broadcast message. They presented a decentralized termination protocol for abnormal operating conditions.

Their main idea consists of broadcasting the message to all sites, including the sender. Then a voting protocol is made on the commit list to ensure order of delivery, as well as to handle network partitioning and site failure. They use a quorum-based approach with a quorum of $\lceil N/2 \div K \rceil$ to handle the partitioning and to overcome the necessity of failure detections, where N is the number of sites and K is the safety margin that represents the number of failed sites or communication links that can fail during phases II and III.

Their protocols consists of two parts: *a normal-condition protocol* and a *termination protocol.* The normal-condition protocol consists of three phases: invitation, notification, and commitment. Their protocols does not assume that any detection procedure is needed for a global consensus on site or link failures.

The main drawback to this scheme is the number of protocol messages because the assumption that the broadcast messages can be used to piggyback the protocol messages may not sufficient decrease the number of messages.

2.4.5 Cristian et al.: (Atomic broadcast in real time)

One property that may be useful in a reliable broadcast protocol is specifying that delivery will occur within a specified amount of time after initiation of the protocol. This property is especially useful in real-time systems and in control applications, where a broadcast that arrives too late may not produce the desired response. If the broadcast is being made to a set of processes to instruct each to begin some action, it might also be desirable that broadcast deliveries occur within a known time interval of one another, so that their actions take place with some degree of simultaneity. The protocols described earlier make no such guarantees; they ensure that broadcasts will be eventually delivered to all non-faulty destinations, but delivery could take an arbitrarily long time.

Cristian et al. [28, 26] describe several broadcast protocols that provide real-time delivery guarantees. For such protocols, one must have timing bounds on various aspects of system behavior, for example, a bound on the time it takes for the system to schedule a process for execution, a bound on the time it takes for a message to travel from one site to another, the ability to schedule an event to occur within a certain time, and so on. Given such bounds, one can devise broadcast protocols by taking into account worst-case timing behavior.

A basic difference exists between these protocols and the ones described earlier. The earlier protocols use explicit message transfer to ensure that a broadcast has arrived at all destinations and to agree on an order of its delivery. The real-time broadcast protocols, on the other hand, use the passage of time to implicitly deduce the same information. As a result, these protocols will, in general, have a lower

communication cost. However, the latency and storage costs are based on worst-case system behavior. If the variance in the duration of system events is low and one has accurate estimates of these times, the latency and storage costs are likely to be low. On the other hand, if the variance is high, then the fact that these costs are based on worst-case behavior might make them unacceptably high.

2.4.6 Garcia-Molina and Spauster: (The propagation graph protocol)

Garcia-Molina and Spauster They presented an atomic broadcast protocol [41] that uses a graph for multicasting messages. The protocol ensures a causal-order delivery between multicasted message. It relies on a graph (propagation graph) to multicast the message and uses a distributed timestamp assignment scheme for ordering messages. This timestamp scheme assigns the timestamping task to a set of processes based on multicasting groups in order to enforce the required order. Their approach allows a smaller number of protocol messages and a faster delivery service. They allowed multiple group interactions and did not rely on any multicasting capabilities of the network. This protocol suffers from the initial cost of building the propagation graph, which would be a minor cost in the case of relatively long-lived groups. The protocol provides a set of reliability modules to handle lost messages. It also tolerates network partitions by resuming execution in the majority partition and terminating execution in the other partitions. Our protocol relies on a similar idea of using a propagation structure for message multicasting; however, we used a different approach to enforce order.

Broadcast protocol	Protocol messages	Time for delivery	Behavior under Networks partition
Chang and Maxemchuck [19]	N, $2N$	N token transfer for N resiliency	May commit messages
Cristian [28]	0	After some fixed pessimistic delay	Not applicable
Birman and Joseph [12]	$2N$	After 2 rounds message exchange	Not applicable
Melliar-Smith, Moser, and Agrawala [56]	Variable due to null ACKs	Random	May commit messages
Luan and Gligor [53]	1, $4N$	3 phase protocol	May commit messages
Garcia-Molina and Spauster [41]	N	Number of levels the message must travel	May commit messages

Table 2.1: Summary of Reliable Broadcast Protocol Performance

Chapter 3

Multicasting in Interconnected Networks

3.1 Introduction

As the demand for economic and effective sharing of resources (data and otherwise) grows, a new environment characterized by interconnected LANs that belong to different, autonomous entities has emerged. Autonomy is manifested, among other things, by different LANs that utilize different ordering criteria for multicasting.

To better serve the multicasting environment, the ideal protocol should be able to function with a small number of protocol messages, to tolerate failures (particularly network partitions), and to utilize the multicasting capabilities of the network if they exist. A bonus would be the ability to implement an intelligent routing scheme, which would decrease the number of messages generated per multicast [31].

The remainder of this chapter is organized as follows. Section 3.2 describes the environment targeted by our work and presents the problems that characterize such an environement. Section 3.3.1 describes our goals and outlines the main steps to

our approach to the solution. Section 3.4 introduces the system model and details the communication structure used to forward and multicast the messages within our system. It also defines the set of rules that are used in building this structure. Section 3.5 presents the layout of the data structures used by the protocol.

3.2 Internetwork Multicasting Problems

Existing distributed applications have been developed mostly for LAN environments. The extension of these applications to wide area networks introduces the problem of largely varied characteristics within these new environments. In order to support the migration of such applications to an internetwork environment, some features must be retained from the LAN environment [32]:

- *Group addressing.* In a LAN, a multicast packet is sent to a group address, which identifies a set of destination hosts. The sender does not need to know the membership of the group and does not need to be a member of the group. Hosts can join and leave groups at will, with no need to synchronize or negotiate with other members or with potential senders to the group. Examples of such addressing is group broadcasting, or simply broadcasting (we remind the reader that we use broadcasting and multicasting interchangeably, but we mean, in fact, group broadcasting) which can be used for such purposes as locating a resource or a server when its specific address is unknown.

- *High probability of delivery.* In a LAN, the probability that a member of a group will successfully receive a broadcast packet sent to the group is usually the same as the probability that the member will successfully receive a unicast packet sent to its individual address. Furthermore, the successful reception by every member is very high in the absence of partitioning. This property allows the designers of end-to-end reliable broadcast protocols to

assume that a small number of retransmissions of a multicast packet will result in successful delivery to all destination group members that are up and reachable.

- *Low delay.* Very little delay is imposed by LANs on the delivery of broadcast packets. This property is important for a number of broadcast applications such as distributed conferencing, parallel computing, and resource location. Also, the delay between the time when a host decides to join a group and the time the host can begin to receive packets addressed to that group, called *join latency*, is very low in a LAN environment. Low join latency is very important for certain applications, such as those that use broadcasting to communicate with migrating processes or mobile hosts, which is typical in military applications.

3.2.1 Multicasting in a Heterogeneous System

Looking back at some protocols for reliable atomic broadcasting, we can see that they utilize the LANs' broadcasting capabilities [53, 56]. These protocols potentially exhibit good performance because of the specific properties of the LAN. Not all LANs have broadcasting capabilities; therefore, the applicability of some of the protocols is restricted. Some other protocols assume a point-to-point link [12]. The generality of the assumption here puts more overhead on the protocol. Typically, a lower performance would be expected for these protocols than for the ones that use the special features of the network. We point out that no "optimum protocol" exists; for each environment, a set of good protocols exists, and we can select the one that satisfies our performance criteria.

The above discussion raises the issue of the utility of having different protocols that can be used in different parts of the network. This issue implies a software

heterogeneity in regard to the broadcasting protocols, in addition to a hardware heterogeneity in different LAN protocols and different computer configurations. The important conclusion is as follows:

> *Because the heterogeneity of the nodes cannot be ignored and the enforcement of a homogeneous environment is not practical for reasons such as autonomy, political, and environmental considerations, we must cope with the existence of this diversity.*

3.2.2 Communication Environment

Many distributed computer systems use a communication mechanism that is physically a broadcast medium, such as an Ethernet [57], a token ring [33], a token bus [76], or a packet radio system. Some of the existing standard communication protocols, however, do not allow distributed systems to use the broadcast capability of the physical communication medium but rather require all messages to be point-to-point from a single source to a single destination [59]. One reason for this practice is to ensure that the same protocol is applicable to networks of different characteristics.

We assume an environment of multi-access networks (LANs and possibly satellite networks) that are interconnected in an arbitrary topology by packet switching nodes (bridges and/or routers). Point-to-point links (both physical links such as fiber-optic circuits and virtual links such as X.25 virtual circuits) may provide additional connections between the switching nodes, or between switching nodes and isolated hosts, but almost all hosts are directly connected to LANs.

The LANs may not be similar (i.e., the capabilities of different LANs vary). Specifically, some LANs may possess multicasting capabilities; others may not. Thus, we have heterogeneous networks connected through the internet.

3.2.3 Failure Assumptions

We cannot expect our protocol to function under a no-fault assumption. Building a distributed system without considering the possibility of, for example, site failure or message loss is not practical. We must consider the failure assumptions that are stated below, especially when working with lower level protocols like the broadcast protocols that are part of the transport-layer service.

1. Sites can exhibit omission failures (i.e., they can fail to send a message when they are supposed to do so).

2. Communication links can fail at any time and can come back up at any time.

3. Messages can be lost, and delays can be arbitrarily long. Note that the loss of a message due to buffer overflow can be more easily modeled by a link failure than a site failure.

4. Network partitions can occur; the probability of occurrence is higher because of the interconnected LAN characteristics [23, 31].

Practicality dictates that our protocol take these failure characteristics into consideration.

3.2.4 Problems With Broadcasting in Heterogeneous Distributed Systems

As mentioned in Section 3.2.1, the issue of software heterogeneity must be considered. Each site must be able to select their own operating system. Because we are forced to consider the existence of distributed applications that run on different LANs, we must also consider how these applications might interact. The same problem is encountered when a distributed application spans different LANs and

has both local operations that involve groups in one LAN and global operations that involve groups in different LANs.

We are primarily interested in the tools that these distributed applications use to perform their functions. Specifically, we are interested in the *reliable broadcast protocols* that are used by these applications. Can each of these applications use the same reliable broadcasting protocol, or are they able to select a suitable broadcasting protocol for their environment? Subsection 3.2.1 discusses the necessity of allowing each site to select a protocol.

The answer to the above question must account for the possibility of having several LANs, each with a different reliable broadcast protocol (like the one presented in subsection 2.4) with different LAN protocols, such as Ethernet, token-ring, or simply point-to-point links. Our primary objective is to develop a protocol that can orchestrate the cooperation between these protocols to achieve a reliable ordered delivery service for InterLAN messages.

The issue is not to design a new broadcasting protocol; rather, our rationale is that if the site is able to choose its network protocol, then why not grant it the same freedom to choose the best reliable broadcast protocol as well. This concept goes along with that of site (network) autonomy; we think that preserving this autonomy is worthwhile because the main criteria for using one broadcast protocol over another is better performance in the local environment.

However, allowing sites to exercise autonomy in choosing both a network and broadcast protocols introduces the problem of how to broadcast on different LANs. Our approach is to develop a global protocol between the local heterogeneous broadcast protocol layer and the application layer (see Figure 3.1). The role of this global protocol would be to interface, organize, convert, and arbitrate for the local protocols.

We may ask ourselves whether the existing reliable broadcast protocols can

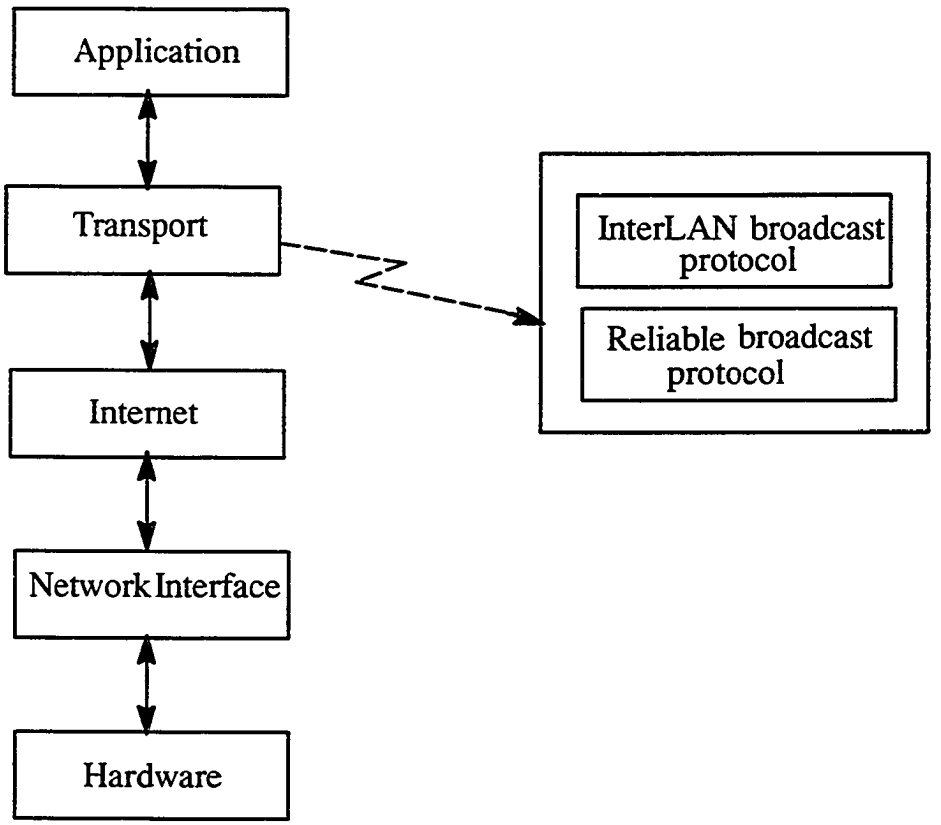


Figure 3.1: Conceptual layers of hardware and protocol software used in internet communications.

achieve this task? The answer is no. To illustrate this assertion, consider the following two cases:

- *Systems with the same broadcasting protocol*

  In this case, we have different LANs that run the same reliable broadcast protocol. The distributed application forces an interaction between these LANs because of a need for cooperation. First, if we assume that the broadcast protocol is one of the protocols that assumes some broadcasting capabilities in the underlying network, then broadcasting over different LANs is obviously not achievable because of the lack of broadcasting capability on the connections between the LANs. Second, if we assume that the broadcast protocol is one of the protocols that does not assume any broadcasting capabilities in the underlying network, then this protocol could perform the required task. However, the performance achieved by the protocol would be degraded because of both its inability to utilize the network capabilities and the inefficient use of the links between the LANs (assuming a point-to-point connection).

- *System with multiple broadcasting protocols*

  In this case, we have different LANs that run different reliable broadcast protocols. Different problems appear here mainly because of the protocol-to-protocol interactions. None of the known broadcast protocols are capable of achieving this type of interaction. Among these problems are:

  1. The absence of a standard interface that allows for this type of interaction among several heterogeneous broadcasting protocols.
  2. The existence of different local and global broadcasting groups. The ability to handle these groups spread over interconnected LANs is simply not addressed by the currently available reliable broadcast protocols.

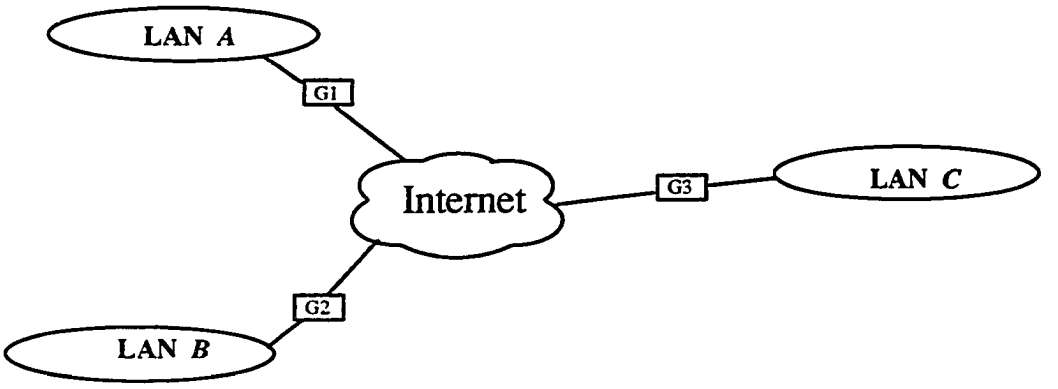


Figure 3.2: Example of three interconnected LANs.

3. The existence of gateways and routers (in the case of indirectly connected networks), which requires a special scheme to handle the protocol to allow the interaction between these interconnected LANs.

3.2.5 Case Study

Assume that we have three connected networks A, B, and C of different types interconnected with point-to-point links as shown in Figure 3.2.

Different applications are running on the networks, and each application may need to send messages to groups in other networks.

First, we must identify the types of messages that can exist in the same network, for example, network A (see Figure 3.3).

- *Local messages.* These are messages that are broadcasted within the same LAN (i.e., *local* messages for LAN A are the messages where the source and the destinations both belong to A).

- *External messages.* These messages originate from a different LAN (i.e., *external* messages are messages for which the destinations but not the source

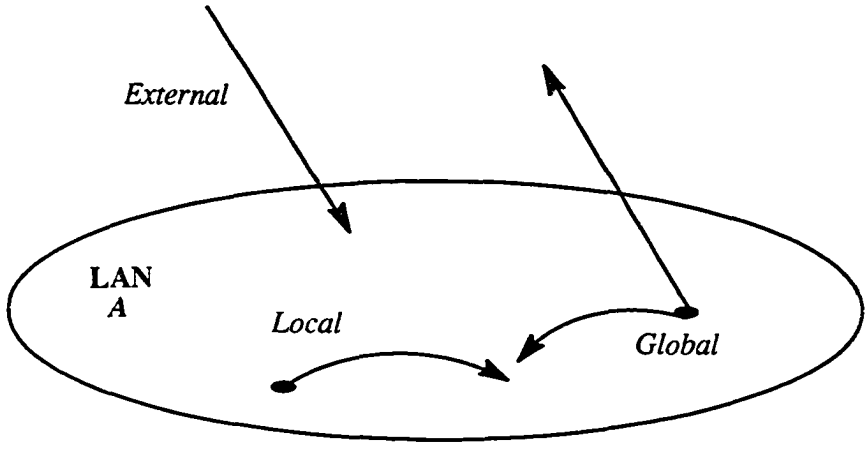


Figure 3.3: Types of messages that can exist in a network.

belong to A).

- *Global messages.* These messages originate from a LAN and have destinations in other LANs, as well as in the local LAN (i.e., *global* messages for A are messages for which the source belongs to A and the destinations belong to both A and some other networks, for example, B and C).

Several distinct cases should be considered:

- **Case 1: only receiving or sending sites**

  Assume each network has only one role in relation to the other networks. By this, we mean that the group in each network may either send or receive global messages but not both. For example, C is a receiver; A and B are senders of global messages to C.

  Assume the existence of a central node to which the *global* messages are forwarded. This node timestamps these messages and, in turn, forwards them to their destinations. This process ensures a global unique timestamp for each message.

The central timestamper will timestamp the message from A and B and send them to C; they will be locally broadcasted by C's gateway. The local broadcast protocol is assigned the task of ordering the messages. The only task performed by network C's gateway would then be to ensure the detection of lost messages sent from the central timestamper. This gateway will always broadcast the messages coming from the central timestamper in the timestamp order.

- **Case 2: both receiving and sending sites**

  What would be the case if some networks both send global and receive external and global messages? For example, assume that C sends messages to and receives messages from A and B.

  All sites that belong to C will direct the global messages to the central timestamper. The central timestamper timestamps each arriving message, which enforces a relative order among the messages coming from A, B, and C. The messages then are directed to their destinations. The message timestamp must be sent back to the message source to be used in the ordering task. An important question to be considered is whether following this timestamp is sufficient to enable the global messages to be ordered accurately.

  We have two main problems here:

  - **Ordering the global messages relative to one another.** The unique timestamp given to the global messages by the central timestamper supposedly helps to do this. Because we have a running broadcasting protocol in each network, these global messages must be delivered to the protocol (see Figure 3.1). Therefore, the local protocol is supposed to enforce this order. The question that arises is:

How can we force the local protocol to adopt the central timestamp order?

- **Ordering the external messages relative to the global and local messages.** This problem is more difficult to solve than the first one. The issue here is a twofold message-ordering problem.

  1. **Global messages relative to local messages.**
     For a particular network, global messages represent messages that are to be sent to a destination outside the network. Such messages must be ordered in relation to both local and global messages. If we assume that the global messages are simply a special type of local message, with destinations outside the local network, then the order of these messages relative to the normal *local* ones could be easily handled by the local broadcasting protocol. Global messages would then be broadcasted locally by the local broadcast protocol.
  2. **External messages relative to global messages.**
     Now that the relative order of global and local messages has been enforced, the next question is:

     How can we enforce the order between the external messages and the global messages through the local broadcasting protocol by using the timestamp given by the central timestamper?

3.3 Statement of Purpose

3.3.1 The Environment

The environment targeted by our research can be divided into two categories:

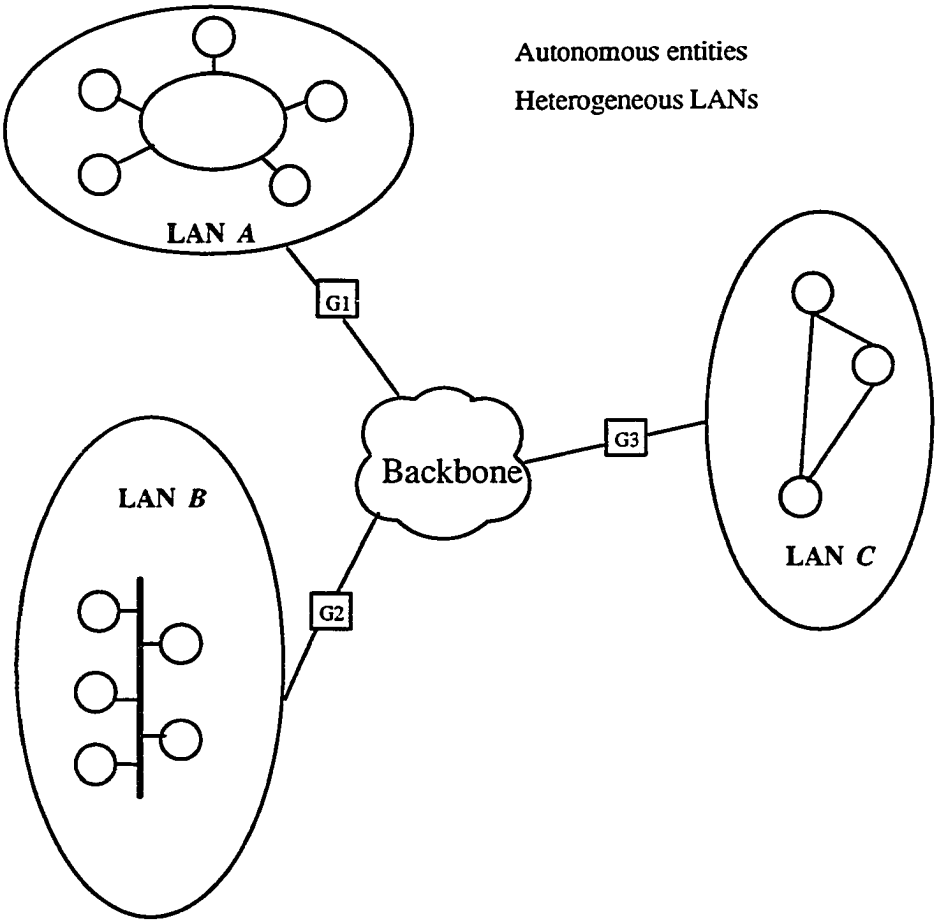


Figure 3.4: Network environment.

- *Network environment*: The targeted environment includes a set of autonomous entities that select their local reliable multicasting protocols (e.g., ABCAST, TOTAL, TOKEN), LAN topology (e.g, bus, point-to-point), and local ordering (e.g., total, causal, FIFO). A heterogeneous environment results that is characterized heterogeneity in LAN protocols and message delivery order. Figure 3.4 outlines the expected network environment.

- *Applications environment*: The targeted environment includes a set of applications that consists of different groups. For example, application Y in

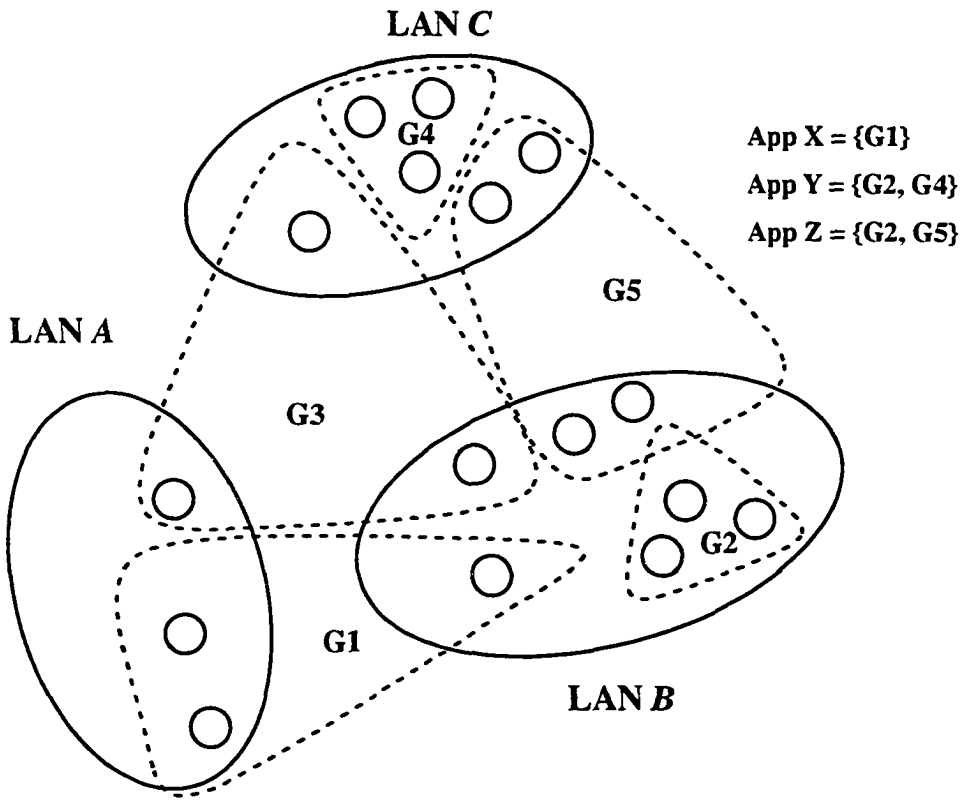


Figure 3.5: Applications environment.

Figure 3.5 contains groups G2 and G4. Each group may have members that belong to the same LAN (e.g., G2 and G4) or to different LANs (e.g., G1, G3, and G5). The same group may belong to different applications (e.g., G2). Each group has autonomy in selecting its local broadcast protocol, as well as the delivery order of the received messages based on the application need.

3.3.2 Goals

Our main goal is not only to design a global standard protocol that can support a reliable atomic broadcast service in an interconnected group of LANs, but also to achieve the utmost utilization of the underlying communication network capabilities. The protocol must take into consideration the following important areas of concern:

- The existence of different LANs, each with its own protocol and topology (LAN autonomy).
- The existence of different broadcasting protocols used in each LAN (protocol heterogeneity).
- The different multicasting and broadcasting capabilities of each network.
- The possibility of omission failure and network partitioning.
- The larger delay for internetwork messages.

The protocol must possess some, if not all, of the following features:

- It must use the features and capabilities of the underlying LANs.
- It must have a small number of protocol messages that are exchanged between LANs to decrease the overhead of the protocol and result in better performance.
- It must eliminate any type of interference in the operation of the local broadcast protocol that runs on each network.
- In the case of LANs that are not directly connected, the information available in the routing tables must be used to broadcast messages to the other networks.

- The high probability of network partitioning and global message loss must be taken into consideration.

- The required order must be enforced by all the local protocols.

A protocol that possesses all of the above features promises to be both efficient and reliable.

3.3.3 Approach to the Solution

The complexity of the problem urges us to pursue a multistep approach to the solution. This approach will allow us to better understand the interaction between the reliable broadcast protocols and our *InterLAN broadcast protocol* and the expected effects of our protocol on the total performance of the system. The four main steps that we have undertaken in our research can be summarized as follows:

1. A set of protocols will be devised that can accomodate different ordering criteria (total and causal order) which comply with the requirements mentioned previously. These protocols should have a common ordering enforcement approach to allow the easy incorporation of multiorder achievement later on.

2. A reliable multicasting protocol will be devised that allows the enforcement of multiorder between messages, based upon the recipient's group ordering requirement. This protocol will use the single-order protocols to achieve this goal. Group-driven ordering allows the removal of unnecessary ordering restrictions, which provides higher performance, delivery of multicasted messages. It also allows cooperation between groups with different ordering requirements and maintains group autonomy.

3. A scheme will be devised to allow the previously developed protocol to interact with different protocols. This step, in part, also provides multicasting

groups with autonomy in selecting their multicasting protocols and does not prevent them from interacting with other groups that rely on different multicasting protocols. Our InterLAN protocol is expected to interact with any of these protocols without any type of direct interference in their operations. This layered architecture has proven successful in the discipline of protocol design for both reliable, and unreliable network delivery services [23].

4. The reliability issues will be examined and reliability modules will be provided for the prementioned protocols to achieve resiliency to the failures assumptions mentioned earlier.

Figure 3.6 shows the development evolution of our protocol suite and the dependencies between the protocols.

3.4 The Communication Model

We propose an ordered multicasting protocol suite designed to support ordered atomic reliable multicasting across interconnected LANs. Our protocol relies on a hierarchical structure in the communication topology. This structure can be one that reflects the actual physical connections, one that is inferred by studying the group interaction, or one that is simply imposed over the message flow to honor the protocol requirements.

Members of one group can be individual processes and/or other groups. The protocol does not restrict the members of a group to the same LAN. Additionally, the protocol allows each group to determine its own ordering criterion (causal or total). Hence, our multicasting environment contains two types of groups: the *causal* groups that enforce a causal order and the *total* groups that enforce a total order. Our protocol can circulate messages that have some addressees in total groups and other addressees in causal groups and can still observe the particular

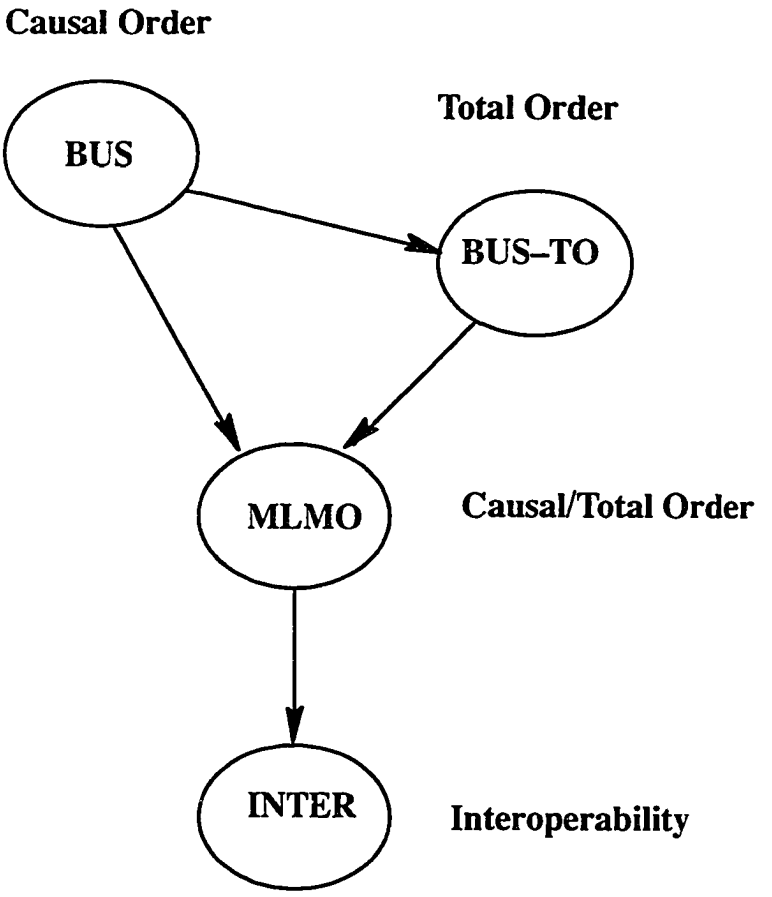


Figure 3.6: Presented protocol suite and its development dependencies.

ordering criterion for each addressee's group. Note that a given group's ordering criterion pertains to those members that are individual processes and not members that are groups because the later would by definition have their own criteria.

Our protocols assume no sequenced delivery service from the underlying communication network. This assumption is realistic because different routes can be used by messages sent from the same sender to the same recipient. For simplicity, the protocol version presented here assumes no failure. Our failure model, as well as proper addendum to the protocol for handling network partitions, message loss, and crash failures, is proposed in Chapter 9.

3.4.1 The Communication Structure

The system is composed of a set of cooperating processes $C = \{p_1, p_2, ..., p_n\}$ with disjoint memory space that uses message passing as the means for interaction. Each process p_i is identified by a ternary-tuple $< n_i, h_i, d_i >$, where $n_i \in N$ the set of LANs, $h_i \in H$ the set of hosts, and $d_i \in D$ the set of process identifiers involved in multicasting at each host. The sets N, H, and D are dynamic and are affected by the join-in and leave of processes to and from C. For simplicity, the processes in set C will be referred to as *"activity processes"*.

Elements of the Communication Structure

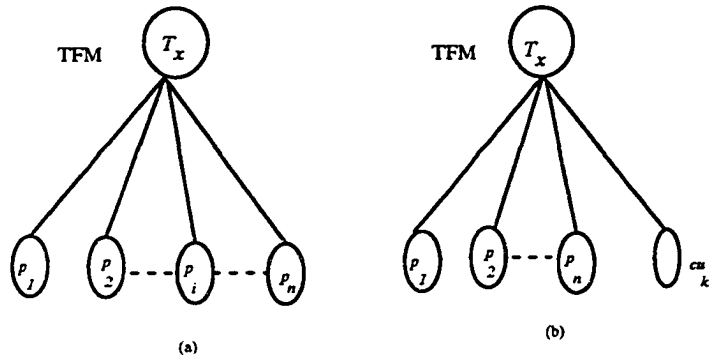
We assume that multicasting can be performed either by the multicasting capabilities of the individual networks or through point-to-point links. First, we must introduce two new terms: the *timestamping, forwarding,* and *multicasting* (TFM) process and the *communication unit (CU).* The TFM process is an independent process responsible for timestamping, forwarding, and multicasting messages. The TFM is not part of the sending or receiving set of processes; that is, it is not a member of C. The CU, in its simplest form, consists of a set of cooperating

processes and a TFM process. We can visualize each CU as a tree with two levels; the TFM process is the root, and the other members are the leaves. The cooperating processes are always the leaves of the hierarchical structure. Examples of CUs are shown in Figure 3.7. The set C is used to build a set of communication units $B = \{cu_1, cu_2, ..., cu_m\}$, where, in general, relatively high interaction occurs among members of the same CU and relatively low interaction occurs among members of different CUs. Three types of messages are distinguished by a given communication unit cu_x: *local* messages to which the sender and the destinations reside in cu_x; *global* messages for which the sender belongs to cu_x and some of the destinations belong to other CUs; and *external* messages for which the sender is not a member of cu_x but some of its destinations are.

Communication between two different CUs must be performed through the TFM that is least common to both. The least common TFM that covers all recipients of a message m is referred to as the *least common ancestor* of m and is denoted by $LCA(m)$.

The CUs are the building blocks of our communication structure. If one of the members of a CU wants to multicast a message to its CU members, it sends the message to the TFM process. The TFM process then timestamps the message and multicasts it to all the members of the CU, including the sender, or it directs the message to a higher level in the communication structure. The messages will be delivered to the receiving processes on the basis of the timestamp.

The set of communication units B can be expressed as $B = \{cu_i : \forall\ t \in cu_i$ either $t \in C$ or $t \in B$ or t *is a* $TFM\}$. Figure 3.8 shows how the structure is built. Processes p_{221}, p_{222}, and p_{223} form a communication unit cu_{22} with $\mathcal{T}_{22}$ as a TFM. Similarly, p_{2421} and p_{2422} form a communication unit cu_{242} with $\mathcal{T}_{242}$ as a TFM. A more complex structure could be formed, for example, if our communication pattern and/or topology implies the addition of cu_{242} to a higher communication



(a) Simple communication unit with cooperative processes as members.

(b) Communication unit with CU as member.

Figure 3.7: Protocol communication units.

structure; then, cu_{24} is formed by having p_{241}, p_{243}, and cu_{242} with $\mathcal{T}_{24}$ as a TFM. Similarly, p_{21}, p_{23}, cu_{22}, and cu_{24} must be linked together in a communication unit cu_2 with $\mathcal{T}_2$ as a TFM. As we can see, this configuration forms a tree structure.

Any CU is interfaced to other CUs through its designated TFM process, which controls the message passing in the CU to and from the other CUs. The communication between two different CUs must be performed through the least common TFM to both. For example, if a message must be multicasted from p_{221} to cu_{22} and cu_{242}, then this message must reach $\mathcal{T}_2$, which is the least common TFM that has both cu_{22} and cu_{242} within its tree structure. Of course, the message, on its way from p_{221} to cu_{242}, will pass all TFMs in the hierarchical structure until cu_{242} (i.e., it will pass by $\mathcal{T}_{22}, \mathcal{T}_2, \mathcal{T}_{24}, \mathcal{T}_{242}$).

If $D(m)$ is the set of destinations of the message m, then $LCA(m)$ can be defined as follows: $LCA(m)$ is a TFM process $\mathcal{T}_x$ such that $D(m) \subseteq cu_x$ and $\forall\ cu_y$ if $D(m) \subseteq cu_y$ then $cu_x \subseteq cu_y$. Therefore, $LCA(m)$ is the TFM of the smallest CU that contains all destinations of message m.

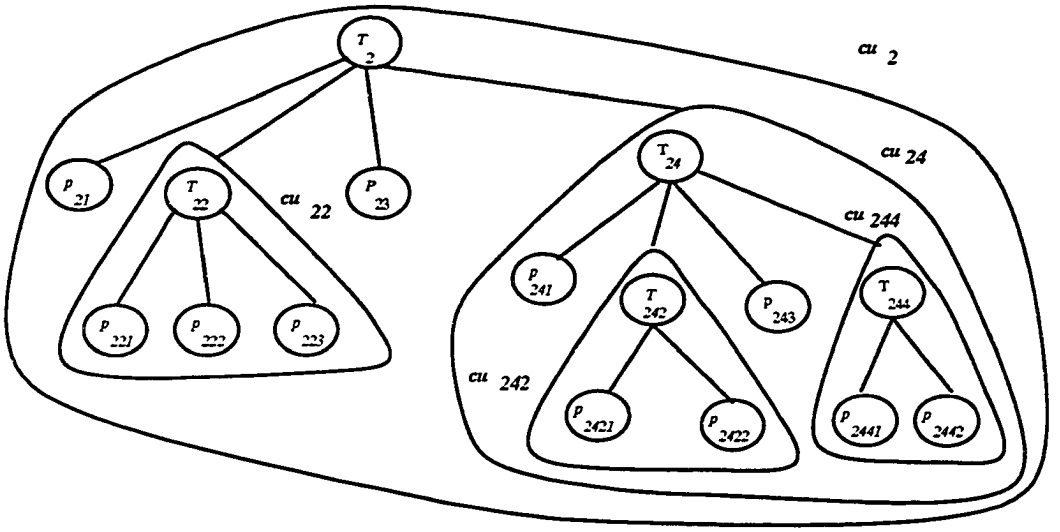


Figure 3.8: Communication structure with multilevel communication units.

Formally, the least common ancestor of two communication units cu_x and cu_y, denoted by $LCA(cu_x, cu_y)$, is a TFM process $\mathcal{T}_p$ such that $\mathcal{T}_x \prec \mathcal{T}_p \wedge \mathcal{T}_y \prec \mathcal{T}_p \wedge \forall \mathcal{T}_z$, where $\mathcal{T}_x$ and $\mathcal{T}_y \prec \mathcal{T}_z$, $\mathcal{T}_p \preceq \mathcal{T}_z$ (where $\prec$ denotes a precedence relation such that $X \prec Y$ means that Y falls along the path between X and the root of the tree).

For clarification and ease of presentation, we have assumed that the TFM process is an independent process; however, the functions of the TFM process can be performed by one of the cooperating processes of its CU.

Communication Unit Formation Rules

Two rules are necessary for mapping this structure to our multicasting and group addressing: the *Subdivision* rule and the *Enclosure* rule.

- *The Subdivision Rule.* A communication unit may not contain a subset of another CU; it must contain either the entire CU or none of it (i.e., for all

cu_x and cu_y either $cu_x \cap cu_y = \phi$, $cu_x \cap cu_y = cu_x$, or $cu_x \cap cu_y = cu_y$).

- *The Enclosure Rule.* If two communication units cu_x and cu_y are contained in another communication unit cu_z, then cu_z must contain LCA(cu_x, cu_y), which is the least common ancestor of cu_x and cu_y. For example, if cu_{242} and cu_{22} in Figure 3.8 are to be in a CU, then $\mathcal{T}_2$ must be in that group because $\mathcal{T}_2 = LCA(cu_{242}, cu_{22})$.

If cu_i is specified as a member of another cu_j, then the TFM of j ($\mathcal{T}_j$) would control the delivery of messages to $\mathcal{T}_i$; on the other hand, message delivery to a member of cu_i remains under the control of $\mathcal{T}_i$. According to our communication structure, a message m must specify a set of CUs as its recipients.

The protocols introduced in this thesis depend on the hierarchical structure imposed on the communication between the processes in the group. An important issue is raised in constructing the hierarchical structure in regard to the CU insertion in the total structure and its effect on the performance. For the example in Figure 3.8, if cu_{24} and cu_{22} have heavy message traffic between them, then a TFM process common to both of them that does not involve the whole group would be the most efficient. A new construction is shown in Figure 3.9.

This hierarchical structure can be built with an algorithm that optimizes the message delay; the frequency of communication between processes and the underlying communication topology are taken into consideration. The algorithm must take care of subgroup addressing patterns while it constructs this hierarchy. The best scheme for building the CU depends on the locality of networks (i.e., the process in the same LAN would belong to the same CU). We have developed an algorithm to build similar structure [77]. This algorithm optimizes the number of levels in the tree and the number of nodes the message must pass by to get to its TFM, by gathering the communication groups into a smaller number of CUs that are closely located in the structure. The algorithm introduces a structure that

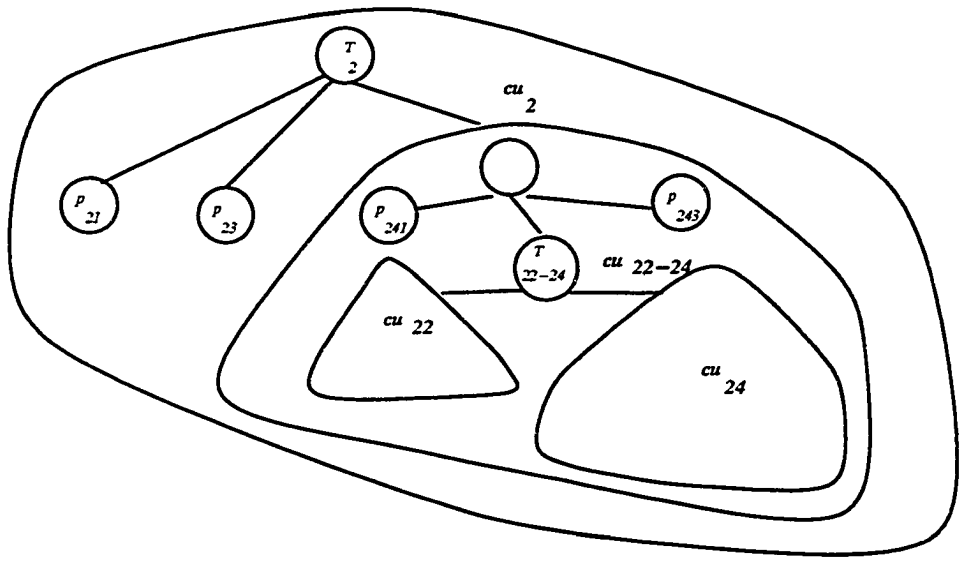


Figure 3.9: Reshaped communication structure to increase performance.

minimizes the number of levels the message must go through to reach its LCA. Also, it accommodates intersecting groups in a similar structure and minimizes the number of levels the message must pass through before delivery.

The protocol performance improves if the TFM process runs at the gateway because the traffic going outside the LAN must pass by the gateway. Thus, the protocol will not add extra hops for timestamping. Note that group addressing is the main factor in the CU construction scheme because of its restriction in subgroup formation. An algorithm that builds this communication structure by considering the cost and frequency of communication between processes must be devised. The dynamic characteristics of the communication need to be considered; this will require a dynamic algorithm that can reconfigure the CU membership

during execution.

This structure will benefit many applications that use a hierarchical communication structure, for example, the communication in a corporation between the branches and the headquarters. Another example from the Computer Supporting Cooperative Work (CSCW) domain is group writing [42, 3], with a running session for writing a book and subsessions for chapter writing and section writing. In a third example from the computer communication domain, the interaction between the sites on the internet is directed in a similar hierarchical structure to deliver messages between LANs through gateways [22]. We believe that from performance perspective, subgroup structure and hierarchical communication is still reasonable in the case of an unclustered communication between members of a group.

3.5 Protocol Data Structures

The data structures used by the protocol to enforce order are composed of queues and other structures for handling timestamps. These structures are shown in Figure 3.10 and are described below. Queues are used to hold the messages before their delivery to achieve the specified order. We need the following four queues:

- *Deliver Queue (DQ).* The messages are buffered for delivery to the process.
- *Local Wait Queue (LWQ).* The local messages are kept waiting for those messages to arrive that are assigned smaller timestamps if any are missing.
- *Global Wait Queue (GWQ).* The global messages are kept waiting for those messages to arrive that are assigned smaller timestamps if any are missing.
- *Out-of-Order Queue (OOQ).* The messages that arrive out of order from the same process are kept until the late or lost messages arrive. With a TFM process, the messages kept in this queue have not been assigned a timestamp from the receiving TFM yet.

The GWQ is mainly added to our structure to prevent delay and extra processing overhead for the local messages because of the existence of global messages in the queues.

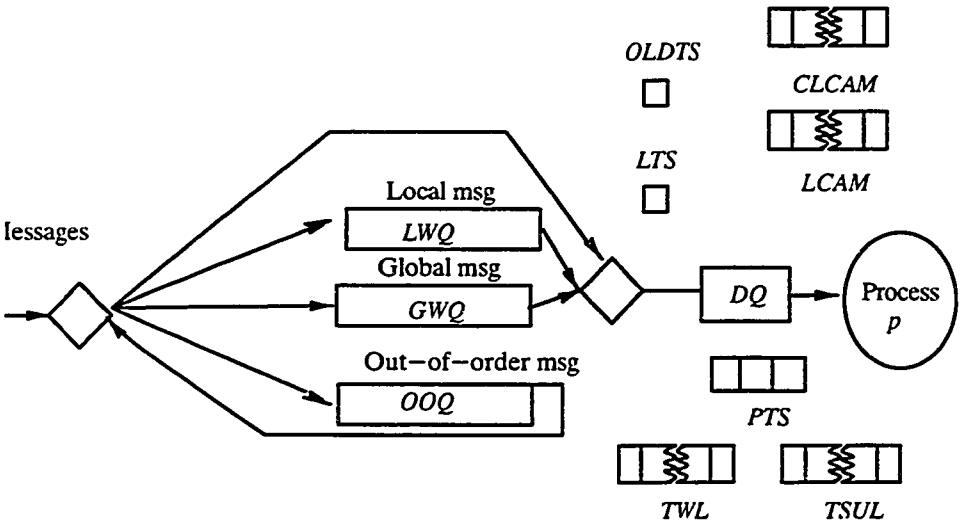


Figure 3.10: Message flow at a process that shows the data structure used.

The timestamping mechanism is responsible for assigning timestamps to the messages and for ensuring the correct order of these assignments by keeping information about other TFM timestamps. We assume in the following definition that p_x is an activity process or a TFM process.

- $MTS_{m_x}[\,]$. A timestamp vector that accompanies the message m_x and carries the timestamp assigned by the sender and the different TFM processes it passes by as it moves along the hierarchy toward $LCA(m_x)$.
- $PTS_{p_x}[\,]$. A local timestamp vector used by the process p_x to keep track of the timestamp of the last message delivered from the TFMs.

- LTS_{p_x}. A local timestamp variable used by the process p_x to stamp the messages sent or passing by it. The LTS_{p_x} always contains the last used timestamp value.
- $OLDTS_{p_x}$. A local timestamp variable used by the process p_x to store the timestamp of the last message sent from p_x to the TFM $\mathcal{T}_x$.

The following data structures are specific to the TFM processes:

- $TWL_{\mathcal{T}_x}$. For a given *total communication unit* TFM (TCU-TFM) $\mathcal{T}_x$, $TWL_{\mathcal{T}_x}$ contains the messages that passed by $\mathcal{T}_x$ in a *One-Way* (OW) path before they gained their LCA timestamp. The TCU-TFM and the OW path are defined in Section 6.2.
- $LCAM_{\mathcal{T}_x}$ and $CLCAM_{\mathcal{T}_x}$. The *least common ancestor message* ($LCAM_{\mathcal{T}_x}$) contains all the messages for which $\mathcal{T}_x$ acts as the LCA. The $LCAM_{\mathcal{T}_x}$ is a temporary list where messages reside until they are committed for delivery. The messages in $LCAM_{\mathcal{T}_x}$ are waiting for a message that has a smaller timestamp from $\mathcal{T}_x$. The *committed least common ancestor message* list ($CLCAM_{\mathcal{T}_x}$) contains the part of these messages that have been committed by $\mathcal{T}_x$ for delivery; the $CLCAM_{\mathcal{T}_x}$ is piggybacked with any message that is traversing the OW path.
- $TSUL_{\mathcal{T}_x}$. The *timestamp updater list* ($TSUL_{\mathcal{T}_x}$) is maintained by each TCU-TFM. Any message directed down the hierarchy along any of its one-way type $\mathcal{A}$ ($\mathcal{OWA}$) paths (see Sections 5.2 and 6.2) is assigned a timestamp by $\mathcal{T}_x$. The message, after it is timestamped, adds an entry to $TSUL_{\mathcal{T}_x}$. This list is used by the messages that pass $\mathcal{T}_x$ in their *two-way* (TW) or one-way type $\mathcal{B}$ ($\mathcal{OWB}$) paths to adjust $PTS_{p_x}[\,]$.

3.6 Conclusion

The protocols introduced in this thesis depend on the hierarchical structure that is imposed on the communication between the processes in the group. This hierarchical structure can be built with an algorithm that optimizes the message delay; the frequency of communication between processes and the underlying communication topology are taken into consideration. This optimization algorithm depends on the network topology in order to decrease the number of circulating messages. The algorithm must account for subgroup addressing patterns while it constructs this hierarchy. The best scheme for building the CU depends on the locality of the networks (i.e., the process in the same LAN would belong to the same CU). The protocol performance improves if the TFM process runs at the gateway. This structure will benefit many applications that use a hierarchical communication structure, for example, communication in a corporation between the branch offices and the headquarters office. We believe that this subgroup structure and hierarchical communication are still reasonable from the performance perspective in the case of an unclustered communication between the members of a group. The protocols also allow only part of the cooperating group to be addressed by creating a set of cooperating subgroups. This results in reducing the traffic over the network because unnecessary messages sent to inactive participants are eliminated.

Chapter 4

BUS: Bottom-Up Stamping Protocol

4.1 Introduction

The Bottom-Up Stamping (BUS) protocol is a reliable multicast protocol that uses the hierarchical structure to achieve a causal order between multicasted messages. In this chapter, we assume a reliable system with no site or link failure and no loss of messages (Chapter 9 deals with these issues). We also assume that a message m is directed to all processes under $LCA(m)$. Chapter 3 as well as the glossary give some details for these terms definitions. As stated earlier in Chapter 3, all messages are directed to the TFM processes of the CUs, which multicast the messages to the members and direct them to a higher level TFM process. The protocol will forward all messages sent from a node to its local TFM for timestamping. If the message destinations are local to its CU (i.e., it is a local message), then the local TFM will multicast the message to the CU members after it has been timestamped. This process allows the TFM to be the unique timestamper for the CU messages; therefore, an order among the messages that

honors the rules presented in Section 2.3.1 can be achieved. If some of the message destinations reside outside the CU (i.e., it is a global message), then the TFM will forward the message up the hierarchy to the higher level TFM after it has been multicasted. The higher level TFM, upon receiving the message, will perform the same procedure until the message reaches its LCA. On its way up, the message is timestamped by each TFM it passes. The members of the CU order the message by using both the message and the node timestamp vectors. All sites deliver the messages that are multicasted by the TFM based on the message timestamp vector, which is described later.

The protocol concept depends on forcing the order through the TFM processes necessary to multicast a message. Assume that a session is running with a communication structure, as in Figure 4.1. Suppose a message m_1 is sent from p_{221} to its CU members (p_{222}, p_{223}). This message will be timestamped and forwarded to $\mathcal{T}_{22}$ (the TFM of p_{221}), which will schedule it for multicasting after it is timestamped. The timestamping is performed in message sending order from p_{221}; if out-of-order messages are detected, then the LWQ holds the message. This timestamp will enforce an order for m_1 among the messages sent from its group members; then m_1 is multicasted to cu_{22}. If m_1 was originally directed to cu_2, then the message will be sent from $\mathcal{T}_{22}$ to $\mathcal{T}_2$, where it will be timestamped and multicasted to cu_2 members. (The path from which the message is traversing will be filtered out from the destinations when multicasted from $\mathcal{T}_2$).

This protocol (as it will be shown later) will honor the causal order outlined in Lamport's rules. We believe that this protocol has wide applicability because most applications in a distributed system (especially with site autonomy) prefer the causal order; the causal order dictates a less restricted order, which results in a faster delivery.

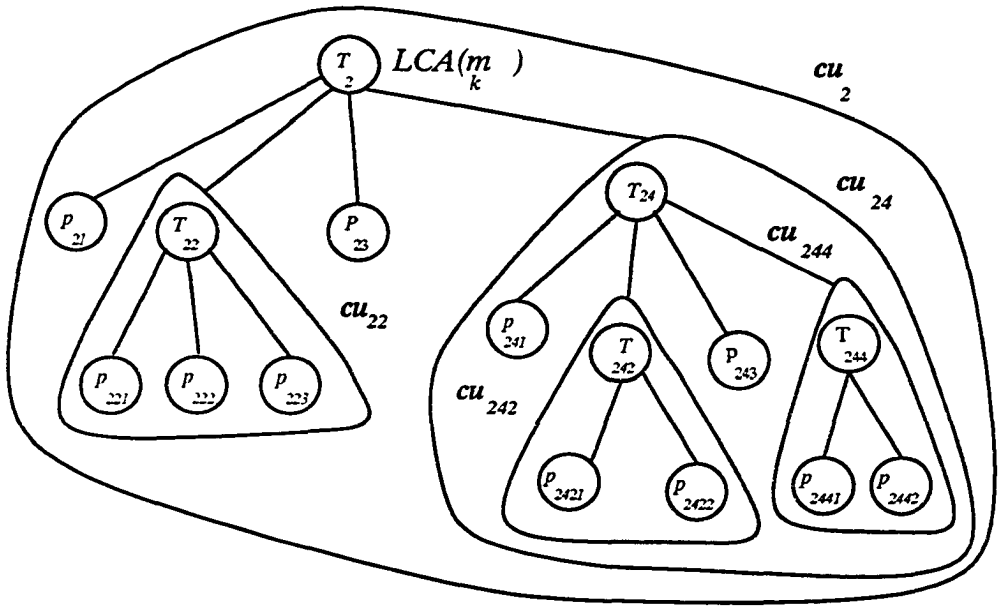


Figure 4.1: Communication structure for BUS protocol.

4.2 BUS Protocol Description

The ordering guaranteed by our protocol honors the *causal* order described in Section 2.3.1. A description of how the protocol works follows. A process (p_{iy}) sends a message m_k to its TFM ($\mathcal{T}_i$). The message needs a two-entry timestamp vector (MTS_{m_k}) to achieve the causal order. The first entry in the vector ($MTS_{m_k}[0]$) is responsible for ensuring the ordered delivery of the messages from the sender p_{iy} to its TFM $\mathcal{T}_i$. This entry contains a copy of the timestamp assigned to the last message sent to $\mathcal{T}_i$. The second entry ($MTS_k[1]$) contains a copy of the timestamp assigned to m_k by p_{iy}. After the timestamp values are assigned to MTS_{m_k}, m_k is directed to $\mathcal{T}_i$.

At $\mathcal{T}_i$, if no messages that are out of order are present, then $PTS_{\mathcal{T}_i}[p_{iy}]$ and $MTS_{m_k}[0]$ should match because $PTS_{\mathcal{T}_i}[p_{iy}]$ has not been updated since the last message from p_{iy} before m_k. The TFM $\mathcal{T}_i$ will update $PTS_{\mathcal{T}_i}[p_{iy}]$ with $MTS_{m_k}[1]$,

and then $\mathcal{T}_i$ will timestamp m_k and add the new timestamp to $MTS_{m_k}[1]$. If the message needs to go up, $\mathcal{T}_i$ will adjust $MTS_{m_k}[0]$ and $OLDTS_{\mathcal{T}_i}$ (see Section 4.3.4 for a description of the OLDTS functionality) then forward m_k to the higher level TFM (see steps 17 through 21 in procedure 4.3). If $\mathcal{T}_i$ is $LCA(m_k)$, then $\mathcal{T}_i$ multicasts m_k to its members and then checks the LWQ for any messages that are eligible for timestamping (see steps 3 through 7 in procedure 4.2). For any such messages, $\mathcal{T}_i$ will repeat the same steps described previously. However, if m_k arrives at $\mathcal{T}_i$ and delayed messages are present (lost messages trigger a similar action and are discussed in Chapter 9), m_k will be admitted to LWQ to wait for these messages to arrive.

For a message moving down the hierarchy from $\mathcal{T}_x$ to $\mathcal{T}_i$, messages that are lost or out of order are detected by comparing $MTS_{m_k}[1]$ and $PTS_{\mathcal{T}_i}[\mathcal{T}_x]$ (see steps 1 through 11 in procedure 4.3). Because $MTS_{m_k}[1]$ carries the timestamp assigned to m_k from $\mathcal{T}_x$ and $PTS_{\mathcal{T}_i}[\mathcal{T}_x]$ carries the last message timestamp delivered from $\mathcal{T}_x$, if $MTS_{m_k}[1]$ does not follow $PTS_{\mathcal{T}_i}[\mathcal{T}_x]$ in timestamp order, then some messages are delayed. If this is the case, m_k will be admitted to the LWQ, where it will wait for the delayed message to arrive (see procedure 4.2).

At one of the receiving sites, p_{jx} (with $\mathcal{T}_j$ as a TFM) will check m_k timestamp ($MTS_{m_k}[1]$) with $PTS_{p_{jx}}[\mathcal{T}_j]$ for out-of-sequence messages. If m_k is not in sequence, it is admitted to p_{jx} LWQ. If it is in sequence, then p_{jx} will adjust $PTS_{p_{jx}}[\mathcal{T}_j]$ with the value in $MTS_{m_k}[1]$ and m_k will be admitted to DQ of p_{jx} to be buffered for delivery (see procedure 4.5).

The communication structure and the order enforced on the messages in the links between the directly connected TFMs in conjunction with the message timestamp vector are enough to achieve the required order.

4.3 BUS Protocol Outline

Three types of procedures handle the messages: the *Sender*, the *Receiver*, and the *TFM* procedures.

4.3.1 Sender

Let m_k be the message sent from p_{iy} of cu_i with $\mathcal{T}_i$ as its TFM process. The sender p_{iy} performs the steps shown in procedure 4.1 to send message m_k.

SENDER$(m_k)\{$

1. $MTS_{m_k}[0] := LTS_{p_{iy}}$
2. *Increment* $LTS_{p_{iy}}$
3. $MTS_{m_k}[1] := LTS_{p_{iy}}$
4. *Send message to* $\mathcal{T}_i$

$\}$

Procedure 4.1 (Sender)

Note here that $\mathcal{T}_i$ (the TFM of p_{iy}) can detect missing messages from p_{iy} through $MTS_{m_k}[1]$ and $PTS_{\mathcal{T}_i}[p_{iy}]$.

4.3.2 TFM

Let $\mathcal{T}_x$ be any TFM in the message path from its sender p_{iy} to any of its destinations p_{jx}, and $\mathcal{L} = LCA(cu_i, cu_j)$. In addition, let *Direct Sender* represent the process by which the message m_k is forwarded to $\mathcal{T}_x$. The TFM $\mathcal{T}_x$ performs the steps shown in procedures 4.2 and 4.3 to forward a message m_k.

$OLDTS_{\mathcal{T}_x}$ is used to provide a reliable delivery scheme between two consecutive level processes while the message is moving up in the hierarchy (see Section 4.3.4

TFM(m_k) {

```
1.    if m_k has not passed by T_x before →
2.        if MULTICASTABLE(m_k) →
3.            For each m_t ∈ LWQ,
4.                {
5.                   if MULTICASTABLE(m_t) → remove m_t from LWQ
6.                   Otherwise return
7.                }
8.        Otherwise
9.            Admit m_k to LWQ
10.       fi
11.   Otherwise
12.       Discard m_k
13.   fi
```

}

Procedure 4.2 (TFM)

for a description of the OLDTS functionality).

4.3.3 Receiver

Let m_k be the message that is received by process p_{jx} of cu_j with $\mathcal{T}_j$ as its TFM process. The receiver p_{jx} performs the steps shown in procedures 4.4 and 4.5 to receive the message m_k.

4.3.4 Remarks

The *GWQ* is not used in this protocol because the protocol views both *local* and *global* messages as *local* messages.

MULTICASTABLE(m_k) {

```
1.    if m_k is on its way down →
2.      if MTS_{m_k}[1] = PTS_{T_x}[Direct Sender] + 1 →
3.          Increment PTS_{T_x}[Direct Sender]
4.          Increment LTS_{T_x}
5.          MTS_{m_k}[1] := LTS_{T_x}
6.          Multicast m_k to cu_x
7.          return true
8.      Otherwise
9.          return false
10.     fi
11.   fi
12.   if m_k is on its way up →
13.     if MTS_{m_k}[0] = PTS_{T_x}[Direct Sender] →
14.         PTS_{T_x}[Direct Sender] := MTS_{m_k}[1]
15.         Increment LTS_{T_x}
16.         MTS_{m_k}[1] := LTS_{T_x}
17.         if T_x ≠ LCA(m_k) →
18.             MTS_{m_k}[0] := OLDTS_{T_x}
19.             OLDTS_{T_x} := LTS_{T_x}
20.             Forward m_k up
21.         fi
22.         Multicast m_k to cu_x
23.         return true
24.     Otherwise
25.         return false
26.     fi
27.   fi
```

}

Procedure 4.3 (Multicastable)

RECEIVE(m_k) {

1. **if** *DELIVERABLE*$(m_k) \rightarrow$
2. *For each* $m_t \in LWQ$,
3. {
4. **if** *DELIVERABLE*$(m_t) \rightarrow$ *remove* m_t *from LWQ*
5. **Otherwise** *return*
6. }
7. **Otherwise**
8. *Admit* m_k *to LWQ*
9. **fi**

}

Procedure 4.4 (Receiver)

DELIVERABLE(m_k) {

1. **if** $(MTS_{m_k}[1] = PTS_{p_{jx}}[T_j] + 1) \rightarrow$
2. *Increment* $PTS_{p_{jx}}[T_j]$
3. *Admit* m_k *to DQ*
4. *return true*
5. **Otherwise**
6. *return false*
7. **fi**

}

Procedure 4.5 (Deliverable)

$OLDTS_{\mathcal{T}_x}$ is used to provide a reliable delivery scheme between two consecutive level processes while the message is moving up the hierarchy. To clarify the functionality of $OLDTS_{\mathcal{T}_x}$, assume that a message m_k is sent from $\mathcal{T}_x$ to $\mathcal{T}_w$ (TFM of $\mathcal{T}_x$). The message m_k is timestamped at $\mathcal{T}_x$ before it is forwarded to $\mathcal{T}_w$ with $LTS_{\mathcal{T}_x}$. This timestamp given at $\mathcal{T}_x$ and assigned to $MTS_{m_k}[1]$ is used at $\mathcal{T}_w$ to be compared with $PTS_{\mathcal{T}_w}[\mathcal{T}_x]$ for message ordering. So that this comparison is useful, all the messages timestamped at $\mathcal{T}_x$ are forwarded to $\mathcal{T}_w$ to put $PTS_{\mathcal{T}_w}[\mathcal{T}_x]$ in sequence with $LTS_{\mathcal{T}_x}$. Because the message that has $\mathcal{T}_x$ as its LCA will not be forwarded to $\mathcal{T}_w$, $PTS_{\mathcal{T}_w}[\mathcal{T}_x]$ will be missing these messages, which indicates that $LTS_{\mathcal{T}_x}$ cannot be used for this comparison. This problem forced us to introduce $OLDTS$ as a timestamp variable at each TFM process to retain the timestamp of the last message that was forwarded to the higher level TFM from this process. When a message m_k is forwarded up in the hierarchy, the first entry in the vector MTS_{m_k} carries a copy of $OLDTS$ (see steps 18 and 19 in procedure 4.3). This entry is responsible for ensuring the ordered delivery of m_k from the sender $\mathcal{T}_x$ to its TFM $\mathcal{T}_w$. The value assigned to $MTS_{m_k}[0]$ should be in sequence with $PTS_{\mathcal{T}_w}[\mathcal{T}_x]$ unless a delayed message exists (see step 13 in procedure 4.3). Then, $OLDTS_{\mathcal{T}_x}$ is adjusted by assigning the timestamp value given to m_k by $\mathcal{T}_x$.

No assumptions are made in regard to the multicasting capabilities of the network. However, the implementation of the protocol assumes that a CU could reside on a set of LANs and that network multicasting is used on each of these LANs if it exists. Messages are forwarded between LANs that have a point-to-point link. The receiving LAN gateway will distribute the message to the LAN members. The protocols also allow part of the cooperating group to be addressed by creating a set of cooperating subgroups, which results in reduction of the traffic over the network because the unnecessary messages sent to inactive participants are eliminated.

The protocol delay time for the message to reach a specific destination is pro-

portional to the number of levels the message must pass to reach this destination. This feature makes the protocol appealing because the delay time is predictable; this feature is a required feature in real-time multicasting [28]. The protocols also have some features that reduce the likelihood of message loss during message navigation in the communication structure, as will be discussed in Chapter 9.

4.4 BUS Protocol Correctness

For the purpose of this work, assume a reliable environment with no failure or loss of messages. This assumption does not weaken our proof because this environment is achievable through a reliability procedure developed for the protocol and explained later. We must prove that the protocol guarantees a causal order for message delivery. To do this, we must show that the protocol adheres to rules 1 and 2 described in Section 2.3.1.

- Rule 1:

  Assume that a CU (cu_i) exists with a set of cooperating processes $\{p_{i1}, p_{i2}, p_{i3}, p_{i4}, ..., p_{in}\}$; $\mathcal{T}_i$ is the TFM of cu_i. Also assume that we have two messages m_1 and m_2 sent from p_{iy}. We must show that at all receiving processes m_1 will be received before m_2. Two messages m_1 and m_2 are sent from p_{iy} of cu_i, where $m_1 \prec m_2$ is directed to $cu_{x_{x \neq i}}$; we know that m_2 is sent after m_1 with no message loss and that both are timestamped at the original site ($MTS_{m_1}[1] < MTS_{m_2}[1]$). If m_2 arrives at $\mathcal{T}_i$ before m_1, then it will be queued in the LWQ until m_1 arrives. Because m_1 will be admitted to $\mathcal{T}_i$ first, it will be timestamped before m_2. Then $\mathcal{T}_i$ will multicast just m_1 and then m_2 to the CU and forward both of them up the hierarchy to the higher TFM. For the copies of the messages going out of $\mathcal{T}_i$, $MTS_{m_1}[1] < MTS_{m_2}[1]$, which means that m_1 will be delivered before m_2 to the members of the CU.

Because the TFM will deliver m_1 before m_2, if the message is to be sent to a higher CU, then m_1 will be sent before m_2. Similarly, message m_1 will be timestamped at the higher CUs (cu_x) before m_2, even if they are received in a different order. The process indicates that $MTS_{m_1}[1] < MTS_{m_2}[1]$ at $\mathcal{T}_x$. As a result, m_1 will be delivered before m_2 at all members of cu_x. We can easily show that at all sites to which both m_1 and m_2 are directed, m_1 will be delivered before m_2.

- Rule 2:

  Assume that $p_{iy} \varepsilon cu_i$ sends message m_1 to cu_j. Assume that the process $p_{jx} \varepsilon cu_j$ (after it receives m_1) sends m_2 to cu_t, where cu_t is also a destination of m_1. To conform to Lamport's second rule, we must prove that m_1 is delivered before m_2 at all common destinations. Because m_1 is received at p_{jx} before p_{jx} sends m_2, m_1 is received and timestamped at $\mathcal{T}_j$ (the TFM of cu_j) before it is sent to p_{jx}. When p_{jx} sends m_2, it is directed to the TFM of cu_j, where it is timestamped, so that $MTS_{m_1}[1] < MTS_{m_2}[1]$. Because the delivery of the messages respects the timestamp order and because no messages are lost, at all members of cu_j m_1 will be delivered before m_2. Because cu_t is a destination for both m_1 and m_2, a CU (cu_m) exists that contains the LCA of cu_i, cu_j, and cu_t (e.g., $\mathcal{L}_m$). Also assume that the LCA of both cu_j and cu_i is $\mathcal{L}_n$, and $\mathcal{L}_m \geq \mathcal{L}_n$ is always true.

  Because m_1 has passed $\mathcal{L}_n$ to get to p_{jx}, m_1 passes $\mathcal{L}_n$ before p_{jx} sends m_2. This order implies that $MTS_{m_1}[1] < MTS_{m_2}[1]$ at $\mathcal{L}_n$. As a result, m_1 will be sent to the higher level before m_2, which indicates that $MTS_{m_1}[1] < MTS_{m2}[1]$ at $\mathcal{T}_z$, where $\mathcal{T}_z$ is any TFM such that $\mathcal{L}_n \prec \mathcal{T}_z \prec \mathcal{L}_m$. Similarly, we can show that m_1 will be delivered before m_2 at all higher CUs until $\mathcal{L}_m$. As they arrive at $\mathcal{L}_m$, both m_1 and m_2 are multicasted down the tree. Similarly, upon arrival at $\mathcal{T}_t$, $MTS_{m_1}[1] < MTS_{m_2}[1]$, which implies that m_1

will be delivered before m_2 at cu_t.

If rules 1 and 2 are both satisfied, then the protocol is guaranteed to ensure a partial order among the messages.

4.5 Conclusion

In this chapter, the Bottom-Up Stamping (BUS) protocol, which is a reliable ordered multicasting protocol, is presented. The BUS protocol ensures a causal order among multicasted messages. The protocol depends on forcing the communication between the processes to follow a certain hierarchical communication structure. The knowledge of this structure allows the efficient multicasting of messages. This protocol is useful for many distributed applications that do not require total order.

The BUS protocol encounters an initial overhead for building the communication structure that is necessary for the functionality of the protocols. However, it still achieves a better performance over many existing multicasting protocols, as shown in Chapter 8. The improved performance is due to the smaller storage requirement and the low communication overhead necessary for the protocol. Also, because the hierarchical structure used can be mapped to the communication topology that the message uses on the internet, no extra protocol messages are necessary to achieve ordered delivery. Also, the use of the CU hierarchy in multicasting decreases the number of physical messages sent on the internet. The protocols assume that the messages have all the CUs under their LCAs in their destinations. The problem with this assumption is the fact that some CUs under LCA(m) will receive m; however, these CUs are not targeted by m. This effect can be diminished if the group memberships are taken into consideration when the structure is built, as described in [77].

Chapter 5

BUS-TO: Bottom-Up Stamping Protocol (Total-Order Version)

5.1 Introduction

The BUS-TO protocol is a reliable multicast protocol that uses the hierarchical structure to achieve a total order between multicasted messages. The total order we adopt in our research is the total order that honors the potential causality property. As in Chapter 4, we assume a reliable system with no site or link failure and no loss of messages (Chapter 9 will deal with these issues). We also assume in this chapter that a message m is directed to all processes under $LCA(m)$; this assumption will be relaxed by the end of the chapter (see Chapter 3 and the glossary for term definitions). As stated earlier in Chapter 3, all messages are directed through the TFM processes of the CUs, which direct it to a higher level TFM process or multicast it to its members. The protocol will forward all messages sent from a node to its local TFM to be timestamped. If the message destinations are local to the CU (i.e., it is a *local message*), then the local TFM (after the message has been timestamped) will multicast it to the CU members.

This process allows the TFM to be the unique timestamper for the CU messages; therefore, an order between the messages that meets the rules presented in Section 2.3.1 can be achieved. If some of the message destinations reside outside the CU (i.e., it is a *global message*), then the TFM will forward the message to the higher level TFM. The higher level TFM, upon receipt of the message, will perform the same procedure until the message reaches its LCA. Once the message reaches its LCA, it will be multicasted down the hierarchy to all its destinations. On its way up, the message is timestamped by each TFM it passes but is not multicasted by these TFMs in this phase. The members of the CU order the message with both the message vector and the node timestamp vector.

Assume that a session is running with a communication structure as in Figure 5.1. Suppose that a message m_1 is sent from p_{221} to its CU members (p_{222}, p_{223}). This message will be timestamped and then forwarded to $\mathcal{T}_{22}$ (the TFM of p_{221}), which schedules the message for multicasting after it has been timestamped. The timestamping is performed in the order the message is sent from p_{221}; if out-of-order messages are detected, then the LWQ is used to keep the message. This timestamp assigns an order for m_1 among the messages sent from its group members, then m_1 is multicasted to cu_{22} with a timestamp vector of size 2. If m_1 is originally directed to cu_2, then the message is not multicasted to $\mathcal{T}_{22}$; rather, it is directed from $\mathcal{T}_{22}$ to $\mathcal{T}_2$, where it is timestamped and multicasted to cu_2 members with a timestamp vector of size 3.

These different timestamps are necessary to ensure a total order for message delivery. The data structure used by the protocol is described in Figure 3.10 and Section 3.5. The acquisition of a timestamp from the TFMs while the message traverses the hierarchy serves mainly to reserve an order slot for the message within each CU. This reservation scheme ensures the potential causality properties [50].

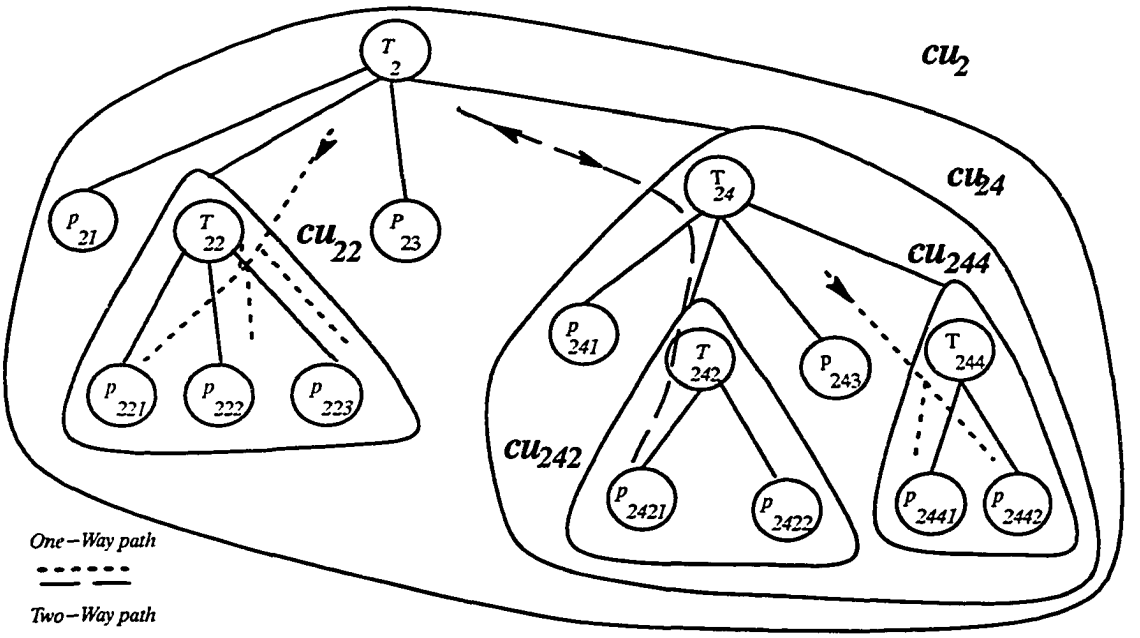


Figure 5.1: Communication structure for BUS-TO protocol.

5.2 BUS-TO Protocol Description

The ordering that is guaranteed by the BUS-TO protocol meets the requirements for both *total* and *causal* order (see Section 2.3.1). The concept of the BUS-TO protocol is simple; for local messages within a CU, all messages will be timestamped at the TFM node. Because each timestamp node is unique, ordering the messages based on this timestamp will ensure a total order for the local messages. For global messages, the timestamps received from TFM along the message path from the sender to the LCA in the way up on the communication hierarchy will be used to order the messages in the CUs. For CUs that have not contributed in passing the message to its LCA, the timestamp of the $LCA(m)$ is used along with other entries in the message timestamp vector. The CUs that have contributed (through the TFMs) to move the message from its sender to its LCA, use the entries of

the message timestamp vector that belongs to the TFMs in the path between the receiving node and $LCA(m)$.

If the message m_k is sent from an activity process p_{iz}, then this message will be directed toward its LCA, where it will be timestamped and ultimately multicasted under its LCA. This message m_k, on its way toward the LCA, passes by all TFMs between p_{iy} and $LCA(m_k)$ in the hierarchy. This defines the TW (*Two-Way* path) for m_k, because m_k traverses this path twice. The first time occurs during the timestamp collection from $\mathcal{T}_i$ to LCA(m_k); the second time occurs after m_k is timestamped at $LCA(m_k)$ and is multicasted in its LCA's subtree. On the contrary, the OW path consists of all TFMs that connect $LCA(m_k)$ and p_{iz} and does not belong to {TW path - $LCA(m_k)$}, such that p_{iz} is a recipient of m_k other than the sender. If $LCA(m_k)$ belongs to an OW path, then this path is a type $\mathcal{A}$ path (denoted $\mathcal{OWA}$); otherwise, it is a type $\mathcal{B}$ path (denoted $\mathcal{OWB}$). For example, in Figure 5.1 m_k is a message that is multicasted by p_{2421} and is received by group cu_2. The $LCA(m_k)$ is $\mathcal{T}_2$, and the TW of m_k is ($\mathcal{T}_2$, $\mathcal{T}_{24}$, $\mathcal{T}_{242}$). An $\mathcal{OWA}$ of m_k is ($\mathcal{T}_2$, $\mathcal{T}_{22}$), and ($\mathcal{T}_{244}$) is the only $\mathcal{OWB}$ path for m_k.

5.3 BUS-TO Protocol Outline

Three types of procedures are available to handle messages: the *Sender*, the *Receiver*, and the *TFM* procedures. The steps for each of these procedures are described below.

5.3.1 Sender

Let m_k be the message sent from process p_{iy} of cu_i with $\mathcal{T}_i$ as its TFM process. The sender p_{iy} performs the steps shown in procedure 5.1 to send message m_k.

SENDER(m_k) {

1. $MTS_{m_k}[0] = LTS_{p_{iy}}$
2. *Increment* $LTS_{p_{iy}}$
3. $MTS_{m_k}[p_{iy}] := LTS_{p_{iy}}$
4. *Send* m_k *to* $\mathcal{T}_i$

}

Procedure 5.1 (Sender)

5.3.2 TFM

Let $\mathcal{T}_x$ be any TFM in the message path from its sender p_{iy} to any of its destinations p_{jx}, and $\mathcal{L} = LCA(cu_i, cu_j)$. Let *Direct Sender* represent the site from which the message m_k was forwarded to $\mathcal{T}_x$. The TFM $\mathcal{T}_x$ will perform one of two handling procedures, depending on whether the message is on its OW path (see procedures 5.2 and 5.3) or its TW path (see procedures 5.4 and 5.5).

- **The OW path procedures**

  Let $\mathcal{OW}$ be the set of processes between $LCA(m_k)$ and $\mathcal{T}_x$ in the OW path and $\mathcal{TW}$ be the set of processes between $\mathcal{T}_x$ and $LCA(m_k)$ that belong to the TW path. Note that $\mathcal{TW}$ could contain just $LCA(m_k)$ if all the paths between $\mathcal{T}_x$ and $LCA(m_k)$ belong to the OW path. Because m_k is on its OW path (i.e., it is moving down the communication hierarchy), the TFM process will execute procedures 5.2 and 5.3.

  The $LCA(m_k)$ timestamp, along with the timestamp copies given to the message from the TFM processes between the receiving process and the LCA(m_k) members of $\mathcal{TW}$, will be used by $\mathcal{T}_x$ to order the message. The message will not acquire a timestamp while it is moving down the hierarchy. However, it will carry copies of the timestamps from the TFMs that are members of

TFM_$\mathcal{OW}(m_k)$ {

1. **if** *MULTICASTABLE_$\mathcal{OW}(m_k)$* →
2. *For each* $m_t \in GWQ$, **if not** *MULTICASTABLE_$\mathcal{OW}(m_t)$* → *exit*
3. **Otherwise**
4. *Admit* m_k *to* GWQ
5. **fi**

}

Procedure 5.2 (TFM_$\mathcal{OW}$)

MULTICASTABLE_$\mathcal{OW}(m_k)$ {

1. **if** *For each* $\mathcal{T}_w$ *in* $\mathcal{TW}$, $MTS_{m_k}[\mathcal{T}_w] = PTS_{\mathcal{T}_x}[\mathcal{T}_w] + 1$
2. **AND**
3. *For each* $\mathcal{T}_z$ *in* $\mathcal{OW}$, $MTS_{m_k}[\mathcal{T}_z] = PTS_{\mathcal{T}_x}[\mathcal{T}_z]$ →
4. Multicast message m_k to cu_x
5. *For each* $\mathcal{T}_w$ *in* $\mathcal{TW}$, Adjust $PTS_{\mathcal{T}_x}[\mathcal{T}_w]$
6. return true
7. **Otherwise**
8. return false
9. **fi**

}

Procedure 5.3 (MULTICASTABLE_$\mathcal{OW}$)

its $\mathcal{OW}$ path as it passes by. This copy is added to the message timestamp vector ($MTS_{m_k}[\ ]$) to ensure its order within the messages that are multicasted from the TFMs along its OW path. This step is important because some messages that are out of order may exist. For example, if m_k arrives at any of the CU members of its OW paths ahead of a local message m_x that was sent by any of these members and if m_x was timestamped by one or more of the $\mathcal{OW}$ TFMs before m_k passed by, then m_k should wait for m_x. This timestamp copy will prevent m_k from being delivered before m_x. Therefore, these entries are needed to ensure the global order between the *global* messages and the relative *local* messages.

- **The TW path procedures**

  In the TW path, the TFM process assumes two roles; one role if the message is moving down the tree (this role is similar to that outlined in the OW path and will be called TFM_$\mathcal{TW}$1) and the second role if the message is moving up. Let $\mathcal{TW}$ be the set of processes located between the sender of the message and $\mathcal{T}_x$ in the TW path and *Direct Sender* be the child process of $\mathcal{T}_x$, from which the message is received. The $OLDTS$ variable is used to ensure that messages are not lost as they come from a process to the TFM (a detailed description of OLDTS functionality is provided in Section 4.3.2). The role assumed by the TFM process if m_k is moving up the hierarchy is outlined in procedures 5.4 and 5.5.

5.3.3 Receiver

Let m_k be the message received by the process p_{jx} of cu_j with $\mathcal{T}_j$ as its TFM process. The receiver p_{jx} executes one of the two handling procedures based on whether or not the message m_k is on its OW path or its TW path. Let $\mathcal{OW}$ be the

TFM_$\mathcal{TW}2(m_k)$ {

```
1.   if MULTICASTABLE_TW2(m_k) →
2.        For each m_t ∈ Wait Queues,
3.          if not MULTICASTABLE_TW2(m_t) → exit
4.   Otherwise
5.        if LCA(m_k) = T_x →
6.             Admit message to LWQ
7.        Otherwise
8.             Admit message to GWQ
9.        fi
10.  fi
```

}

Procedure 5.4 (TFM_$\mathcal{TW}2$)

set of processes located between $LCA(m_k)$ and $\mathcal{T}_j$ that belongs to the OW path and $\mathcal{TW}$ be the set of processes between $\mathcal{T}_j$ and $LCA(m_k)$ that belongs to the TW path. Note that $LCA(m_k)$ could be the only member of $\mathcal{TW}$ if $\mathcal{T}_j$ belongs to one of the OW paths.

- **The OW path procedures**

  The OW path module will follow procedures 5.6 and 5.7. Note that we do not test the LWQ because the *global* messages on its OW path cannot block a *local* message. If the message m_k does not follow the timestamp of its LCA entry in the timestamp vector at the receiver or if any of the corresponding entries of the TFMs other than its LCA do not match, then the message is added to the GWQ.

- **The TW path procedures**

  The TW path module will follow procedures 5.8 and 5.9.

MULTICASTABLE_$\mathcal{TW}2(m_k)$ {

```
1.    if For each T_w in TW − Direct Sender, MTS_{m_k}[T_w] = PTS_{T_x}[T_w] +1
2.          AND
3.    MTSm_k[0] = PTS_{T_x}[Direct Sender] →
4.        Increment LTS_{T_x}
5.        MTS_{m_k}[T_x] := LTS_{T_x}
6.        For each T_z in TW − Direct Sender, Increment PTS_{T_x}[T_z]
7.        PTS_{T_x}[Direct Sender] := MTS_{m_k}[Direct Sender]
8.        if T_x is not the message LCA →
9.          MTS_{m_k}[0] := OLDTS
10.         OLDTS := LTS_{T_x}
11.         Forward m_x up the tree
12.       Otherwise
13.         Multicast m_k to cu_x
14.       fi
15.       if m_k ∈ GWQ →
16.             Remove m_k from GWQ
17.       fi
18.       return true
19.   Otherwise
20.       return false
21.   fi
}
```

Procedure 5.5 (MULTICASTABLE_$\mathcal{TW}2$)

RECEIVE_$\mathcal{OW}(m_k)$ {

1. **if** *DELIVERABLE_*$\mathcal{OW}(m_k) \rightarrow$
2. *For each* $m_t \in GWQ$, **if not** *DELIVERABLE_*$\mathcal{OW}(m_t) \rightarrow$ *exit*
3. **Otherwise**
4. *Admit* m_k *to* GWQ
5. **fi**

}

Procedure 5.6 (Receiver_$\mathcal{OW}$)

DELIVERABLE_$\mathcal{OW}(m_k)$ {

1. **if** *For each* $\mathcal{T}_w$ *in* $\mathcal{TW}, PTS_{p_{jx}}[\mathcal{T}_w] = MTS_{m_k}[\mathcal{T}_w] + 1$
2. **AND**
3. *For each* $\mathcal{T}_z$ *in* $\mathcal{OW}, PTS_{p_{jx}}[\mathcal{T}_z] = MTS_{m_k}[\mathcal{T}_z] \rightarrow$
4. *Admit message to DQ*
5. *For each* $\mathcal{T}_w \in \mathcal{TW}$, *Increment* $PTS_{p_{jx}}[\mathcal{T}_w]$
6. **if** $m_k \in GWQ \rightarrow$
7. *Remove* m_k *from* GWQ
8. **fi**
9. *return true*
10. **Otherwise**
11. *return false*
12. **fi**

}

Procedure 5.7 (Deliverable_$\mathcal{OW}$)

RECEIVE_$\mathcal{TW}(m_k)$ {

```
1.    if DELIVERABLE_TW(m_k)
2.        For each m_t ∈ Wait Queues, if not DELIVERABLE_TW(m_t) →
      exit
3.    Otherwise
4.        if m_k is a local message
5.            Admit m_k to LWQ
6.        Otherwise
7.            Admit m_k to GWQ
8.        fi
9.    fi
```

}

Procedure 5.8 (Receive_$\mathcal{TW}$)

DELIVERABLE_$\mathcal{TW}(m_k)$ {

```
1.    if For each T_z in TW, MTS_{m_k}[T_z] = PTS_{p_jx}[T_z] + 1 →
2.        Admit message to DQ
3.        For each T_z in TW, Adjust PTS_{p_jx}[T_z]
4.        if m_k ∈ GWQ
5.            OR
6.        m_k ∈ LWQ →
7.            Remove m_k from Wait Queue
8.        fi
9.        return true
10.   Otherwise
11.       return false
12.   fi
```

}

Procedure 5.9 (DELIVERABLE_$\mathcal{TW}$)

5.4 BUS-TO Protocol Correctness

We must show that the message order will be maintained in accordance with the criteria (rules 1 and 2) introduced in Section 2.3.1. Remember that we assume a reliable environment with no loss of messages or site failure.

- Rule 1:

  A CU (cu_i) has a set of cooperating processes $\{p_{i1}, p_{i2}, p_{i3}, p_{i4}, ..., p_{in}\}$, and $\mathcal{T}_i$ is the TFM of cu_i. Assume that we have two messages m_1 and m_2 sent from p_{iy}. We must show that m_1 will be received before m_2 at all receiving processes.

  Two messages m_1 and m_2 are sent from p_{iy} of cu_i, where $m_1 \prec m_2$ is directed to $cu_{x_{x\neq i}}$. Assume that m_2 is delivered at one of the cooperating processes $p_{xl_{x\neq i}} \varepsilon C$ before m_1. Because m_2 is delivered before m_1, this implies that $\forall\ x_{x\in\mathcal{TW}}\ MTS_{m_2}(x) \leq MTS_{m_1}(x)$ ($\mathcal{TW}$ is the set of TFMs in the path between p_{xl} and $LCA(m_2)$ that belongs to the TW path).

  Because m_2 and m_1 are sent from p_{iy}, this implies that $MTS_{m_1}[p_{iy}] \prec MTS_{m_2}[p_{iy}]$. If we assume that m_2 arrives at $\mathcal{T}_i$ before m_1 and is admitted for timestamping, then $MTS_{m_2}(\mathcal{T}_i) \prec MTS_{m_1}(\mathcal{T}_i)$; however, this is not the case. Because the message delivery is made in accordance with the timestamp order and because a site will delay the messages until those with a smaller timestamp are delivered, even if m_2 arrives before m_1 it will be admitted to the *wait queues* until m_1 arrives and is passed to the DQ. As a result, $MTS_{m_1}(\mathcal{T}_i) \prec MTS_{m_2}(\mathcal{T}_i)$ at $\mathcal{T}_i$. Similarly, we can show that at any $\mathcal{T}_x$, $\mathcal{T}_x \leq lower(LCA(m_1), LCA(m_2)), MTS_{m_1}[\mathcal{T}_x] \prec MTS_{m_2}[\mathcal{T}_x]$, which contradicts our assumption and shows that m_2 will not be delivered before m_1. This proves the first potential causality case of Lamport [50] for the message delivery.

- Rule 2:

  We must also show that the protocol follows the second rule of Lamport [50], which is introduced in Section 3.1.

  Assume that p_{jx} sends m_1 to cu_i, and m_1 is delivered to p_{iy}. Then p_{iy} sends a message m_2 to cu_i. In accordance with Lamport's potential causality, m_1 must be delivered before m_2 at all processes where m_1 and m_2 are both to be delivered.

  Assume that $p_{iy} \varepsilon cu_i$ sends message m_1 to cu_j. After receiving m_1, $p_{jx} \varepsilon cu_j$ sends m_2 to cu_t, which is also a destination of m_1. According to Lamport, m_1 should be delivered before m_2. The relationship of cu_j to cu_t can be one of four cases in our hierarchy: $cu_t = cu_j$, cu_j is a descendant of cu_t, cu_t is a descendant of cu_j, or cu_t and cu_j are in a brotherhood relation. In order to show adherence to Lamport's rule 2, we must show for the first three cases that the order condition will be honored.

  1. $cu_t = cu_j$.

     Because m_1 is delivered at p_{jy} before p_{jy} sends m_2, this implies that m_1 has already been timestamped at $\mathcal{T}_j$. When m_2 arrives at $\mathcal{T}_j$, it will be timestamped. Because $\mathcal{T}_j$ increases its timestamp with each message, $MTS_{m_1}(\mathcal{T}_j) \prec MTS_{m_2}(\mathcal{T}_j)$. At any of the cooperating processes of CU_j, m_2 will be delivered after m_1 because of the timestamp order.

  2. cu_j is a descendant of cu_t.

     (a) If m_1 passes cu_j before it passes cu_t, then p_{iy} is a descendant of cu_j; m_1 is timestamped at cu_i and moves on for its higher timestamp at cu_t. When it arrives at $LCA(m_1)$, m_1 will be multicasted down the tree. In other words, cu_t will multicast m_1 to its members and forward it down to cu_j. After m_1 is delivered at p_{jx}, p_{jx} will send

m_2, and m_2 will move toward its LCA as it gains its timestamps. In addition, $\forall \mathcal{T}_z : p_{jx} \prec \mathcal{T}_z \prec lower(LCA(m_1), LCA(m_2))$, which m_2 will pass and which will cooperate in constructing the m_1 timestamp vector. So, $\forall \mathcal{T}_z : p_{jx} \prec \mathcal{T}_z \prec lower(LCA(m_1), LCA(m_2))$, and $MTS_{m_1}[\mathcal{T}_z] \prec MTS_{m_2}[\mathcal{T}_z]$. In accordance with rule 1, we can show that m_1 is delivered before m_2.

(b) If m_1 passes cu_t before cu_j.

In this case, a timestamp is given to m_1 from a higher level TFM; this timestamp will be unique to the TFM. Upon its arrival at p_{jx}, m_1 will be delivered, and then m_2 will be sent. Clearly, m_2 will be submitted after m_1 at both cu_j and cu_t because of its higher value timestamp assignment at the TFMs on its way toward its LCA.

Because m_1 has already passed these processes on its OW path, it has received a copy of the timestamp entry of each TFM ($PTS_{\mathcal{T}_z}[\mathcal{T}_z]$, where $\mathcal{T}_z \in OW\ path\ of\ m_1$). This implies that m_2 timestamp values at these TFMs (common TFMs in the m_1 OW path and m_2 TW path) will be larger. As a result, $MTS_{m_1}[\mathcal{T}_z] \prec MTS_{m_2}[\mathcal{T}_z]$, where $\mathcal{T}_z$ are the common TFMs in both the m_1 OW path and the m_2 TW path.

3. If cu_t is a descendant of cu_j.

This case is similar to the previous one.

4. cu_t and cu_j are in a brotherhood relation.

The primary timestamp that affects the order is the timestamp given by the $LCA(m_1)$. Let $\mathcal{T}_{j,t} = LCACU(cu_j, cu_t)$. Because $LCA(m_1)$ should be higher or equal to the smaller CU that contains both cu_j and cu_t ($\mathcal{T}_{j,t}$), m_1 is directed to both of them. So m_1 will be timestamped at ($\mathcal{T}_{j,t}$) and then will proceed to $LCA(m_1)$ (if higher than $\mathcal{T}_{j,t}$).

After m_1 arrives at cu_j, the process p_{jx} will forward m_2 to its LCA. Because m_2 targets both cu_j and cu_t, it will stop at least at $\mathcal{T}_{j,t}$; $\mathcal{T}_{j,t}$ is the first common parent process. This implies that $MTS_{m_1}[\mathcal{T}_{j,t}] \prec MTS_{m_2}[\mathcal{T}_{j,t}]$. Because the timestamp given to m_2 by $\mathcal{T}_{j,t}$ is part of the message vector, $MTS_{m_2}[\mathcal{T}_{j,t}] \prec PTS_{p_t}[\mathcal{T}_{j,t}]$.

For those TFMs in the TW path of m_2, which begins at $\mathcal{T}_j$ and extends through $lower(LCA(m_1), LCA(m_2))$, m_1 has already visited the common TFMs in this path. This implies that *for all* $\mathcal{T}_z$ *where* $p_{jx} \prec \mathcal{T}_z \prec lower(LCA(m_1), LCA(m_2))$, $MTS_{m_1}[\mathcal{T}_z] \prec MTS_{m_2}[\mathcal{T}_z]$, which forces m_2 to wait until m_1 is delivered. Therefore, m_1 is delivered before m_2.

5.5 TDS: Top-Down Stamping Protocol

The *Top-Down Stamping (TDS) Protocol* achieves a reliable multicasting delivery of messages. In addition to honoring Lamport rules, the TDS protocol achieves a total order between multicasted messages. In this protocol, the message does not gain the timestamp on its way up as in the BUS-TO protocol; the timestamp is gained in the message path down the hierarchical structure. A message m_k sent from any process is timestamped at the process and then sent directly to LCA(m_k). The message upon its arrival to LCA(m_k) will be checked for possible delivery (i.e., it is not violating any ordering criteria) and either timestamped then multicasted down the hierarchy or kept in one of the queues until it is ready for timestamping. If the message reaches another TFM, then a similar procedure to the one outlined above is followed. Before multicasting the message to its CU members, each TFM will add an entry to the message timestamp vector. If the message is received by a cooperating process, then its timestamp vector is checked. If the vector does not identify a violation of any ordering criteria, then the message is deliverable and

will be admitted to the delivery queue DQ; otherwise, the message is added to one of the wait queues.

Several lists are added to allow the protocol to work correctly and efficiently. A list, the *Holding List* (HL_S), is added at each node S. It contains all the message IDs that have been sent out of a node S but haven't reached S on their way down. The HL_S is added at each sender S in the hierarchy to ensure the enforcement of the order between messages sent from the same node. Another list, *Received List* (RL_T), is added at each TFM (T), which contains the messages received by this TFM.

By directing the message immediately to its LCA, we saved the time the message needs to reach the LCA by passing through all intermediate TFM processes in the hierarchical structure. Also, message m_k does not block any local message m_x where $Sender(m_k) \prec LCA(m_x) \prec LCA(m_k)$, unless $Sender(m_k) = Sender(m_x)$.

The overhead for using this protocol can be directed between two main factors: the space overhead needed at each cooperating process to keep HL_S and the one required at each TFM to keep RL_T. This space is finite because messages are removed from lists based on certain criteria. The second factor is presented in HL_{m_k} that accompanies the message. The protocol during the message path decreases the size of the holding list at each TFM by marking the delivered entries in the list. A decrease in the size of HL_{m_k} helps to lower the number of comparisons performed at lower level TFMs. Another overhead that is encountered is the need for the cooperating processes to know the communication structure in order to direct the messages to the required LCAs. In addition, for correct protocol functionality the TFMs must be able to identify the relationships between the different LCAs. This knowledge is necessary in comparing the HL_{m_k} entries, which requires the delivery of the messages contained in HL_{m_k} that have an LCA in a higher level than $LCA(m_k)$ before m_k.

5.6 Conclusion

In this chapter, the Bottom-Up Stamping-Total Order (BUS-TO) protocol, which is a reliable ordered multicasting protocol, is presented. The development of this protocol was fueled by the need for multicasting protocols that can support the existence of inter-group and intra-group messages in an interconnected LAN environment. The protocol depends on forcing the communication between the processes to follow a certain hierarchical communication structure. The knowledge of this structure allows efficient multicasting for local messages. The BUS-TO protocol ensures a total order among multicasted messages. An initial overhead is encountered to build the communication structure necessary to support the protocol. However, the protocol performs better than many existing multicasting protocols, as will be shown in Chapter 8. This superior performance is due to smaller storage requirements and a reduced communication overhead. Also, because the hierarchical structure can be mapped to the communication topology used by the message, therefore, no extra protocol messages are required to achieve ordered delivery. Also, the use of the CU hierarchy in multicasting decreases the number of physical messages sent on the internet; the protocol assumes that the messages have all CUs under their LCAs in their destinations. The problem with this assumption is that some CUs under $LCA(m)$ will receive m although they are not targeted by m. This effect can be diminished if the group memberships are taken into consideration when the CU structure is built. Furthermore, the introduction of additional data structures can eliminate the need to make this assumption.

Chapter 6

MLMO: Multi-LAN Multi-Order Protocol

6.1 Introduction

As the demand for economic and effective sharing of resources (data and otherwise) grows, a new environment characterized by interconnected LANs that belong to different autonomous entities has emerged. Autonomy is manifested, among other things, by different LANs that adopt possibly different ordering criteria for multicasting.

Numerous ordered reliable atomic multicasting protocols have been proposed [19, 12, 63, 56]. The majority of these protocols adopt the assumption of a single LAN that has multicasting capabilities [19, 63]. Unfortunately, almost all of the proposed protocols enforce only one ordering criterion system-wide. Birman and Joseph [12] have proposed a multicasting protocol that can handle multiple message streams, each associated with a single ordering criterion. Messages from the same stream are ordered for delivery according to this criterion, independent of the recipient. In effect, this deprives the recipients of their autonomy in determining

their own criteria for ordering delivery of incoming messages. Conclusively, neither the enforcement of a single ordering criterion nor the elimination of recipient control over the ordering criterion is acceptable in this heterogeneous setting.

This chapter proposes the Multi-LAN Multi-Order protocol (MLMO) designed to support ordered atomic reliable multicasting across interconnected LANs. Our protocol insists on a hierarchical structure in the communication topology. This structure can be one that reflects the actual physical connections, one that is inferred by studying the group interactions, or one that is simply imposed over the message flow to honor the protocol requirements. The protocol uses the same communication model outlined in Chapter 3, with the communication hierarchy shown in Figure 6.1. Members of one group can be individual processes and/or other groups. The protocol does not restrict the members of a group to the same LAN. Additionally, the protocol allows each group to determine a causal or total ordering criterion. Therefore, our multicasting environment contains two types of groups: the *causal groups* that enforce a causal order and the *total groups* that enforces a total order. Notice that our protocol can circulate messages that have some destinations in total groups and other addressees in causal groups yet still observe the particular ordering criterion for each addressee's group. Note that a given group's ordering criterion pertains to members that are individual processes and not members that are groups, because the later would, by definition, have their own criteria.

The MLMO assumes no sequenced delivery service from the underlying communication network. This assumption is realistic because different routes can be used by messages sent from the same sender to the same recipient. Our failure model for handling network partitions, message loss, and crash failures is presented in Chapter 9.

The remainder of the chapter is organized as follows. Section 6.2 presents the

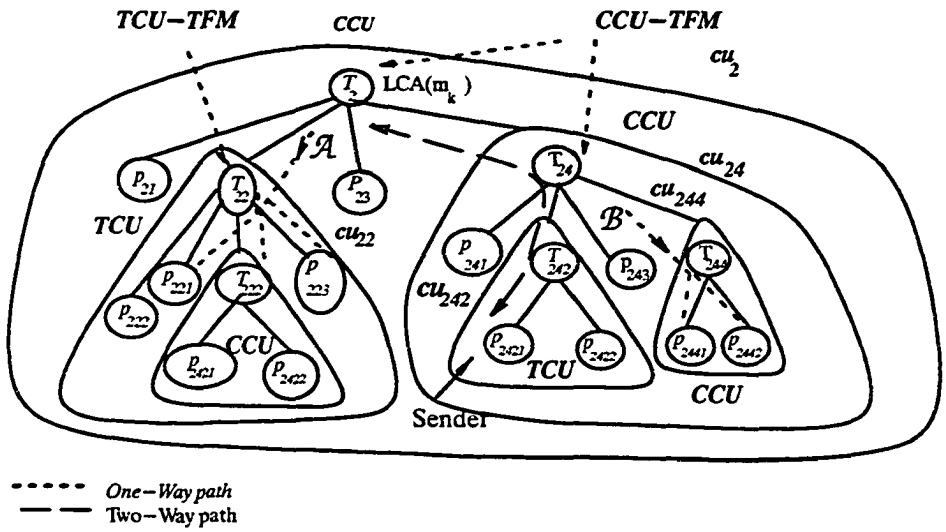


Figure 6.1: Communication structure for MLMO that shows both CCU and TCU.

MLMO protocol and explains how the protocol handles the multicasting of different messages. Several problems that result from the accommodation of multiple ordering criteria are also discussed in this section. The protocol is outlined in Section 6.3. Finally, conclusion are presented in Section 6.5.

6.2 The MLMO Protocol

Two types of communication units can be identified under the MLMO protocol: a causal-order communication unit (CCU) and a total-order communication unit (TCU). In a CCU, a causal order is enforced; in a TCU, a total order is enforced. Note that this total order is common to all TCUs in our communication structure.

The MLMO is realized as two separate yet interacting protocols, one for handling the CCUs and the other for handling the TCUs. The interaction between the two protocols is driven by the timestamping vectors assigned by the TFMs. A

CCU runs a modified version of the BUS protocol (see Chapter 4 and [78, 80, 79]). The BUS protocol forwards a message m to the TFM of the sender, where it is timestamped and multicasted to members of the sender's CU. The message m is then forwarded to the parent of its TFM until the $LCA(m)$ has been reached. As proven in Section 4.4, this protocol guarantees a causal order among the multicasted messages. A TCU runs a modified version of the BUS-TO protocol (see Chapter 5), which enforces a total order among multicasted messages.

The major challenge that confronts the MLMO is not only to enforce different ordering criteria for message delivery but to ensure that no conflicts arise as a result of enforcing such different orders. Specifically, because ordering criteria may not be totally independent, the enforcement of one can potentially violate another. For example, total-order enforcement for delivering messages m_1 and m_2, which both originate from process p_i, should not violate their inherent causal relationship (order). Hence, the objectives of our protocol can be stated as:

1. All messages interrelated by a causal order are guaranteed to be delivered in their causal order to all recipients (regardless of whether they are members of CCUs or TCUs). This rule will be referred to as the "causal order" rule.
2. All messages *not* interrelated by a causal order are guaranteed to be delivered in identical order to all recipients that are members of TCUs. This rule will be referred to as the "total order" rule.

A detailed description of the MLMO protocol is given below.

6.2.1 Protocol Description

Assume that m_k is a message sent from p_{iy} of cu_i (with $\mathcal{T}_i$ as its TFM process) and that one of the recipients is p_{jx} of cu_j (with $\mathcal{T}_j$ as its TFM process). The message m_k is timestamped at p_{iy} and then forwarded to $\mathcal{T}_i$ on its way toward $LCA(m_k)$.

The message may encounter two types of TFMs: a TCU type (TCU-TFM) or a CCU type (CCU-TFM). These two types require different steps to achieve the correct order. Message m_k is directed up the hierarchy toward its LCA, where it is timestamped and ultimately multicasted in the subtree of its LCA. On its way toward its LCA, m_k passes by all TFMs between p_{iy} and $LCA(m_k)$ in the hierarchy. This defines the TW path for m_k because m_k traverses this path twice. The first time occurs during the timestamp collection from $\mathcal{T}_i$ up the hierarchy to $LCA(m_k)$. The second time occurs after m_k is committed for delivery and is multicasted in its LCA's subtree. On the other hand, the OW path is one that contains all TFMs that connect $LCA(m_k)$ and p_{iz} and does not belong to {TW path - $LCA(m_k)$}, such that p_{iz} is a recipient of m_k other than the sender. Two types of paths are encountered here: $\mathcal{OWA}$ and $\mathcal{OWB}$ (readers are referred to Chapter 5 for the definitions of these paths). For example, in Figure 6.1 m_k is a message multicasted by p_{2421} and received by group cu_2, $\mathcal{T}_2$ is the LCA(m_k), TW of m_k is ($\mathcal{T}_2$, $\mathcal{T}_{24}$, $\mathcal{T}_{242}$), and ($\mathcal{T}_2$, $\mathcal{T}_{22}$) is an $\mathcal{OWA}$ of m_k; ($\mathcal{T}_{244}$) is the only $\mathcal{OWB}$ path for m_k.

As it passes by either CCU or TCU on the way to its LCA, m_k is checked for the correct timestamp. If it is out of timestamp order, then m_k is kept in OOQ; otherwise, the message is timestamped. If m_k is at $\mathcal{T}_x$ such that cu_x is a CCU, then a copy of m_k is multicasted to cu_x. If $\mathcal{T}_x$ is not the $LCA(m_k)$, then m_k is sent to the next highest TFM.

As it arrives at LCA(m_k), m_k is assigned a timestamp and multicasted down the hierarchy. The $\mathcal{OWA}$ paths will receive the message for the first time. In these paths, the LCA timestamp will be the main ordering timestamp. Message m_k is delivered at each TFM, where it is timestamped and multicasted to the CU members. As a message traverses down the hierarchy on $\mathcal{OWA}$ paths, the TFMs will execute either CCU_TFM(m_k) or TCU_TFM_DN_OW(m_k), depending on whether

the TFM is CCU-TFM or a TCU-TFM, respectively (see the MLMO protocol in Section 6.3). Upon the arrival of m_k at any of the recipient processes p_{jx}, p_{jx} will execute either CCU_RECEIVE(m_k) or TCU_RECEIVE_OW(m_k), depending again on the type of $\mathcal{T}_x$.

Meanwhile, LCA(m_k) will forward m_k down its TW path for delivery to all processes under this path. Some recipients are found along the TW path (p_{2422} in Figure 6.1), while others are reachable from the TW path along an $\mathcal{OWB}$ path (p_{2442} in Figure 6.1). For those TFMs that are in $\mathcal{OWB}$ paths, either procedure CCU_TFM(m_k) or TCU_TFM_DN_OW(m_k) is executed. Upon arrival at any of the recipient processes p_{jx} along $\mathcal{OWB}$ paths, p_{jx} will execute either CCU_RECEIVE(m_k) or TCU_RECEIVE_OW(m_k). Alternatively, for those TCU-TFMs that are members of the TW path, procedure TCU_TFM_DN_TW(m_k) will be executed. Note that the timestamp comparison here is based on the TFM timestamps given to the message by all members of $\mathcal{AT}$, where $\mathcal{AT} = \{\mathcal{T}_x : LCA(m_k) \geq \mathcal{T}_x > \textit{recipient TFM}\}$. For the CCU-TFMs that are members of the TW path, the CCU-TFM(m_k) will be executed. Similarly, upon arrival at any of the recipient processes p_{jx} along the TW path, p_{jx} will execute either CCU_RECEIVE(m_k) or TCU_RECEIVE_TW(m_k).

Note that the MLMO protocol meets the requirements of both the causal-order rule and the total-order rule. The total-order rule is satisfied because the delivery order of messages to all recipients that belong to TCUs is identical. On the other hand, the causal-order rule is satisfied because any CCU delivers messages to its members in accordance with causal order. Messages sent by a CCU or a TCU can be received by processes in CCUs or TCUs.

The basic MLMO protocol described above allows message exchange between the CCUs and the TCUs. This interaction allows a message to be delivered based on the order enforced by its recipient's CU. The order of the messages for all

recipients that belong to TCUs is the same. However, the CCU that delivers the messages enforces a causal order. According to the basic MLMO protocol presented above, the TCU-TFM in the TW path, when visited for the first time by m_k, will timestamp the message and forward it to its LCA. If any CCU exists under this TCU-TFM, the message will not be delivered until it gains its LCA timestamp. Obviously, the CCU does not need the LCA timestamp to enforce a causal order; hence, blocking the message by the TCU results in a delay of message delivery that would have not occurred if all CUs in a communication structure were CCUs. In order to eliminate this blocking effect, a bypass approach has been introduced to the protocol. This approach is described in the following subsections.

6.2.2 Message Bypass Problems

To eliminate the blocking effects of the TCU-TFM, the bypass approach is introduced to the MLMO protocol. The main idea behind this approach is to allow the message to bypass the TCU-TFM and go to any CCU under this TCU-TFM in the $\mathcal{OWB}$ path. The bypass approach will speed up the delivery of the messages to the CCUs in the $\mathcal{OWB}$ paths. Assume that a message m_k is sent from p_{iy} along the TW path toward its LCA and that it reaches a TCU-TFM ($\mathcal{T}_x$). After it is timestamped at $\mathcal{T}_x$ (where $\mathcal{T}_x$ is any TFM in TW and $\mathcal{T}_x \neq LCA(m_k)$) and is forwarded up the hierarchy, a copy of m_k is multicasted in the $\mathcal{OWB}$ paths that contain any CCU under $\mathcal{T}_x$. This excludes the paths from which m_k originated. When m_k reaches $\mathcal{T}_y$ (a TFM under $\mathcal{T}_x$ in one of the $\mathcal{OWB}$ paths), m_k will be scheduled for timestamping. The message m_k is not carrying an LCA timestamp; therefore, the message can only be delivered to CU_y members if CU_y is a CCU because causal order does not require the LCA timestamp. If CU_y is a TCU, then $\mathcal{T}_y$ will not deliver m_k to its processes and will forward m_k along the paths that contain any CCU under $\mathcal{T}_y$, after m_k is timestamped. This process essentially eliminates the

blocking of messages from delivery to the CCUs because of the existence of a TCU between the receiving processes and the TW path members. Because the CCU is in a heterogeneous environment (one that contains both CCUs and TCUs), further steps must be taken to ensure that this causal order does not contradict any total order assumed by the TCUs.

After m_k reaches its LCA, it is forwarded along its TW path for delivery. All processes that are not in $\mathcal{OWA}$ paths (processes that are either directly managed by the TFM in TW or $\mathcal{OWB}$) would have already received the message; therefore, in addition to those CUs in the $\mathcal{OWA}$ paths, only the TCUs in the TW path and the $\mathcal{OWB}$ paths other than $LCA(m_k)$ will be targeted by the message on its way down. Message m_k, on its way down, will pass by some of the TCU-TFMs under TW for the second time. It could have been timestamped by these TCU-TFMs when the message bypassed them for delivery to the CCU-TFMs before the LCA timestamp was gained. The message, from the time it is timestamped by these TCUs until it is reflected to the PTS_{TW}, is called a *hidden message*. Message m_k must gain the same timestamp assigned to the copy of m_k directed through this path earlier. This problem is known as the *timestamp incarnation* problem and will be discussed later. In this path, the CCU-TFM must disregard the copy of the message directed down because the message was already delivered to the CCUs. These CCUs simply forward the copied messages down the hierarchy.

Although the TCU-TFM obtains messages before they receive the LCA timestamp, these messages are unable to be delivered by any TCU-TFM until their LCA timestamp is received. The TCU-TFM will bypass these messages down the hierarchy and will timestamp them to reserve an order. These messages, however, will not be delivered at any TCU-TFM and will not change any of the timestamp vectors kept at the TCU-TFMs until the version that has the LCA timestamp is received. The $PTS_{T_x}[\ ]$ is not updated because an update with m_k information

implies that m_k has been delivered to $\mathcal{T}_x$.

The bypass approach affects the delivery of the message in the TCUs because blocking can affect messages whose LCAs reside under the TW path of m_x. Because these messages would have been delivered without waiting for the $LCA(m_x)$ timestamp, the addition of the bypass to the protocol blocks it. Therefore, the bypass approach may cause a delay in message delivery. This delay is eminent if the frequency of similar cases is high. In addition, the blocking effect depends on how the CCUs and TCUs are distributed in the structure, the frequency of global messages, and their LCA positions in the structure.

6.2.3 Timestamp Gap Adjustment

A problem occurs when m_k is traveling along its $\mathcal{A}$ paths after it receives its LCA timestamp (see Figure 6.1). To better visualize this problem, assume that a message m_y is sent from $s_{m_y} \in \mathcal{A}_{m_k}$ and that $LCA(m_y) > LCA(m_k)$. Also assume that m_y is on its way toward $LCA(m_y)$, with m_k traveling along its $\mathcal{A}$ path. Assume that $\mathcal{T}_x$ is a TCU-TFM in the path from s_{m_y} to $LCA(m_k)$ and that both m_k and m_y will meet at $\mathcal{T}_x$ (m_y gains a smaller timestamp than m_k). With the normal timestamp delivery order, m_y should be delivered at $\mathcal{T}_x$ before m_k because $MTS_{m_k}[\mathcal{T}_x] > MTS_{m_y}[\mathcal{T}_x]$. Because m_y has not yet gained its LCA timestamp, m_k will be admitted to one of the waiting queues until the copy of m_y that carries its LCA timestamp is delivered; m_y will keep going toward its LCA and will pass the TFMs that m_k has already passed on its way down. This means that for each $\mathcal{T}_z$, where $\mathcal{T}_x < \mathcal{T}_z \leq LCA(m_k)$, $MTS_{m_k}[\mathcal{T}_z] < MTS_{m_y}[\mathcal{T}_z]$. A timestamp ordering conflict occurs here between the messages going down their $\mathcal{A}$ paths and the messages going up their TW paths. To overcome this problem, the protocol does not allow the messages that are moving along their $\mathcal{A}$ paths to change the timestamp vector (PTS[ ]) at the TCU-TFMs. The messages will be

timestamped at each TFM along the $\mathcal{A}$ paths. The delivery at the CCU-TFM will be handled by comparing the message and process timestamp vectors (MTS and PTS). The delivery at the TCU-TFMs is performed with the message LCA timestamp and a message list ($CLCAM_{m_k}$) that is carried with the message. The $CLCAM_{m_k}$ contains the message identifiers that should have been delivered before the message to the processes along this path (this list will be described in Section 6.2.5). Because the message will not update the TCU-TFM timestamp vector while it is going down its $\mathcal{A}$ paths, a problem is created in the delivery of the messages that are moving down the hierarchy in their TW and $\mathcal{B}$ paths. These messages will be blocked because they appear to be missing messages. These missing messages may not be actually missing; however, they are viewed as such because the messages in their $\mathcal{A}$ paths do not update the PTS vectors. This results in a *timestamp gap problem* at these TCU-TFMs. To solve this problem, a new structure, the *Timestamp Updater List* (TSUL), is added at each TCU-TFM. Any message on its way down the hierarchy, when it passes any TFMs that belong to its $\mathcal{A}$ paths, is assigned a timestamp and is not allowed to change the timestamp vector of the TCU-TFM. The message, after it has been assigned a timestamp, adds an entry to the TSUL. This entry contains the timestamp assigned to the message by the TFM. Any message on its way down its TW path will check the list and will group all messages that precede it in timestamp order. When the message is delivered at any of the lower sites, the $PTS_{\mathcal{T}_x}[\,]$ will be modified with the entries gathered from the TCU-TFMs of the TW path.

6.2.4 Timestamp Incarnation

One of the problems encountered because of the bypass scheme occurs when m_k gains its LCA timestamp and is directed down the hierarchy to be delivered to its TCUs along $\mathcal{OWB}$ paths. Assume that $\mathcal{T}_x$ is a TCU in an $\mathcal{OWB}$ path. Because of

the bypass scheme, m_k has been sent during the timestamp collection phase to $\mathcal{T}_x$ and, thus, has obtained a timestamp $MTS_{m_k}[\mathcal{T}_x] = t_0$ upon arrival. In accordance with MLMO, m_k will eventually arrive for the second time at $\mathcal{T}_x$ after it gains its LCA timestamp and will obtain a new timestamp $MTS_{m_k}[\mathcal{T}_x] = t_2$ such that $t_2 > t_0$. If m_p has arrived at $\mathcal{T}_x$ and obtained a timestamp $MTS_{m_k}[\mathcal{T}_x] = t_1$, where $t_0 < t_1 < t_2$, then a timestamp conflict results. According to t_0, m_p must be delivered *after* m_k; according to t_2, m_p must be delivered *before* m_k.

To solve this problem, MLMO prescribes that m_k should never obtain the t_2 timestamp and that the first copy of m_k should be incarnated by its second copy. In other words, upon the second arrival of m_k, MLMO sets $MTS_{m_k}[\mathcal{T}_x] = t_0$. One important problem that arises during message delivery along $\mathcal{OWA}$ paths is called *timestamp ordering conflict.* The problem occurs when one message m_l that is traversing its TW path meets another message m_k that is traversing one of its $\mathcal{OWA}$ paths at a TCU-TFM ($\mathcal{T}_x$). Assume that LCA(m_l) > $LCA(m_k)$ and $MTS_{m_k}[\mathcal{T}_x] > MTS_{m_l}[\mathcal{T}_x]$. For each $\mathcal{T}_z$, where $\mathcal{T}_x < \mathcal{T}_z \leq LCA(m_k)$, $MTS_{m_k}[\mathcal{T}_z] < MTS_{m_l}[\mathcal{T}_z]$. A conflict arises in imposing a total order between m_k and m_l.

To achieve the timestamp reassignment or incarnation, each TCU-TFM that bypassed the message and timestamped it before the LCA timestamp is gained must identify the message after it receives its LCA timestamp. An additional list, called *the Timestamp Wait List (TWL)*, is needed to handle these messages. The list contains those messages that are timestamped by the TCU-TFM before they gain their LCA timestamps. When a message arrives at the TFM with its LCA timestamp, it is matched against the TWL. If the message is on the list, the timestamp is added to the message timestamp vector and multicasted to the TCU members. After the message is matched and its timestamp reassigned, the message entry is removed from the TWL.

The same timestamp must be reassigned to the message while it passes by $\mathcal{B}$

TFMs because a new timestamp assignment would result in a wrong message order between CUs. The TWL is not an infinite list because this list is the first time the message is directed down the hierarchy from the CCU delivery phase (i.e., it bypasses the TCUs). The message is removed from the list the second time the message is directed through this path for TCU delivery. Because the message must come back for TCU delivery, it will be removed from the TWL. The size of the TWL depends on the time the message takes to come back to the TFM after it is added to the list. This time depends on the number of levels the TFM is located from the message LCA and the frequency of messages sent that have the TFM as part of the $\mathcal{B}$ set.

6.2.5 Committed Message List

One problem, first described in Section 6.2.3, is the timestamp conflict between messages on an $\mathcal{A}$ path and messages on a TW $\mathcal{B}$ path. To resolve this conflict, we presented the possibility of using the LCA timestamp and the CLCAM list to deliver the message. The CLCAM list identifies the other messages that should be delivered before the current message. Assume that m_k, after it arrives at $LCA(m_k)$, is assigned a timestamp and then is multicasted down the hierarchy. The OW paths immediately under $LCA(m_k)$ (paths marked with $\mathcal{A}$ in Figure 6.1) will receive the message for the first time. For m_k, the LCA timestamp will be the main ordering timestamp for message delivery on these paths. The message, on its way down the $\mathcal{A}$ paths, will not modify the process timestamp vector ($PTS_{\mathcal{I}_x}[\,]$) because of the *timestamp gap problem* discussed in Section 6.2.3. Each TFM keeps two lists, the LCAM and the CLCAM. The LCAM, which is kept at each TFM, contains all messages for which the particular TFM acted as their LCA. The CLCAM contains the part of these messages that have been committed by the LCA. The message, when timestamped at its LCA, will be admitted to the

SENDER(m_k)**{**

1. $MTS_{m_k}[0] = LTS_{p_{iy}}$
2. *Increment* $LTS_{p_{iy}}$
3. $MTS_{m_k}[p_{iy}] = LTS_{p_{iy}}$
4. *Send message to* $\mathcal{T}_i$

}

Procedure 6.1 (Sender)

LCAM and will be multicasted down the hierarchy. The LCAM is a temporary list where messages reside until they are committed for delivery. The messages in $LCAM_{\mathcal{T}_j}$ are waiting for messages that received a smaller timestamp from $\mathcal{T}_j$ and have not passed by $\mathcal{T}_j$ with their LCA timestamp. The arrival of this message will trigger a relocation of messages from LCAM to CLCAM. The CLCAM is the list carried with any message going down its $\mathcal{A}$ path and will be checked at any TCU-TFM in its OW path. The messages in CLCAM should be delivered before m_k. In the case of missing messages, m_k is admitted to the GWQ wait for the missing messages.

6.3 Protocol Outline

Let m_k be a message that is sent from p_{iy} of cu_i with $\mathcal{T}_i$ as its TFM and is received by the process p_{jx} of cu_j; $\mathcal{T}_j$ is the TFM process. Three types of processes handle the messages: the *Sender*, the *Receiver*, and the TFM procedures.

6.3.1 Sender

The sender p_{iy} performs the steps shown in procedure 6.1 to send m_k.

CCU_RECEIVE(m_k) {

1. **if** *CCU_DELIVERABLE*$(m_k) \rightarrow$
2. *For each* $m_t \in LWQ$,
3. **if not** *CCU_DELIVERABLE*$(m_t) \rightarrow$ *exit*
4. **Otherwise**
5. **if** $MTS_{m_k}[T_j] > PTS_{p_{jx}}[T_j] \rightarrow$
6. *Admit* m_k *to LWQ*
7. **Otherwise**
8. *Discard* m_k
9. **fi**
10. **fi**

}

Procedure 6.2 (CCU_RECEIVE)

6.3.2 Receiver

The receiver processes will run one of two procedures, which depends on whether the receiver process is a member of a TCU or a CCU. If the receiver process p_{jx} is a member of a CCU, then the CCU_RECEIVE and CCU_DELIVERABLE procedures will be executed. If p_{jx} is a member of a TCU, then the procedures to be executed will vary based on the position of p_{jx} in relation to the TW and the OW paths. If p_{jx} is a member of the TW path, then procedures TCU_RECEIVE_TW and TCU_DELIVERABLE_TW will be executed. If, on the other hand, p_{jx} is a member of the OW path, then procedures TCU_RECEIVE_OW and TCU_DELIVERABLE_OW will be executed.

- **CCU protocol**

  The p_{jx} will execute the steps outlined in procedures 6.2 and 6.3. Step 4 of procedure 6.2 is added to eliminate the multicasting of the message by a CCU-TFM on its way down the tree for the second time. This step is not

CCU_DELIVERABLE(m_k) {

```
1.    if (MTS_{m_k}[T_j] = PTS_{p_jx}[T_j] + 1) →
2.        Increment PTS_{p_jx}[T_j]
3.        Admit message to DQ
4.        if m_k ∈ LWQ →
5.            Remove m_k from LWQ
6.        fi
7.        return true
8.    Otherwise
9.        return false
10.   fi
```

}

Procedure 6.3 (CCU_DELIVERABLE)

necessary if the TFM filters the messages traveling along the hierarchy in a TW path and sends the messages only in paths that contain a TCU. This step eliminates the multicasting of the messages in the CCUs along the TW path hierarchy. To achieve this filtering task, the TFM must be aware of the structure of the hierarchy beneath it, which requires extra overhead to update the view of the hierarchy at each TFM but prevents the extra multicasted messages in the internet. Another approach, which is a compromise between the two previous processes, is to direct these messages to the TFMs under the TW hierarchy and let the CCU-TFM discard these messages. This approach eliminates the multicasting of these messages to the CCU.

- **TCU protocol**

  As mentioned before, one of two different handling procedures must be followed, depending on whether or not the message is on its OW paths or TW path. Let $\mathcal{OW}$ be the set of processes located between LCA(m_k) and $\mathcal{T}_j$ that

TCU_RECEIVE_OW(m_k) {

1. **if** m_k *already passed by* $LCA(m_k) \rightarrow$
2. **if** $TCU\_DELIVERABLE\_OW(m_k) \rightarrow$
3. *For each* $m_t \in GWQ$,
4. **if not** $TCU\_DELIVERABLE\_OW(m_t) \rightarrow$ *exit*
5. **Otherwise**
6. *Admit* m_k *to* GWQ
7. **fi**
8. **Otherwise**
9. *Discard* m_k
10. **fi**

}

Procedure 6.4 (TCU_RECEIVE_$\mathcal{OW}$)

belongs to the OW path and $\mathcal{TW}$ be the set of processes between $\mathcal{T}_j$ and LCA(m_k) that belongs to the TW path. Note that $\mathcal{TW}$ can be empty if the entire path between $\mathcal{T}_j$ and LCA belongs to the OW path.

- **The OW path module.**

  The OW path module follows the steps outlined in procedures 6.4 and 6.5. Step 1 in procedure 6.4 is added to eliminate the messages that have not gained their LCA timestamp from delivery. This happens because the messages are multicasted down the hierarchy to be delivered to the CCU (bypass approach). This step can be eliminated if the TCU-TFMs filter these messages and do not forward them to the TCU group members. This modification will be discussed when we present the TFM procedure in regard to its impact on other parts of the CCU procedure. Note here that we do not test the LWQ for possible message delivery

TCU_DELIVERABLE_OW(m_k) {

1. if *For each* $\mathcal{T}_w$ *in* $\mathcal{TW}, PTS_{p_{jx}}[\mathcal{T}_w] = MTS_{m_k}[\mathcal{T}_z] + 1$
2. **AND**
3. *For each* $m \in LCAM_{m_k}, PTS_{p_{jx}}[LCA(m)] \geq CLCAM[m] \rightarrow$
4. *Admit message to DQ*
5. *For each* $\mathcal{T}_w \in \mathcal{TW}$, *Increment* $PTS_{p_{jx}}[\mathcal{T}_w]$
6. if $m_k \in GWQ \rightarrow$
7. *Remove* m_k *from GWQ*
8. fi
9. *return true*
10. **Otherwise**
11. *return false*
12. fi

}

Procedure 6.5 (TCU_DELIVERABLE_OW)

because the global message in its OW path cannot block a local message.

- **The Two-Way path module.**

  The TCU_RECEIVE_TW follows the same procedure as the TCU_RECEIVE_OW presented before with one modification: TCU_DELIVERABLE_TW is used instead of TCU_DELIVERABLE_OW. Procedure 6.6 is designed to handle the timestamp gap for the version of MLMO that implements the bypass approach. Procedure TCU_DELIVERABLE_OW provides the necessary steps to handle the TSUL presented in Section 6.2.3 to solve the timestamp gap problem. Any message on its way down the hierarchy in the OW path is tiem-stamped and is not allowed to change the timestamp vector of the

TCU_DELIVERABLE_TW(m_k) {

1. **if** *For each* $\mathcal{T}_z$ *in* $TW, MTS_{m_k}[\mathcal{T}_z] = PTS_{p_{jx}}[\mathcal{T}_z] + 1 \rightarrow$
2. *Admit message to DQ*
3. *For each* $\mathcal{T}_z$ *in* TW, *Increment* $PTS_{p_{jx}}[\mathcal{T}_z]$
4. *Update* $PTS_{p_{jx}}$ *from* $TSUL_{m_k}$
5. **if** $m_k \in GWQ$ *or* $m_k \in LWQ \rightarrow$
6. *Remove* m_k *from Wait Queue*
7. **fi**
8. *return true*
9. **Otherwise**
10. *return false*
11. **fi**

}

Procedure 6.6 (TCU_DELIVERABLE_TW)

TCU-TFM. After the message is timestamped, an entry is added to the TSUL. This entry contains the timestamp assigned to the message by the TFM. Any message on its way down the TW path will check the list and from it will group all messages that have a smaller timestamp. When the message is delivered at any of the lower sites, the $PTS_{\mathcal{T}_x}[\mathcal{T}_y]$ will be modified with this list (see step 4 in procedure 6.6), where $\mathcal{T}_y$ is the site that has given the tiemstamp.

6.3.3 TFM Procedure

The TFM processes will run one of two procedures, depending on whether the TFM process is a member of a TCU or a CCU. If the TFM process $\mathcal{T}_x$ is a member of a CCU, then the CCU_TFM and CCU_MULTICASTABLE procedures will

be executed. If $\mathcal{T}_x$ is a member of a TCU, then the procedures to be executed will vary depending on the position of $\mathcal{T}_x$ in relation to the TW and the OW paths. If $\mathcal{T}_x$ is a member of the TW path, then two sets of procedures can be executed based on whether the message is on its way up or down in the hierarchy. If the message is on its way down, then procedures TCU_TFM_DN_TW and TCU_MULTICASTABLE_DN_TW will be executed. If the message is in its way down, then procedures TCU_TFM_UP_TW and TCU_MULTICASTABLE_UP_TW will be executed. If, on the other hand, $\mathcal{T}_x$ is a member of one of the OW paths, then procedures TCU_TFM_DN_OW and TCU_MULTICASTABLE_DN_OW will be executed.

- **CCU protocol**

  Let $\mathcal{T}_x$ be any TFM in the message path from its sender to any of its destinations, and let $\mathcal{L}$ be the least common ancestor of cu_i and cu_j (LCACU(cu_i, cu_j)). Let m_k be the message sent from p_{iy} of cu_i with $\mathcal{T}_i$ as its TFM process; the message will pass by $\mathcal{T}_x$. Let $\mathcal{OW}$ be the set of TFM processes in the OW path between $\mathcal{T}_x$ and the LCA of m_k; $\mathcal{T}_x$ will execute procedures 6.7 and 6.8.

  The test in step 1 is performed to determine if the message has been timestamped before by the CCU-TFM. If this is the case, the message is forwarded to the dependent TCU-TFMs because all the CCU-TFMs have already received the message.

  Step 7 is executed if the message is timestamped before by $\mathcal{T}_x$ and is on its way down after it reaches its LCA. This step can be modified to direct the message only to the paths that contain a TCU-TFM because the message may have been received by $\mathcal{T}_x$ before (see Section 6.2.2 for the bypass approach). As a result, $\mathcal{T}_x$ timestamps the message and multicasts it to its descendants; therefore, the message is received by all CCU-TFMs in the

CCU_TFM(m_k) {

```
1.   if m_k has not been timestamped by T_x →
2.       if CCU_MULTICASTABLE(m_k) →
3.           For each m_t ∈ LWQ,
4.               if not CCU_MULTICASTABLE(m_t) → exit
5.       Otherwise
6.           Admit m_k to LWQ
7.       fi
8.   Otherwise
9.       Direct m_k to the descendant TFMs
10.  fi
```

}

Procedure 6.7 (CCU_TFM)

descendant hierarchy.

Two approaches can be used in forwarding the message down the hierarchy if it has been previously timestamped by the TFM:

- The CCU-TFMs, after the receipt of a message (regardless of wether the message has been received before or not), assume that the receiver is responsible for identifying this message and discarding it. This option eliminates any extra overhead on the TFM, increases the number of messages on the network, and eliminates the need to maintain a view information at the TFM in regard to TCU/CCU membership in the descendant hierarchy. The previously described protocol implements this approach.
- The CCU-TFM checks the message and forwards it down the hierarchy only if a TCU-TFM exists in the descendant hierarchy of the CCU-TFM. The TCU-TFM then multicasts it to the CU members. This

CCU_MULTICASTABLE(m_k) {

```
1.    if m_k was in its way down →
2.      if MTS_{m_k}[Direct Sender] = PTS_{T_x}[Direct Sender] + 1 →
3.          Increment PTS_{T_x}[Direct Sender]
4.          Increment LTS_{T_x}
5.          MTS_{m_k}[T_x] = LTS_{T_x}
6.          Multicast m_k to cu_i
7.          return true
8.      Otherwise
9.          return false
10.     fi
11.   Otherwise
12.     if PTS_{T_x}[Direct Sender] = MTS_{m_k}[0] →
13.         Increment LTS_{T_x}
14.         MTS_{m_k}[T_x] = LTS_{T_x}
15.         PTS_{T_x}[Direct Sender] = MTS_{m_k}[Direct Sender]
16.         MTS_{m_k}[0] = OLDTS
17.         if T_x ≠ LCA(m_k) →
18.             OLDTS = LTS_{T_x}
19.             Forward m_k up
20.         fi
21.         Multicast m_k to cu_i
22.         return true
23.     Otherwise
24.         return false
25.     fi
26.   fi
```

}

Procedure 6.8 (CCU_MULTICASTABLE)

approach optimizes the number of extra messages on the network; however, it requires the knowledge of the existence of any TCU-TFM in the descendant hierarchy.

Therefore, the approach adopted in forwarding the messages become an optimization problem between the number of messages and the overhead of maintaining the view management.

- **TCU protocol**

  Let m_k be the message sent from p_{iy} of cu_i with $\mathcal{T}_i$ as its TFM process; im_k passes by $\mathcal{T}_x$. Let $\mathcal{T}_x$ be any TFM in the message path from its sender to any of its destinations. Let $\mathcal{TW}$ be the set of processes located between the sender of the message and $\mathcal{T}_x$ in the TW path, and let $\mathcal{S}$ be the child process of $\mathcal{T}_x$ from where the message is received. Let $\mathcal{AT}$ be the set of ancestor TFMs located between LCA(m_k) and $\mathcal{T}_x$ that have already assigned a timestamp to m_k.

  The manner in which $\mathcal{T}_x$ handles the messages varies, depending on the position of $\mathcal{T}_x$ in relation to the TW and the OW paths. If $\mathcal{T}_x$ is a member of the TW path, then two sets of procedures can be executed based on whether or not the message is on its way up or down in the hierarchy. If the message is on its way down its TW path, then procedures 6.9 and 6.10 will be executed. If the message is on its way down its OW paths, then procedures 6.11 and 6.12 will be executed. If, on the other hand, $\mathcal{T}_x$ is on its way up in its TW path, then procedures 6.15 and 6.16 will be executed.

  - *The message is on its way down.*

    Procedures 6.9 and 6.10 handle the message in the TW path. Similarly, procedures 6.11 and 6.12 handle the message in the OW path. Another version of the protocol could assume a different approach, which would

TCU_TFM_DN_TW *(m_k)* {

1. **if** *TCU_MULTICASTABLE_DN_TW(m_k)* →
2. *For each $m_t \in GWQ$,*
3. **if not** *TCU_MULTICASTABLE_DN_TW(m_t)* → *exit*
4. **Otherwise**
5. *Admit m_k to GWQ*
6. **fi**

}

Procedure 6.9 (TCU_TFM_DN_TW)

allow CCU-originated messages to keep a causal order in the TCU rather then a total order. The causal order will help in a faster delivery of the CCU message at the TCU instead of waiting for the CCU-originated message to gain the LCA timestamp. The message could be delivered at its arrival at the TCU-TFM without the LCA timestamp.

– *The message is on its way up.*

Procedures 6.15 and 6.16 handle the messages in this route. The $OLDTS$ variable is used to ensure that no lost messages arrive from the child process. The sequence cannot be tested with $MTS_{m_k}[\mathcal{S}] = PTS_{T_x}[\mathcal{S}]+1$ because $\mathcal{T}_x$ does not receive messages for which $\mathcal{S}$ is the LCA.

Step 17 in procedure 6.16 should be modified if T_x has information about the hierarchy under it. This will eliminate multicasting if the hierarchy does not have a CCU.

TCU_MULTICASTABLE_DN_TW(m_k) {

1. **if** m_k *has its LCA timestamp* $\rightarrow$
2. *Adjust* $PTS_{\mathcal{T}_x}$ *from* $TSUL_{m_k}$
3. **if** *For each* $\mathcal{T}_w$ *in* $\mathcal{AT}$, $MTS_{m_k}[\mathcal{T}_w] = PTS_{\mathcal{T}_x}[\mathcal{T}_w] + 1 \rightarrow$
4. *Remove* m_k *from* TWL
5. *Adjust* $PTS_{\mathcal{T}_x}[\mathcal{T}_x]$
6. *For each* $\mathcal{T}_w \in \mathcal{AT}$, *Adjust* $PTS_{\mathcal{T}_x}[\mathcal{T}_w]$
7. **if** *For each* m *in* $LCAM, MTS_{m_k}[\mathcal{T}_x] <$ *all* TWL *messages* $\rightarrow$
8. *Move* m *to* $CLCAM$
9. **fi**
10. *Move all messages with* $TS < MTS_{m_k}[\mathcal{T}_x]$ *from* $TSUL_{\mathcal{T}_x}$ *to* $TSUL_{m_k}$
11. *Multicast* m_k *to* cu_x
12. *return true*
13. **Otherwise**
14. *return false*
15. **fi**
16. **Otherwise**
17. *Discard* m_k
18. **fi**

}

Procedure 6.10 (TCU_MULTICASTABLE_DN_TW)

TCU_TFM_DN_OW(m_k) {

1. **if** *TCU_MULTICASTABLE_DN_OW*$(m_k) \rightarrow$
2. *For each* $m_t \in GWQ$,
3. **if not** *TCU_MULTICASTABLE_DN_OW*$(m_t) \rightarrow$ *exit*
4. **Otherwise**
5. *Admit* m_k *to GWQ*
6. **fi**

}

Procedure 6.11 (TCU_TFM_DN_OW)

6.4 MLMO Protocol Correctness

The MLMO protocol relies on running a combination of BUS and BUS-TO protocols in both the CCU and the TCU. Assume that all CUs in the communication structure are of type CCU, so that they all run the BUS protocol. Section 4.4 has shown that the BUS protocol achieves a causal order for multicasted messages. This proof can be easily extended to show that the MLMO protocol can achieve a causal order under the previously stated assumption.

Similarly, let us assume that all CUs in the structure are of type TCU, so that they all run a modified version of the BUS-TO protocol. Section 5.4 has shown that the BUS-TO protocol guarantees a total order for multicasted messages. Similarly, MLMO can achieve a total order for such setup.

Assume a general setting that combines both CCUs and TCUs within the same structure and without implementing the bypass approach. The MLMO directs the message through the communication hierarchy with the same method that BUS-TO uses, with only one difference: CCU type units are allowed in the TW path to deliver the message without waiting for the LCA timestamp. This delivery at

TCU_MULTICASTABLE_DN_OW(m_k) {

1. **if** *For each* $\mathcal{T}_w$ *in* $\mathcal{AT}$, $MTS_{m_k}[\mathcal{T}_w] = PTS_{\mathcal{T}_x}[\mathcal{T}_w] + 1$ —
2. **if** m_k *is timestamped by* $LCA(m_k)$ —
3. **if** m_k *not already timestamped by* t_x —
4. **if** *For each* $m \in CLCAM_{m_k}, PTS_{\mathcal{T}_x}[LCA(m)] \geq CLCAM_{m_k}$ —
5. *Increment* $LTS_{\mathcal{T}_x}$
6. *Add* $CLCAM_{\mathcal{T}_x}$ *to* m_k
7. $MTS_{m_k}[\mathcal{T}_x] = LTS_{\mathcal{T}_x}$
8. *Add* m_k *to* $TSUL_{\mathcal{T}_x}$
9. *Multicast* m_k *to* cu_x
10. *For each* $\mathcal{T}_w \in \mathcal{AT}$, *Adjust* $PTS_{\mathcal{T}_x}[\mathcal{T}_w]$
11. *return true*
12. **Otherwise**
13. *return false*
14. **fi**
15. **Otherwise**
16. *PROCESS_TWL*(m_k)
17. *Multicast* m_k *to* cu_x
18. *return true*
19. **fi**
20. **Otherwise**
21. *PROCESS_CLCAM*(m_k)
22. *return false*
23. **fi**
24. **Otherwise**
25. *return false*
26. **fi**

}

Procedure 6.12 (TCU_MULTICASTABLE_DN_OW)

PROCESS_TWL(m_k) {

1. $MTS_{m_k}[\mathcal{T}_x] = TWL[m_k]$
2. *Adjust* $PTS_{\mathcal{T}_x}[\mathcal{T}_x]$
3. *For each* $\mathcal{T}_w \in \mathcal{AT}$, *Adjust* $PTS_{\mathcal{T}_x}[\mathcal{T}_w]$
4. *remove* m_k *from TWL*

}

Procedure 6.13 (PROCESS_TWL)

PROCESS_CLCAM(m_k) {

1. **if** *For each* $m \in CLCAM_{m_k}, PTS_{\mathcal{T}_x}[LCA(m)] \geq CLCAM_{m_k}$ —
2. *Increment* $LTS_{\mathcal{T}_x}$
3. $MTS_{m_k}[\mathcal{T}_x] = LTS_{\mathcal{T}_x}$
4. *Insert* m_k *into TWL*
5. *Direct* m_k *to TFM ancestor of* $\mathcal{T}_x$
6. **fi**

}

Procedure 6.14 (PROCESS_CLCAM)

TCU_TFM_UP(m_k) {

1. **if** *TCU_MULTICASTABLE_UP*$(m_k) \rightarrow$
2. *For each* $m_t \in$ *Wait Queues,*
3. **if not** *TCU_MULTICASTABLE_UP*$(m_t) \rightarrow$ *exit*
4. **Otherwise**
5. *Admit message to GWQ*
6. **fi**

}

Procedure 6.15 (TCU_TFM_UP)

TCU_MULTICASTABLE_UP(m_k) {

1. **if** $MTSm_k[0] = PTS_{T_x}[\mathcal{S}] \rightarrow$
2. *Increment* LTS_{T_x}
3. $MTS_{m_k}[T_x] := LTS_{T_x}$
4. *For each* T_z *in* $\mathcal{TW} - \mathcal{S}$, *Increment* $PTS_{T_x}[T_z]$
5. $PTS_{T_x}[\mathcal{S}] := MTS_{m_k}[\mathcal{S}]$
6. **if** T_x *is not the message LCA* $\rightarrow$
7. $MTS_{m_k}[0] := OLDTS$
8. $OLDTS := LTS_{T_x}$
9. *Forward* m_x *up the tree*
10. **Otherwise**
11. **if** *TWL is empty* $\rightarrow$
12. *admit* m_k *to CLCAM*
13. **Otherwise**
14. *admit* m_k *to LCAM*
15. **fi**
16. **fi**
17. *Multicast* m_k *to* cu_x
18. **if** $m_k \in GWQ \rightarrow$
19. *Remove* m_k *from GWQ*
20. **fi**
21. *return true*
22. **Otherwise**
23. *return false*
24. **fi**

}

Procedure 6.16 (TCU_MULTICASTABLE_UP)

the CCUs is an implementation of the BUS protocol. Thus, it would not violate the causal order at these CCUs, as shown in Section 4.4. The prementioned CCUs will not pass the message to any TCU in the subtree until the LCA timestamp is received again. That is, the existence of a CCU in the message TW path will not be seen by the TCU under it. From the TCU's perspective, the entire structure contains only TCU units, which achieves a total order, as shown earlier.

For the CCUs that are not in the TW path of the message, these CUs will get the message after it has gained its LCA timestamp. The execution the BUS protocol on these messages will result in a total order and will meet the requirements of the potential causality property as required.

The addition of the bypass approach to the protocol allows the message to bypass the TCUs in the subtrees under the TW path to be delivered to the CCU units. This bypass is possible because the CCU does not need to wait for the LCA timestamp. The proofs presented in Sections 4.4 and 5.4 can be extended to show that both types of CUs will honor the required order.

6.5 Conclusion

A Multi-LAN Multi-Order protocol for multicasting in heterogeneous distributed systems has been proposed. The protocol allows group members to span different LANS and enables each group to adopt its own ordering criteria for message delivery. Our protocol relies on a hierarchical communication structure. The benefit of this structure are twofold. First, it enables the communication structure to be potentially aligned with the internet routing topology, which minimizes the number of protocol messages. Second, as a result of this alignment the protocol can exercise control on its routing scheme, which effectively decreases the actual number of multicasted messages. The protocol achieves a degree of latency, de-

pending on the ordering criteria adopted. For example, groups that adopt causal order under MLMO do not suffer unnecessary delays in message delivery because of the presence of groups that have adopted total order. The protocol performance is affected by the ratio of intra to inter group traffic; it performs better for larger ratios.

Chapter 7

INTER: A Multi-Protocol Interface

7.1 The Interface Protocol

Interoperability between the MLMO protocol and local protocols is crucial in an autonomous environment. Interoperability allows different applications with heterogeneous local protocols to multicast messages to each other. Thus, applications do not have to be rewritten in order to conform to any one multicasting protocol.

The interface is built around the assumption that different CUs, each with a different ordering criteria, can coexist. The design of the MLMO protocol allows the interface to be added as a layer between the applications and the multicasting layer. The added layer achieves an order among messages going to and from a CU, independent of the particular multicasting protocol that is running in the CU. These, local protocols can effectively handle all messages they receive as if they are local to their CU. Therefore, a local protocol can function autonomously in performing multicasting in its own CU. Note that the interface is responsible for readjusting the order between the local and global messages from one side and the

CU_x Local Protocol Order	Encapsulation	Condition
Total	ETCU	CU_x and other TCUs/ETCUs in MLMO must enforce one total order
Total	ECCU	CU_x does not want to adhere to a MLMO total order
Causal	ECCU	CU_x wants to adhere to causal order among other CUs
Causal	ETCU	not permitted

Table 7.1: Permissible Encapsulation Types

external messages from the other side to ensure a correct delivery order.

Each CU that runs a local protocol is encapsulated by the interface so that it appears to other CUs in MLMO either as a CCU (referred to as ECCU) or as a TCU (referred to as a ETCU). The type of encapsulation (ECCU or ETCU) depends on the order enforced by the local protocol, as well as on the desired relationship between the encapsulated CU and other CUs on MLMO. Table 7.1 describes the permissible encapsulation types and their corresponding conditions. Also, Figure 7.1 shows the communication structure and the encapsulation of the local protocol.

An ECCU guarantees a causal delivery of the ECCU's messages in relation to other messages that are circulating in the system, regardless of whether the local protocol enforces total or causal order. However, note that if the local protocol guarantees a total order, the interface must deliver global messages in the same order that is enforced locally. An ETCU will guarantee a total-order delivery of both the global messages of this ETCU and the global messages that originate from

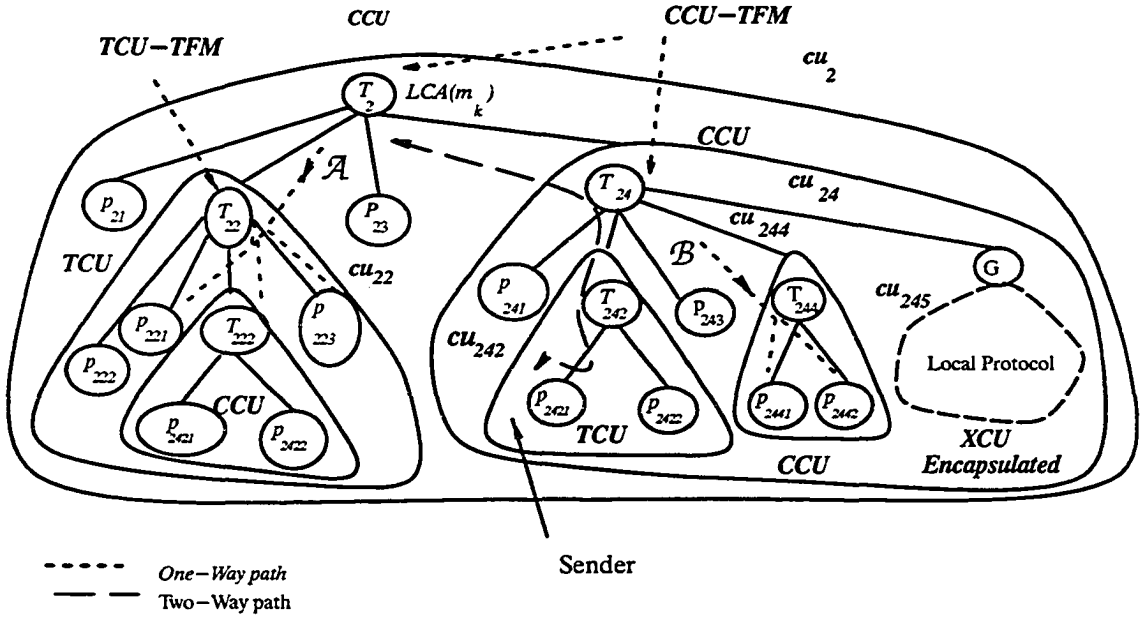


Figure 7.1: Communication structure for INTER that shows encapsulated unit.

other TCUs (i.e., external messages). The protocol will also guarantee a causal order delivery of the messages that originate from the ETCU to the CCUs on its destination list.

Figure 7.2 shows the message flow to and from the local and global side of the gateway, which constitutes the interfaces between the global and local protocols. The gateway is divided into two processes: the *Local Gateway* (G_L) and the *Global Gateway* (G_G). The G_L is added as a new node to the local multicasting group and runs the local protocol along with an interface module. The interface module ensures the enforcement of the order dictated by the local protocol upon the delivery of messages outside the group. On the other side, G_G is added as a multicasting node to the communication structure and, therefore, runs the MLMO protocol with the interface protocol to interact with G_L on one side and its TFM

in the communication hierarchy on the other side.

7.1.1 ECCU

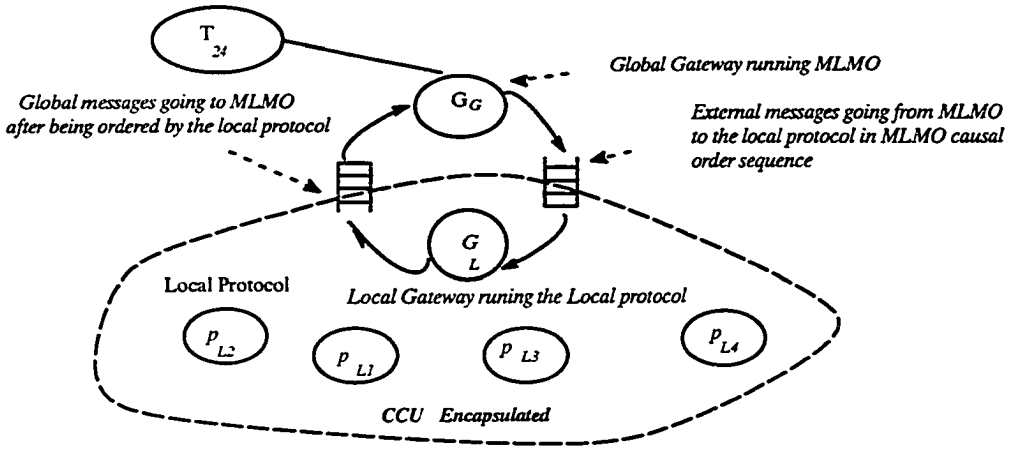
Any protocol can be encapsulated in a CCU regardless of whether the local protocol enforces a total or causal order. In this case, the external and global messages are delivered according to causal-order rules. Messages are delivered to a site that implements both a local protocol and the CCU-TFM protocol. After the local protocol makes an ordering decision and delivers the message to its group members, G_L, once it receives the message, will forward it to G_G (the global side of the gateway). Then, G_G multicasts the message passed from G_L with MLMO to enforce the order imposed by the local protocol. Because G_L sends the messages in the order dictated by the local protocol (G_G is forced to multicast these messages in the same order) this guarantees that the causal order dictated by the local protocol is not violated. The global protocol considers the ECCU to be one node G_G; therefore, all messages that leave G_G will be guaranteed the same order.

7.1.2 ETCU

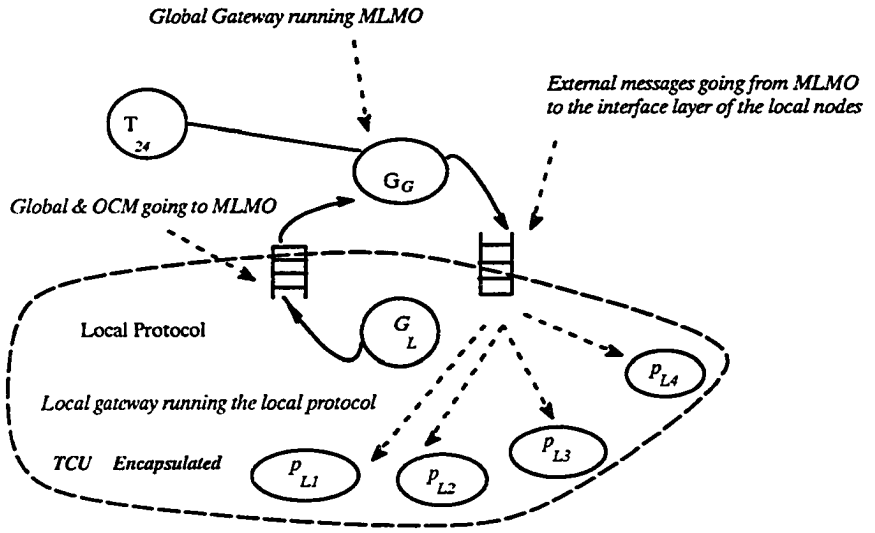
The protocol relies on the general fact that any total-order local protocol encapsulated in a TCU should use a timestamp-based protocol. This assumption is justifiable because the literature does not report any total-order protocol that currently use a different approach.

Our approach for building an interface for total ordering depends on the assignment of a timestamp that is based on the negotiation between the local and global parts of the interface. Because both local and global protocols are timestamp-based protocols, this agreed-upon timestamp can drive the entire ordering process.

The first problem that is encountered by the ETCU protocol is the different timestamp-assignment algorithms for global and local protocols (e.g., the local



(a) CCU that encapsulates local protocol.



(b) TCU that encapsulates local protocol.

Figure 7.2: Message flow between global and local agents in a gateway process.

protocol could be using the ABCAST or the TOKEN algorithm while the global maybe using MLMO). Therefore, one of the requirements for the protocol is a scheme that can map the two timestamps to achieve this ordering criteria.

The protocol relies on the two gateway processes G_L and G_G. The G_L process. as a member of the local multicasting group, upon receipt of a global message will pass it to G_G. The G_G process then tags the message as a global message and forwards it to its TFM. The MLMO protocol ensures that those messages that leave G_G are delivered to all sites in the same order. When an order is decided by the local protocol, a correction vector is sent to the LCA of the message involved. The message is blocked at its LCA until this order-correction message is received. If the messages are in the same order as originally sent from G_G, then the $LCA(m)$ releases the message and multicasts it to all nodes in its subtree. If the order differs, then the LCA will reassign the timestamp between these messages so that the message that was first will replace the one that has the smallest timestamp in the unreleased list from G_G. The following section describes in more detail incoming, outgoing, and external message handling.

7.2 Message Handling in the Interface Protocol

7.2.1 Global Message Handling

1. Assume that one of the local protocol sites multicasts a global message. The local protocol then passes the message to the local sites.
2. As the local protocol is determining a local order for the messages, a discrepancy may arise about when to forward the message to the MLMO protocol. Two alternate approaches can be used to solve this problem:

   (a) **Wait until the local order is decided and then forward the**

message to G_G.

In this approach, the protocol holds the message at G_L until a local order is agreed upon. Then, the message is forwarded to G_G so that it can be sent to its destinations outside the local CU by MLMO. The MLMO protocol provides an order for the message among the external messages and forwards the message for delivery to its members, including G_G. While MLMO obtains a global order for the message, the local protocol holds the message so that the delivery of any external messages that should be delivered before this message can occur. Different approaches can be adopted to perform this task. After this order is agreed upon, the message is released for delivery to the local nodes at ETCU. The interface protocol ensures that the local delivery of the external messages adheres to the relative order of the global messages achieved by MLMO.

(b) **Forward the message and correct the order later.**

While the negotiation is underway for determining the local order, G_L forwards the message to G_G. The G_G process timestamps the message, tags it as a global message, and then forwards it to its TFM.

As the local and global protocols try to achieve an order for global messages, MLMO ensures that messages going out of G_G are delivered to all sites with the same order. A problem occurs because the order of delivery from the gateway to the global protocol of the message is not the final order. This order changes, depending on the local protocol negotiation. For example, two global messages m_1 and m_2 are sent from the gateway to the global protocol in the order m_1, m_2. From the global protocol perspective, m_1 has to be delivered before m_2 to honor Lamport's rule 1 because from the point of view of the global

protocol both m_1 and m_2 originate from the same site (G_G). However, the possibility exists that m_1 and m_2 are from different local sites, so that the final local order could be m_2, m_1. As a result, the global order and the local order disagree from the TCU point of view.

To overcome this problem without having to wait for local order to be achieved before the global order, a new message is introduced. This message is the *Order Correction Message* (OCM), which corrects this ordering problem. In this case, the global message will wait at its LCA until the OCM is received. After a local order is predicted, a correction vector is sent to the LCA of the message involved. When OCM is received by the message's LCA, one of two situations may occur.

i. **The messages are in the right order as originally sent from G_G.**

   The $LCA(m)$ will then release the message and multicast it to all descendants. The G_G process, upon receiving the message, will pass it back to G_L to update the timestamp vector, which will be used later with external messages.

ii. **The messages are not in the right order.**

   The $LCA(m)$ will reassign the timestamp between these messages such that the message that was agreed upon to be first will replace the one from G_G that has the smallest timestamp in the unreleased list. The order will be affected in the TCU but will not be affected in any of the CCUs that have already delivered the message. The message reassignment (or, more appropriately, incarnation) will switch the data content in one message with the one in the other; the other timestamp data will remain unaffected. As a result, the reassignment will be transparent to all other nodes in the TW paths of

the message. The TFMs in the TW path must use the new data contents of the message for multicasting.

In case (*a*), a problem occurs in regard to the external messages because the delivery of the local message must wait for the LCA timestamp to be given. As a result, *Total delay = Delay of local protocol order + Delay of global protocol.* In case (*b*), *Total delay = max(both protocols).* However, we adopt the approach presented in case (a) to order ECCU messages because the delay incurred in determining a causal order is minor in comparison with the time needed to send the OCM. The approach presented for case (b) is used in ETCU message ordering.

3. After the order agreement between the global and local protocols is reached, the local delivery of the message must be performed. Two approaches can achieve this delivery to conform to the agreed-upon order.

   (a) **Ask the local protocol to carry the delivery.**

   In this approach, the delivery is achieved through G_L, which is a member of the local group. In order for G_L to achieve message mixing between both global and external messages, the local protocol must be lead to this ordering decision indirectly through a sequence of timestamp assignments. This approach requires that the interface be dependent on the local protocol. For example, in the case of ABCAST [12], assume that we have a global message m_1 and an external message m_2 and that MLMO has decided to deliver m_2 before m_1. Also assume that m_1 was received by G_L before m_2. If G_L delays the assignment of the G_L timestamp until the global protocol provides an order between m_1 and m_2, then G_L can enforce this order by assigning a higher value timestamp to m_1. This timestamp is higher than the timestamps of the other nodes for m_1 and higher than the expected timestamp assigned by

all nodes to m_2. This timestamp assignment results in a final timestamp for m_1 that is larger than the final timestamp for m_2, which forces a local delivery of m_2 before m_1. This approach provides more autonomy than the second one (see (b)). However, this approach may not be feasible with the local multicasting protocols. We adopt this protocol in ECCU because the causal order allows us to rely on the G_L as the single source for multicasting external messages to the local nodes. Achievement of a causal order in a similar manner is possible because the local protocol provides a single-source ordering. Therefore, whatever order G_L uses to multicast these messages will be honored by the local protocol.

(b) **Achieve the delivery through the interface layer.**

In this approach, G_G sends the external message to the interface layer that resides between the network layer that runs the local multicasting protocol and the application layer at each local node in the ETCU. This layer, after it receives the external message, as well as the global and local messages, will insert the external message in the delivery queue ordered by the local protocol in the same order provided by MLMO. This process will allow the delivery order to be achieved in accordance with the global order without violation of the local order. This approach is adopted in the ETCU message delivery because the global message is forwarded to the global protocol before a final order is agreed upon locally.

7.2.2 Local Message Handling

The local protocol is responsible for achieving the local order and for reliability in delivering local messages. Thus, our protocol task is to honor this order and to merge the stream of local and global messages delivered from the local protocol

with the external messages. As previously mentioned, a new node is added to the local group; the gateway local agent G_L. The main function of this node is to serve as a link between the local and the global protocol; G_L passes the global messages from the local group to the gateway global agent G_G, and G_L runs the local protocol and the gateway procedure.

- Assume that one of the local protocol sites multicasts a local message m_k. The local protocol begins to deliver m_k to the members of the group, including the gateway (G_L).

- The local protocol attempts to deliver m_k, and the message is intercepted by the interface sublayer that resides between the multicast sublayer and the application layer. This new sublayer does not change the order achieved by the local protocol but assists in merging the external messages with both the local and global messages.

7.2.3 External Message Handling

External message order is determined by the global protocol MLMO. The gateway global agent G_G receives the message from MLMO with the MLMO receive procedure (described in Section 6.3) and then forwards it, along with its timestamp structure, to the interface layer.

- Assume that an external message m_k, which targets the local group, is delivered at G_G after receiving m_k; G_G schedules it for delivery and begins to to deliver m_k to the members of the group, including the gateway G_L.

- The interface sublayer at the local group members, upon receipt of the message, checks for delivery of global messages that were scheduled for global delivery prior to m_k under MLMO. The message m_k waits for the delivery of

these messages (and all the local messages that were scheduled for delivery prior to these global messages) by the local protocol.

7.2.4 Order Correction Message (OCM)

As mentioned in Section 7.2.1, the order correction message is responsible for adjusting the order of outgoing message if it differs from the initial order of delivery of these messages to the global protocol (MLMO). For example, suppose m_1 and m_2 are delivered from the local to the global protocol in the order stated before and that the local protocol has determined its order to be m_2, m_1. If m_1 is delivered to the global protocol before m_2, then the global protocol ensures that m_1 will be delivered before m_2. To readjust this order, the order correction message is sent to the LCA of m_1 and m_2 to indicate the new order. The order correction message causes a timestamp exchange between m_1 and m_2 in all the TCU-TFMs visited by m_1 and m_2.

The timestamp switching process can be performed based on one of two approaches: *message id switching* and *timestamp switching.*

7.2.5 Multi-Protocol Interface

This interface runs on each local site that is involved in multicasting. The function of this layer, which is located between the local protocol and the application layer, is to insert the external messages among the global and local messages in accordance with the order determined by MLMO. This layer is necessary because the enforcement of an order on the external messages in relation to the global messages must be performed in general by a higher layer than the local protocol. Also, the enforcement of this order by the local protocol undermines the autonomy of the local protocol. In addition, the interface must be customized to accommodate each local protocol. No assumption is made on how the CCU and the TCU are added

to the communication structure; they can be in any distribution. This enables a possible extension to this protocol to allow dynamic ordering by allowing a CU to change between a TCU or a CCU.

7.3 Protocol Outline

Five types of procedures handle the messages: the *Sender*, the *Receiver*, the *TFM*, the *Gateway*, and the *Interface* procedures. In this section, we focus on the procedures that are specific to the interface protocol, which include both the gateway and the interface procedures. The procedures that are similar to the ones mentioned in the MLMO description in Chapter 6 will not be repeated.

7.3.1 Sender

Let m_k be a message sent from p_{iy} of cu_i with $\mathcal{T}_i$ as its TFM; m_k is received by a process p_{jx} of cu_j with $\mathcal{T}_j$ as its TFM process. The message m_k is timestamped at p_{iy}, and the message timestamp vector is adjusted by setting both $MTS_{m_k}[p_{iy}]$ and $MTS_{m_k}[0]$; $MTS_{m_k}[0]$ always carries the last message timestamp sent from a process (a TFM or a cooperative process) to its higher level TFM to ensure that no messages are out of order (see Section 6.3.3 for more details).

The sender process will follow different procedures, depending on whether it is a local site, a gateway, or a process that is running the MLMO protocol. If p_{iy} is a member of the local group, then p_{iy} will use the local protocol procedure to send and receive messages. If p_{iy} is a gateway local agent (G_L), then p_{iy} will use procedure 7.1 to send m_k. If p_{iy} is a gateway global agent (G_G), then p_{iy} will use procedure 7.2 to send the messages passed to it from G_L if the message is a global message or an OCM. If p_{iy} is running the MLMO protocol, then p_{iy} will execute procedure 6.1.

SENDER-GATEWAY $(m_k)\{$

1. *Increment* $LTS_{p_{iy}}$
2. $MTS_{m_k}[p_{iy}] = LTS_{p_{iy}}$
3. *Deliver* m_k *to the local protocol*

$\}$

Procedure 7.1 (Local Gateway Agent Sender)

SENDER $(m_k)\{$

1. $MTS_{m_k}[0] = LTS_{p_{iy}}$
2. *Increment* $LTS_{p_{iy}}$
3. $MTS_{m_k}[p_{iy}] = LTS_{p_{iy}}$
4. *Send message to* $\mathcal{T}_i$

$\}$

Procedure 7.2 (Sender)

7.3.2 Receiver

The receiver process will run one of two sets of procedures, depending on whether it is a member of a TCU or a CCU. If the receiver process p_{jx} is a member of a CCU, then the CCU_RECEIVE and CCU_DELIVERABLE procedures will be executed. If p_{jx} is a member of a TCU, then the procedures to be executed will vary depending on the position of p_{jx} in relation to the TW and the OW paths. If p_{jx} is a member of the TW path, then procedures TCU_RECEIVE_TW and TCU_DELIVERABLE_TW will be executed. If, on the other hand, p_{jx} is a member of the OW paths, then procedures TCU_RECEIVE_OW and TCU_DELIVERABLE_OW will be executed. If p_{jx} is a gateway global agent (G_G), then it will execute a set of procedures based on whether it is an agent for an ETCU or an ECCU. If G_G is a gateway for an ECCU, then procedure ECCU_RECEIVE_GLOBAL_GATEWAY and ECCU_DELIVERABLE_GLOBAL_GATEWAY will be executed. If G_G is a gateway for an ETCU, then the procedures to be executed will vary between procedures ETCU_RECEIVE_GATEWAY_OW and ETCU_DELIVERABLE_GATEWAY_OW if G_G is in the message OW path or procedure ETCU_RECEIVE_GATEWAY_TW and procedure ETCU_DELIVERABLE_GATEWAY_TW if G_G is in the TW path.

- **CCU protocol**

  If p_{jx} is a process that runs the MLMO protocol, then procedures 6.2 and 6.3 in Chapter 6 will be executed. If p_{jx} is a G_G for an ECCU, then procedures 7.3 and 7.4 will be executed. When G_G decides to deliver the message, it will actually pass it to the gateway local agent (G_L); G_L will then execute the Inter-Sender procedure listed in procedure 7.1. The traffic between G_G and G_L is controlled with a timestamp scheme that provides a more reliable communication between them. The use of a timestamp scheme between G_G

ECCU_RECEIVE_GLOBAL_GATEWAY(m_k) {

1. **if** *CCU_DELIVERABLE*$(m_k) \rightarrow$
2. *For each* $m_t \in LWQ$,
3. **if not** *CCU_DELIVERABLE*$(m_t) \rightarrow$ *exit*
4. **Otherwise**
5. **if** $MTS_{m_k}[\mathcal{T}_j] > PTS_{p_{jx}}[\mathcal{T}_j] \rightarrow$
6. *Admit* m_k *to LWQ*
7. **Otherwise**
8. *Discard* m_k
9. **fi**
10. **fi**

}

Procedure 7.3 (ECCU_RECEIVE_GLOBAL_GATEWAY)

and G_L also allows both processes to be allocated at different sites, which increases the protocol resiliency and decreases the recovery cost.

- **TCU protocol**

  As mentioned previously, the gateway uses two different handling procedures, depending upon whether the message is in one of its OW paths or its TW path; $\mathcal{OW}$ is again the set of processes located between $LCA(m_k)$ and $\mathcal{T}_j$ that belongs to the OW path, and $\mathcal{TW}$ is the set of processes between $\mathcal{T}_j$ and $LCA(m_k)$ that belongs to the TW path. Note that $\mathcal{TW}$ could be empty if all paths between $\mathcal{T}_j$ and LCA belong to the OW path.

  - *The OW path module.*

    The OW path module will follow the steps outlined in procedures 7.5 and 7.6. The G_G process does not forward the message to G_L, instead it is directed to the interface layer at the local nodes. This approach

ECCU_DELIVERABLE_GLOBAL_GATEWAY (m_k) {

```
1.    if (MTS_{m_k}[T_j] = PTS_{p_jx}[T_j] + 1) →
2.        Increment PTS_{p_jx}[T_j]
3.        Forward m_k to G_L
4.        if m_k ∈ LWQ →
5.            Remove m_k from LWQ
6.        fi
7.        return true
8.    Otherwise
9.        return false
10.   fi
```

}

Procedure 7.4 (ECCU_DELIVERABLE_GLOBAL_GATEWAY)

ETCU_RECEIVE_GATEWAY_OW (m_k) {

```
1. if m_k already passed by LCA(m_k) →
2.    if TCU_DELIVERABLE_OW(m_k) →
3.        For each m_t ∈ GWQ, if not TCU_DELIVERABLE_OW(m_t) → exit
4.    Otherwise
5.        Admit m_k to GWQ
6.    fi
7. Otherwise
8.    Discard m_k
9. fi
```

}

Procedure 7.5 (ETCU_RECEIVE_GATEWAY_OW)

ETCU_DELIVERABLE_GATEWAY_OW(m_k) {

1. **if** *For each* $\mathcal{T}_w$ *in* $\mathcal{TW}, PTS_{p_{jx}}[\mathcal{T}_w] = MTS_{m_k}[\mathcal{T}_z] + 1$
2. **AND**
3. *For each* $m \in LCAM_{m_k}, PTS_{p_{jx}}[LCA(m)] \geq CLCAM[m] \rightarrow$
4. *Multicast* m_k *to the local nodes interface*
5. *For each* $\mathcal{T}_w \in \mathcal{TW}$, *Increment* $PTS_{p_{jx}}[\mathcal{T}_w]$
6. **if** $m_k \in GWQ \rightarrow$
7. *Remove* m_k *from GWQ*
8. **fi**
9. *return true*
10. **Otherwise**
11. *return false*
12. **fi**

}

Procedure 7.6 (ETCU_DELIVERABLE_GATEWAY_OW)

(Section 7.2.3) is provided to allow the local nodes to merge the global, external, and local messages without making changes to the local protocol. The interface layer, after the receipt of the message, will execute procedure ??.

- *The TW path module.*

  The ETCU_RECEIVE_GATEWAY_TW will follow the same procedure as ETCU_RECEIVE_GATEWAY_OW presented before with one modification: ETCU_DELIVERABLE_GATEWAY_TW will be called instead of ETCU_DELIVERABLE_GATEWAY_OW. Procedure ETCU_DELIVERABLE_GATEWAY_TW provides the necessary steps for handling the *Timestamp Updater List (TSUL)*, which was intro-

ETCU_DELIVERABLE_GATEWAY_TW(m_k) {

1. **if** *For each* $\mathcal{T}_z$ *in* $TW, MTS_{m_k}[\mathcal{T}_z] = PTS_{p_{jx}}[\mathcal{T}_z] + 1 \rightarrow$
2. *Multicast* m_k *to the local nodes interface*
3. *For each* $\mathcal{T}_z$ *in* TW, *Increment* $PTS_{p_{jx}}[\mathcal{T}_z]$
4. *Update* $PTS_{p_{jx}}$ *from* $TSUL_{m_k}$
5. **if** $m_k \in GWQ$ *or* $m_k \in LWQ \rightarrow$
6. *Remove* m_k *from Wait Queue*
7. **fi**
8. *return true*
9. **Otherwise**
10. *return false*
11. **fi**

}

Procedure 7.7 (ETCU_DELIVERABLE_GATEWAY_TW)

duced in Section 6.2.3 to solve the timestamp gap problem.

7.3.3 TFM Procedure

The TFM processes will run one of two procedures, depending on whether it is a member of a TCU or a CCU. If the TFM process $\mathcal{T}_x$ is a member of a CCU, then the INTER_CCU_TFM, CCU_TFM, and CCU_MULTICASTABLE procedures will be executed. If $\mathcal{T}_x$ is a member of a TCU, then the procedures to be executed will vary, based on the position of $\mathcal{T}_x$ in relation to the TW and OW paths. If $\mathcal{T}_x$ is a member of the TW path, then two sets of procedures can be executed, based on whether the message is on its way *up* or *down* in the hierarchy. If the message is on its way down, then procedures TCU_TFM_DN_TW and TCU_MULTICASTABLE_DN_TW will be executed. If the message is on its

way up, then procedures INTER_TCU_TFM_UP_TW, TCU_TFM_UP_TW, and TCU_MULTICASTABLE_UP_TW will be executed. If, on the other hand, $\mathcal{T}_x$ is a member of one of the OW paths, then procedures TCU_TFM_DN_OW and TCU_MULTICASTABLE_DN_OW will be executed.

- **CCU protocol**

  Let $\mathcal{T}_x$ be any TFM in the message path from its sender to any of its destinations, and $\mathcal{L}$ be the least common ancestor of cu_i and cu_j (LCACU(cu_i, cu_j)). Let m_k be the message sent from p_{iy} of cu_i with $\mathcal{T}_i$ as its TFM process; m_k passes by $\mathcal{T}_x$. Let $\mathcal{OW}$ be the set of TFMs processes in the OW path between $\mathcal{T}_x$ and the LCA of m_k. Procedures 7.8, 7.9, and 7.10 will be executed by $\mathcal{T}_x$.

- **TCU protocol**

  Let m_k be the message sent from p_{iy} of cu_i with $\mathcal{T}_i$ as its TFM process; m_k passes by $\mathcal{T}_x$. Let $\mathcal{T}_x$ be any TFM in the message path from its sender to any of its destinations. Let $\mathcal{TW}$ be the set of processes located between the sender of the message and $\mathcal{T}_x$ in the TW path, and $\mathcal{S}$ be the child process of $\mathcal{T}_x$ from where the message is received. Let $\mathcal{AT}$ be the set of ancestor TFMs located between $LCA(m_k)$ and $\mathcal{T}_x$ that have already assigned a timestamp to m_k.

  The method with which $\mathcal{T}_x$ handles the messages varies, depending on the position of $\mathcal{T}_x$ in relation to the TW and the OW paths. If $\mathcal{T}_x$ is a member of the TW path, then two sets of procedures can be executed, based on whether the message is on its way *up* or *down* in the hierarchy. If the message is on its way down the TW path, then procedures 7.11 and 7.12 will be executed. If the message is on its way down its OW paths, then procedures 7.13 and 7.17 will be executed. If, on the other hand, $\mathcal{T}_x$ is on its way up in its TW path, then procedures 7.18, 7.16, and 7.19 will be executed.

```
INTER_CCU_TFM(m_k) {
 1.   if OCM or GLOBAL message →
 2.      if OCM message →
 3.         Timestamp m_k using OCM timestamp structure
 4.        if T_x = LCA(m_k) →
 5.           Message_Timestamp_Switch(m_k)
 6.        Otherwise
 7.           Forward m_k up
 8.        fi
 9.      Otherwise
10.        if m_k is going up →
11.           Timestamp m_k
12.           Admit m_k to OCMHOLD
13.        Otherwise
14.           if m_k is timestamped by LCA(m_k) →
15.              If timestamp switching is needed →
16.                 Update messages in LWQ and GWQ
17.              Remove message from OCMHOLD
18.              CCU_TFM(m_k)
19.           fi
20.        fi
21.     fi
22.  Otherwise
23.        CCU_TFM(m_k)
24.  fi
}
```

Procedure 7.8 (INTER_CCU_TFM)

```
CCU_TFM(m_k) {
    if m_k has not been timestamped by T_x →
 1.     if CCU_MULTICASTABLE(m_k) →
 2.         For each m_t ∈ LWQ,
 3.             if not CCU_MULTICASTABLE(m_t) → exit
 4.                 if not CCU_MULTICASTABLE(m_t) → exit
 5.             Otherwise
 6.                 Admit m_k to LWQ
 7.             fi
 8.     Otherwise
 9.             Direct m_k to the descendant TFMs
10.     fi

}
```

Procedure 7.9 (CCU_TFM)

- *The message is on its way down.*

  Procedures 7.11 and 7.12 will handle the message in the TW path. Similarly, procedures 7.13 and 7.17 will handle the message in the OW path.

- *The message is on its way up.*

  Procedures 7.18, 7.16, and 7.19 handle the messages on this route. The $OLDTS$ variable is used to ensure that no lost message arrives from the child process because the sequence cannot be tested with $MTS_{m_k}[\mathcal{S}] = PTS_{\mathcal{T}_x}[\mathcal{S}] + 1$. It cannot be tested because $\mathcal{T}_x$ does not receive the messages that have $\mathcal{S}$ as the LCA.

CCU_MULTICASTABLE(m_k) {

```
1.   if m_k was on its way down →
2.     if MTS_{m_k}[Direct Sender] = PTS_{T_x}[Direct Sender] + 1 →
3.         Increment PTS_{T_x}[Direct Sender]
4.         Increment LTS_{T_x}
5.         MTS_{m_k}[T_x] = LTS_{T_x}
6.        Multicast m_k to cu_i
7.         return true
8.     Otherwise
9.         return false
10.    fi
11.  Otherwise
12.    if  PTS_{T_x}[Direct Sender] = MTS_{m_k}[0] →
13.        Increment LTS_{T_x}
14.        MTS_{m_k}[T_x] = LTS_{T_x}
15.        PTS_{T_x}[Direct Sender] = MTS_{m_k}[Direct Sender]
16.        MTS_{m_k}[0] = OLDTS
17.        if T_x ≠ LCA(m_k) →
18.            OLDTS = LTS_{T_x}
19.            Forward m_k up
20.        fi
21.       Multicast m_k to cu_i
22.        return true
23.    Otherwise
24.        return false
25.    fi
26.  fi
```

}

Procedure 7.10 (CCU_MULTICASTABLE)

TCU_TFM_DN_TW(m_k) {

1. **if** *TCU_MULTICASTABLE_DN_TW*$(m_k) \rightarrow$
2. *For each* $m_t \in GWQ$,
3. **if not** *TCU_MULTICASTABLE_DN_TW*$(m_t) \rightarrow$ *exit*
4. **Otherwise**
5. *Admit* m_k *to GWQ*
6. **fi**

}

Procedure 7.11 (TCU_TFM_DN_TW)

7.4 Conclusion

In this chapter, we presented a new approach for allowing interoperability between our protocol suite and different existing multicasting protocols. We have presented INTER as an interface built around the assumption that different CUs, each with a different ordering criteria, can coexist. INTER is added as a layer between the applications and the multicasting layer. The added layer achieves an order among messages going to and from a CU, independent of the particular multicasting protocol that is running. These local protocols can effectively handle all messages they receive as if they are local to their CU. Therefore, a local protocol can function autonomously in performing multicasting in its own CU. The interface is responsible for readjusting the order between the local and global messages from one side and the external messages from the other side to ensure a correct delivery order.

This order, being causal or total, is selected based on the application needs. A notable advantage to our approach is the ability to accommodate the coexistence of multiple heterogeneous intra-group multicasting protocols. Specifically, an encapsulation protocol is described that effectively acts as an interface that connects

TCU_MULTICASTABLE_DN_TW(m_k) {

1. **if** m_k *has its LCA timestamp* $\rightarrow$
2. *Adjust* $PTS_{\mathcal{T}_x}$ *from* $TSUL_{m_k}$
3. **if** *For each* $\mathcal{T}_w$ *in* $\mathcal{AT}$, $MTS_{m_k}[\mathcal{T}_w] = PTS_{\mathcal{T}_x}[\mathcal{T}_w] + 1 \rightarrow$
4. *Remove* m_k *from* TWL
5. *Adjust* $PTS_{\mathcal{T}_x}[\mathcal{T}_x]$
6. *For each* $\mathcal{T}_w \in \mathcal{AT}$, *Adjust* $PTS_{\mathcal{T}_x}[\mathcal{T}_w]$
7. **if** *For each* m *in* $LCAM$, $MTS_{m_k}[\mathcal{T}_x] <$ *all* TWL *messages* $\rightarrow$
8. *Move* m *to* $CLCAM$
9. **fi**
10. *Move all messages with* $TS < MTS_{m_k}[\mathcal{T}_x]$ *from* $TSUL_{\mathcal{T}_x}$ *to* $TSUL_{m_k}$
11. *Multicast* m_k *to* cu_x
12. *return true*
13. **Otherwise**
14. *return false*
15. **fi**
16. **Otherwise**
17. *Discard* m_k
18. **fi**

}

Procedure 7.12 (TCU_MULTICASTABLE_DN_TW)

TCU_TFM_DN_OW(m_k) {

1. **if** *TCU_MULTICASTABLE_DN_OW*$(m_k) \rightarrow$
2. *For each* $m_t \in GWQ$,
3. **if not** *TCU_MULTICASTABLE_DN_OW*$(m_t) \rightarrow$ *exit*
4. **Otherwise**
5. *Admit* m_k *to GWQ*
6. **fi**

}

Procedure 7.13 (TCU_TFM_DN_OW)

PROCESS_TWL(m_k) {

1. $MTS_{m_k}[\mathcal{T}_x] = TWL[m_k]$
2. *Adjust* $PTS_{\mathcal{T}_x}[\mathcal{T}_x]$
3. *For each* $\mathcal{T}_w \in \mathcal{AT}$, *Adjust* $PTS_{\mathcal{T}_x}[\mathcal{T}_w]$
4. *remove* m_k *from TWL*

}

Procedure 7.14 (PROCESS_TWL)

PROCESS_CLCAM(m_k) {

1. **if** *For each* $m \in CLCAM_{m_k}, PTS_{\mathcal{T}_x}[LCA(m)] \geq CLCAM_{m_k} \rightarrow$
2. *Increment* $LTS_{\mathcal{T}_x}$
3. $MTS_{m_k}[\mathcal{T}_x] = LTS_{\mathcal{T}_x}$
4. *Insert* m_k *into TWL*
5. *Direct* m_k *to TFM ancestor of* $\mathcal{T}_x$
6. **fi**

}

Procedure 7.15 (PROCESS_CLCAM)

TCU_TFM_UP(m_k) {

1. **if** *TCU_MULTICASTABLE_UP*$(m_k) \rightarrow$
2. *For each* $m_t \in$ *Wait Queues,*
3. **if not** *TCU_MULTICASTABLE_UP*$(m_t) \rightarrow$ *exit*
4. **Otherwise**
5. *Admit message to GWQ*
6. **fi**

}

Procedure 7.16 (TCU_TFM_UP)

any protocol that performs multicasting in a process group to the MLMO protocol. An added feature to our approach is its elimination of any need to alter the local multicasting protocols. This enables the MLMO protocol to achieve interoperability of multiple intra-group multicasting protocol, such that full autonomy is upheld.

TCU_MULTICASTABLE_DN_OW(m_k) {

1. **if** *For each* $\mathcal{T}_w$ *in* $\mathcal{AT}$, $MTS_{m_k}[\mathcal{T}_w] = PTS_{\mathcal{T}_x}[\mathcal{T}_w] + 1$ →
2. **if** m_k *is timestamped by* $LCA(m_k)$ →
3. **if** m_k *is not already timestamped by* t_x →
4. **if** *For each* $m \in CLCAM_{m_k}$, $PTS_{\mathcal{T}_x}[LCA(m)] \geq CLCAM_{m_k}$ →
5. *Increment* $LTS_{\mathcal{T}_x}$
6. *Add* $CLCAM_{\mathcal{T}_x}$ *to* m_k
7. $MTS_{m_k}[\mathcal{T}_x] = LTS_{\mathcal{T}_x}$
8. *Add* m_k *to* $TSUL_{\mathcal{T}_x}$
9. *Multicast* m_k *to* cu_x
10. *For each* $\mathcal{T}_w \in \mathcal{AT}$, *Adjust* $PTS_{\mathcal{T}_x}[\mathcal{T}_w]$
11. *return true*
12. **Otherwise**
13. *return false*
14. **fi**
15. **Otherwise**
16. *PROCESS_TWL*(m_k)
17. *Multicast* m_k *to* cu_x
18. *return true*
19. **fi**
20. **Otherwise**
21. *PROCESS_CLCAM*(m_k)
22. *return false*
23. **fi**
24. **Otherwise**
25. *return false*
26. **fi**

}

Procedure 7.17 (TCU_MULTICASTABLE_DN_OW)

INTER_TCU_TFM_UP (m_k) {

```
1.   if OCM OR (GLOBAL message AND T_x = LCA(m_k)) →
2.       if OCM message →
3.         if Missing message
4.             Add to missing OCM list
5.             Send retransmission request
6.         fi
7.          Timestamp m_k using OCM timestamp structure
8.         if T_x = LCA(m_k) →
9.             Message_Timestamp_Switch(m_k)
10.            For each m_t ∈ OCMHOLD, if m_t ∈ OCM →
11.                Remove m_t from OCMHOLD
12.                TCU_TFM_UP(m_t)
13.        Otherwise
14.            Forward m_k up
15.        fi
16.      Otherwise
17.            Timestamp m_k
18.            Admit m_k to OCMHOLD
19.      fi
20.  Otherwise
21.       TCU_TFM_UP(m_k)
22.  fi
}
```

Procedure 7.18 (INTER_TCU_TFM_UP)

TCU_MULTICASTABLE_UP(m_k) {

1. **if** $MTSm_k[0] = PTS_{\mathcal{T}_x}[\mathcal{S}] \rightarrow$
2. *Increment* $LTS_{\mathcal{T}_x}$
3. $MTS_{m_k}[\mathcal{T}_x] := LTS_{\mathcal{T}_x}$
4. *For each* $\mathcal{T}_z$ *in* $\mathcal{TW} - \mathcal{S}$, *Increment* $PTS_{\mathcal{T}_x}[\mathcal{T}_z]$
5. $PTS_{\mathcal{T}_x}[\mathcal{S}] := MTS_{m_k}[\mathcal{S}]$
6. **if** $\mathcal{T}_x$ *is not the message LCA* $\rightarrow$
7. $MTS_{m_k}[0] := OLDTS$
8. $OLDTS := LTS_{\mathcal{T}_x}$
9. *Forward* m_x *up the tree*
10. **Otherwise**
11. **if** *TWL is empty* $\rightarrow$
12. *admit* m_k *to CLCAM*
13. **Otherwise**
14. *admit* m_k *to LCAM*
15. **fi**
16. **fi**
17. *Multicast* m_k *to* cu_x
18. **if** $m_k \in GWQ \rightarrow$
19. *Remove* m_k *from GWQ*
20. **fi**
21. *return true*
22. **Otherwise**
23. *return false*
24. **fi**

}

Procedure 7.19 (TCU_MULTICASTABLE_UP)

Chapter 8

Performance Issues

8.1 Introduction

Performance is a critical measure of practicality of our approach. To determine the practicality of using any of our protocols, we must provide some performance figures to show how well the protocols perform. Several approaches can be used to conduct our performance study, including the development of an analytical or a simulation model and the use of prototyping. We considered each of these options at the outset of this performance study. Prototyping is a costly approach; we could not justify the allocation of enough resources without providing some preliminary evidence of the effectiveness of the new protocols. We also considered the development of a simulation model for the new set of protocols. Several approaches were evaluated for this option, such as the development of a simulator with one of the existing simulation languages or the use of one of the network simulation environments like NETSIM from MIT [46] or MaRS from the University of Maryland [5]. The problem we encountered with the use of simulation was the lack of any performance study reported in the literature based on a simulation model under a similar environment; as a result, we would need to develop a simulation

model for each protocol to be used in our evaluation to provide a fair basis of comparison. The development alone would require a massive effort, in addition to the significant amount of simulation time required to get a confident interval.

We then investigated the possibility of developing an analytical model to evaluate our protocols. The main advantage of an analytical model is that the computer processing time required to solve it is much smaller than that required for simulation. We first considered both queueing network models [25] and markov chain models [75] for performance evaluation. For queueing network models, a class of these models has a product form that provides an efficient solution. We encountered a problem in the analysis of our model because of the intractability of the network model presented. This intractability is caused by a multiple resource possession problem and state-dependent service rates [48, 49, 55]. As a result, the queuing network model violates the product form requirements. These problems can be solved with a markov chain model. The markov chain model is a good method for representing system behavior in blocking and conflict situations [49, 74]. However, a drawback to the markov model is the exponential growth in the number of states that can exist in representing a real system like this, which makes the solution infeasible [64]. We concluded from our investigation that both the queueing network and markov chain models were not applicable in our case.

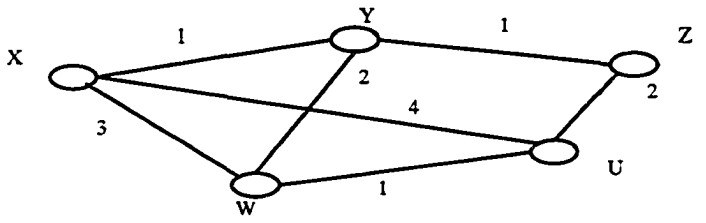
We finally decided to use a simplified deterministic model to evaluate the performance of the system. Two models were considered; a point-to-point network and a multicast network. We are using the models presented by Garcia-Molina and Spauster in reference [41]. The models are macro level models and are intended only for identifying trends and major performance factors. The goal is not to predict exactly how the protocols perform on a given system. To do this, we would need to provide a more complex model that incorporates network topology, congestion, routing, acknowledgment, and failures. The model does not present

all of these factors, but as we will see, it is still capable of providing an index of the major differences in performance among the different protocols.

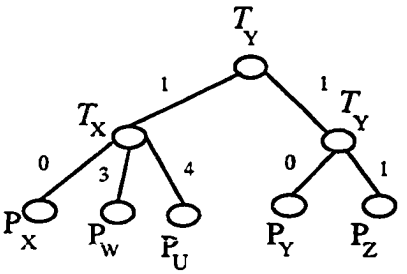
We focus on three main performance indexes: N, the number of messages required to send a multicast; D, the time elapsed between the beginning of the ordered multicast and the time when all members of the multicast destination group deliver the message; and O, the extra overhead on the sites to deliver the message. Here, we compare our hierarchical approach to the two-phase protocol of Birman and Joseph [12] and to a centralized version of Chang and Maxemchuck [19].

8.2 Examples

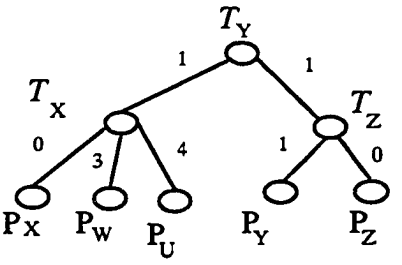
In order to get a preliminary idea in regard to the performance of the protocols, we took a simple example of five nodes involved in the multicasting process with the following processes $\{p_U, p_W, p_X, p_Y, p_Z\}$. The multicasting processes are divided into two groups: $g_1=\{p_U, p_W, p_X\}$ and $g_2=\{p_Y, p_Z\}$. To perform an initial comparison, we assumed three different connectivity charts between these five nodes with different communication costs. We built two different communication structures for each example to see the effect of the structure on the protocol behavior. The communication structure in structure A used the same process site for running multiple TFMs; structure B used one site to run only a single TFM. Structure A had less resiliency to the TFM's site failure and a higher cost of recovery from such failure. We checked the ABCAST and CBCAST for each of the different examples, and we checked a centralized version that relies on receiving the timestamp from a central node and broadcasting it back to the multicasting group. The ABCAST is a three-round protocol in which the sender sends a message to all members and then waits to receive a timestamp back from all the nodes. From



(a) Connected sites with communication cost.



(b) Communication Structure A.



(c) Communication Structure B.

Figure 8.1: Example 1 - a set of connected sites and possible communication structures.

these returning timestamps, it calculates the final timestamp and sends it back to all members. Therefore, in calculating the delay encountered in message delivery under ABCAST, we note that it will vary between 2*{the shortest path between the sender and the farthest process in the group}, which is the cost for delivery at sender, and 3*{the shortest path between the sender and the farthest process in the group}, which is the cost of delivery at this site. For example, consider table 8.1 and figure 8.1, with the assumption that p_X wants to send a message m to the group.

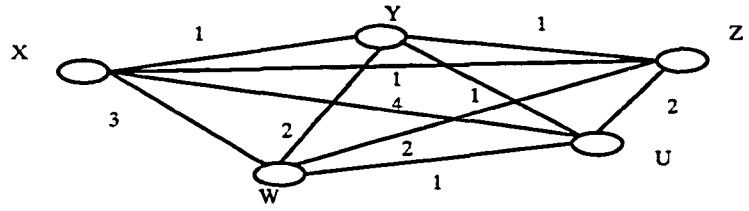
Message m will be sent by p_X to all members of the group, which requires four units of time to arrive at the farthest node (U). The nodes send a reply with a

Multicast Protocol	X send	Z send	W send	Average
Structure A				
BUS	0 to 4	1 to 6	3 to 7	1 to 6
BUS-TO	1 to 6	1 to 6	5 to 9	2 to 7
Structure B				
BUS	0 to 4	0 to 6	3 to 7	1 to 6
BUS-TO	2 to 6	2 to 6	5 to 9	3 to 7
CBCAST	1 to 7	1 to 8	2 to 7	1 to 7
ABCAST	8 to 12	6 to 9	8 to 12	7 to 10
Central Protocol (X central)	0 to 4	2 to 6	4 to 8	2 to 6
Central Protocol (Y central)	1 to 4	1 to 4	3 to 6	1 to 4

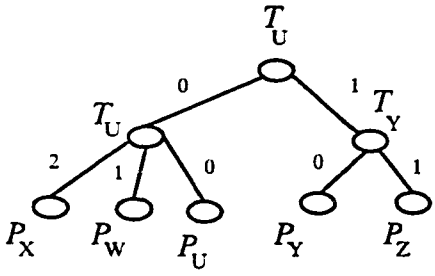
Table 8.1: Number of Hops for Example 1.

timestamp assigned to the message, which takes a maximum of four more units before these two rounds of messages are completed. Therefore, after eight units, p_X can assign a final timestamp and send it to all group members. At this point, p_X can also deliver m locally (first delivery occurs after eight units). However, p_W will not deliver m until it receives the final timestamp after twelve units. On the other hand, CBCAST, which ensures a causal-order delivery, is modeled through the tree with the shortest span over the group members. The central node achieves total order through two rounds of messages from the sender to the central node and then from the central node to the group.

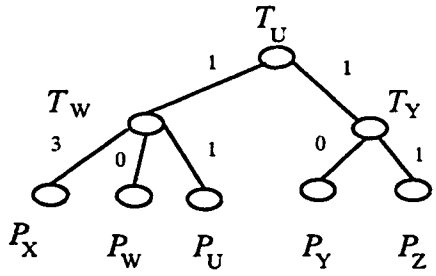
Figures 8.2 and 8.3 show connection graphs and associated costs for each connection. Examples 8.1 and 8.3 show partially connected sites; example 2 shows fully connected sites. All three examples have the same sites with different connectivity; however, they keep the same communication cost for similar edges. These examples are constructed this way so that the behavior of the different protocols



(a) Connected sites with communication cost.



(b) Communication Structure A.



(c) Communication Structure B.

Figure 8.2: Example 2 - a set of connected sites and possible communication structures.

can be compared with variations in connectivity.

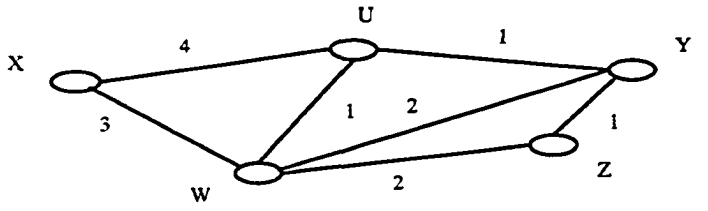
As we stated before, structure A is built without regard for resiliency and the effects of site failure (i.e., several TFMs are allowed to run on the same site). For example, in Figure 8.1 structure A runs two TFMs on site U. Structure B is constructed to increase the level of system resiliency because no more than one TFM process runs per site. Tables 8.1, 8.2, and 8.3 show the expected first and last delivery of the messages sent from different processes (p_W, p_X, p_Z). Each entry in the table shows the first and last delivery of the message as follows (*first delivery*, *last delivery*). We assume that the messages target the whole group. This assumption favors the evaluation of the ABCAST, CBCAST, and centralized protocols because the hierarchical protocols target subgroups with a lower cost than the other protocols.

Multicast Protocol	X send	Z send	W send	Average
Structure A				
BUS	2 to 4	1 to 4	1 to 3	1 to 4
BUS-TO	2 to 4	2 to 4	1 to 3	2 to 4
Structure B				
BUS	3 to 6	1 to 6	0 to 3	1 to 5
BUS-TO	5 to 8	3 to 6	2 to 5	3 to 6
CBCAST	3 to 7	2 to 7	3 to 7	3 to 7
ABCAST	6 to 9	6 to 9	6 to 9	6 to 9
Central Protocol (X central)	0 to 4	2 to 4	3 to 6	2 to 3
Central Protocol (Y central)	1 to 3	1 to 3	2 to 4	1 to 3

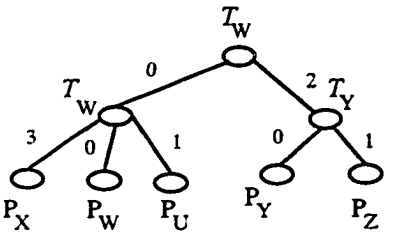
Table 8.2: Number of Hops for Example 2.

Both the BUS, the Central, and the CBCAST protocols enforce a causal order; the BUS-TO and the ABCAST protocol ensure a total order between multicasted messages. The BUS-TO protocol outperforms the ABCAST for all cases. The main reason for this difference in performance (which can be clearly seen in Tables 8.1, 8.2, and 8.3) is due to the three rounds of messages necessary to achieve this order in the ABCAST protocol. The BUS-TO protocol achieves total order through a more distributed protocol than ABCAST; this approach achieves a higher performance and resiliency.

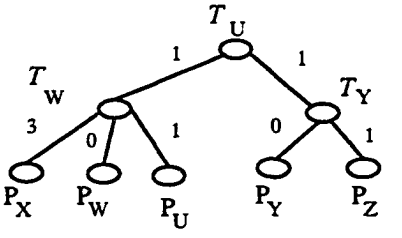
Tables 8.1 and 8.2 clearly show that the BUS-TO protocol is outperformed by the central protocol; however, the BUS-TO protocol achieves a higher performance in the example shown in Table 8.3. The results in Tables 8.1 and 8.2 are not unexpected because of the extra levels the message must pass to get to its LCA before it is multicasted to its group. These extra levels are not necessary with the central protocol because the message must move only one level in the hierarchy. Note



(a) Connected sites with communication cost.



(b) Communication Structure A.



(c) Communication Structure B.

Figure 8.3: Example 3 - a set of connected sites and possible communication structures.

the change in behavior in Table 8.3 in favor of the BUS-TO protocol; this change is due to the selection of the central node. The BUS-TO protocol outperforms the central protocol when either X or Y is used as the central node; this observation is reversed when W is used instead. The selection of the central node and the TFM sites are crucial to the performance of the protocol. This dependence may be a disadvantage of the hierarchical protocols in a highly dynamic group-membership environment because the cost of restructuring the hierarchy may be a burden to the protocol. The importance of the optimization algorithm (which builds the communication structure) on the performance of the protocol is also evident. The centralized protocol should achieve the best performance over any total-order protocol, given a wise selection of the central node. The performance of

Multicast Protocol	X send	Z send	W send	Average
Structure A				
BUS	3 to 6	1 to 6	0 to 3	1 to 5
BUS-TO	3 to 6	3 to 6	0 to 3	2 to 5
Structure B				
BUS	3 to 6	1 to 6	0 to 3	1 to 5
BUS-TO	5 to 8	3 to 6	2 to 5	3 to 6
CBCAST	3 to 7	1 to 9	2 to 8	2 to 8
ABCAST	10 to 15	10 to 15	6 to 9	9 to 13
Central Protocol (X central)	0 to 5	5 to 10	3 to 8	3 to 8
Central Protocol (Y central)	5 to 10	1 to 6	2 to 7	3 to 8
Central Protocol (W central)	3 to 5	2 to 5	0 to 3	2 to 4

Table 8.3: Number of Hops for Example 3

the BUS-TO protocol falls between ABCAST and the central protocol. However, the central protocol suffers a lower site-failure resiliency than the BUS-TO. Also, with the central protocol the high traffic directed to the central node may cause a bottleneck, which could result in a longer delay in message delivery.

Also notice that the BUS protocol outperforms the CBCAST protocol in examples 2 and 3. In example 1, the CBCAST protocol outperforms the BUS protocol in some cases (sender p_W for structures A and B in Table 8.1) but is outperformed by the BUS protocol in some other cases. However, the average performance index generally indicates that the BUS protocol achieves a faster delivery than the CBCAST protocol. The delivery time in this case relies on where the message is sent from in relation to the TFM sites.

The results show that both the centralized and hierarchical protocols are sensitive to the central site and the TFMs. This sensitivity is evident if the results

from structure A and structure B in Tables 8.1, 8.2, and 8.3 are compared. The results from the use of different central nodes for the centralized protocol can also be compared.

The three examples presented clearly show that the hierarchical protocols achieve a much faster delivery with structure A than with B. With structure B, the message encountered greater delays in delivery because of the extra hops the message must go through to reach its LCA (because no two TFMs run on the same site). As mentioned earlier, the intention of these examples is not to estimate the performance of the BUS and the BUS-TO protocols but to provide a sense of the expected improvement in delivery time and lowered communication cost.

8.3 The Point-to-Point Model

The first models assume a point-to-point network like the ARPANET. In a similar network, a process p that sends a message m to n processes actually sends n messages. When process p sends a message m, we expect that some processing cycles P will be added to send out m. Then, m will take time L to reach its destination (network delivery time). Obviously, both P and L are subject to different parameters such as the relative location of the sender and receiver, site load, and network load. However, for simplicity we will assume that P and L are constant at an average or worst-case value.

Let us consider first the number of messages N required to deliver an ordered multicast from process p to n destinations. The ISIS two-phase protocol [12] requires n messages to get the message to the recipients, then another n messages to return the local timestamps to the sender. The sender then sends the final timestamp to all recipients. This results in $3n$ messages for the two-phase protocol.

The centralized protocol requires that a single message be sent first from the

sender to the central site; $n-1$ messages are sent from the central site to the recipients. If extra nodes exist in the path from the central sites to the nodes, then the number of messages becomes $n+E$, where E is the expected number of extra nodes.

The network delivery time, which is the second item in our performance index, can be calculated similarly. For the two-phase approach, which requires three rounds of execution, the delay will be $L+nP$ in the first round for the message to be received by all destinations. The total delay for the second round, in which the sender receives the returning destination timestamps, will be $L+P$. The third round, in which the final timestamp is sent back to the destination, requires a delay of $L+nP$. The total cost is $3L+(2n+1)P$ for the two-phase protocol. The delay in the centralized protocol is $L+P$ to transport the message from the sender to the central node and then $L+(n-1)P$ for the message to go from the central node to all destinations including the sender. The total cost for the centralized protocol is $2L+nP$. For our hierarchical approach, the cost of the protocol depends on the length of the longest path from the LCA to the group member that is farthest away (at depth d_d) and the path from the sender to the LCA (at depth d_u). The delay from the source to the LCA is $(d_u+d_d)(L+P)$. The delay from the LCA to the group members at depth d in a worst-case scenario would be $dL+dP$. The total cost for the hierarchical approach is $2d(L+P)+nP$. A comparison of the performance indexes is shown in Table 8.4.

The performance of the hierarchical protocol depends on the value of d. In most cases of interest, d is relatively small because it tends to be on the order of $log_x(n)$, where x is the order of the tree.

In a comparison of the results in table 8.4, we can clearly see that for N (the number of messages) the hierarchical approach is significantly more efficient than the two-phase approach because d is a small number ($< log_x(n)$).

	Centralized	Two-phase	Hierarchical
# of messages	n	$3n$	$n + (d_u + d_d)$
Delivery time	$2L + nP$	$3L + (2n + 1)P$	$(d_u + d_d)(L + P) + nP$

Table 8.4: Performance Index for Point-to-Point Model

If the number of levels the message must pass through d is less than 2, then the hierarchical protocol will encounter less delay than the two-phase protocol. If d is greater than 2, then the hierarchical protocol may incur more delay than the two-phase protocol. However, we expect d to be smaller than 2 for smaller groups. Small groups are much more common in ordered multicasting applications such as replica control, in which only a few copies are kept at a small number of sites because the cost of maintaining replicated copies is high. Also note that if the processing cost P becomes more significant than L, then the hierarchical approach will perform better than the two-phase model. Also, if the hierarchical approach maps the physical connection and the TFMs are run at the gateway, a significant improvement will result in delivery time because all hops that the message takes to reach its LCA will be part of the route it would normally take to the rest of the group; therefore, no extra hops are taken.

Figures 8.4-8.8 show different performance curves for the centralized, two-phase, and hierarchical protocols under the point-to-point model. Figures 8.4, 8.5, and 8.6 are evaluated for a total number of nodes N= 1000 for different group sizes (1000, 100, 50, and 20, respectively). Figure 8.4 shows the delivery time of the three protocols for a group size of 1000. This group size should result in a similar performance between the hierarchical and the centralized protocols. This similarity results because when the group size is equal to N, then we have a tree of only one level, which makes the hierarchical communication structure resemble that of the centralized structure. This similarity can be seen in Figure 8.4, where

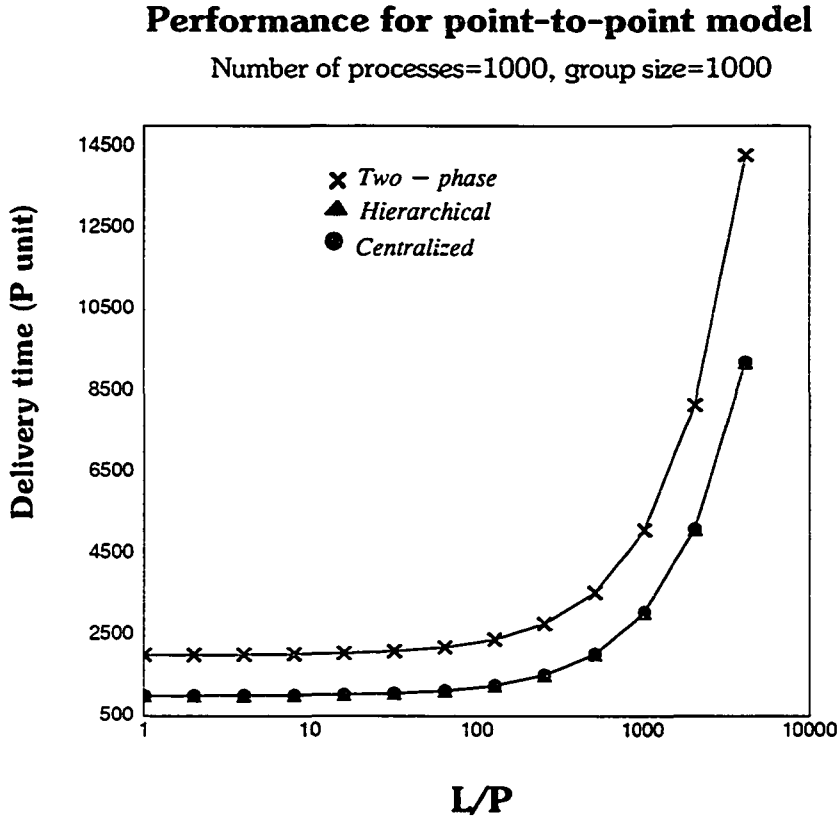


Figure 8.4: Performance curves for group size equal to 1000 with point-to-point model.

the curves for the centralized and the hierarchical protocols match. This arrangement is a best-case performance for our protocol. Both the hierarchical and the centralized protocols out-perform the two-phase protocol with this configuration. Figure 8.5 shows the delivery times for the three protocols for a group size of 100 processes. The performance curve for the hierarchical protocol falls between the curves for the centralized and the two-phase protocol, the hierarchical protocol achieves a performance similar to that of the centralized protocol for small values of L/P and performs more like the two-phase protocol for larger values of L/P. Figure 8.6 shows the same curves for a group of size 50. The same observation is seen in Figure 8.5; however, the two-phase protocol outperforms the hierarchical protocol for $L/P > 2048$.

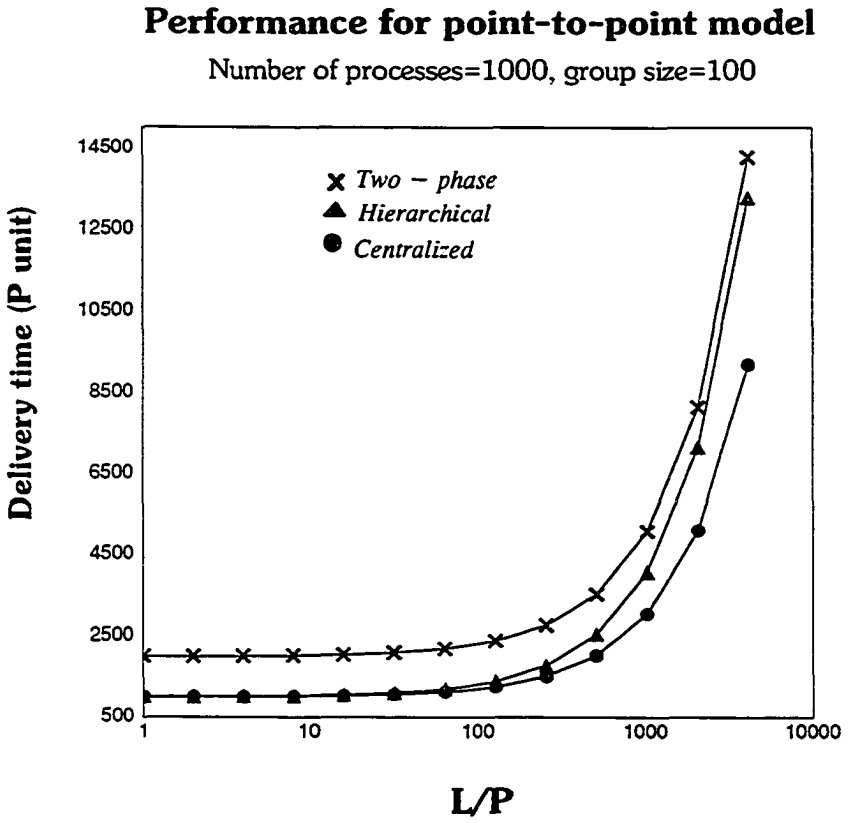


Figure 8.5: Performance curves for group size equal to 100 with point-to-point model.

The curves shown in Figures 8.4, 8.5, and 8.6 clearly show that the group size affects the performance of the hierarchical protocol. Here, we have assumed that the message is directed to all the processes; this assumption favors both the centralized and two-phase protocols over the hierarchical protocol. Figure 8.8 shows the hierarchical protocol performance with different group sizes. The figure plots curves for N = 100, 1000, and 10000 for different group sizes. The figures shows a shorter delivery time for larger group sizes. It also shows a shorter delivery time for smaller values of N.

In general, we conclude that the hierarchical protocol in its best-case performance for the provided model can achieve a performance similar to that of the centralized protocol. It outperforms the two-phase protocol for all group sizes

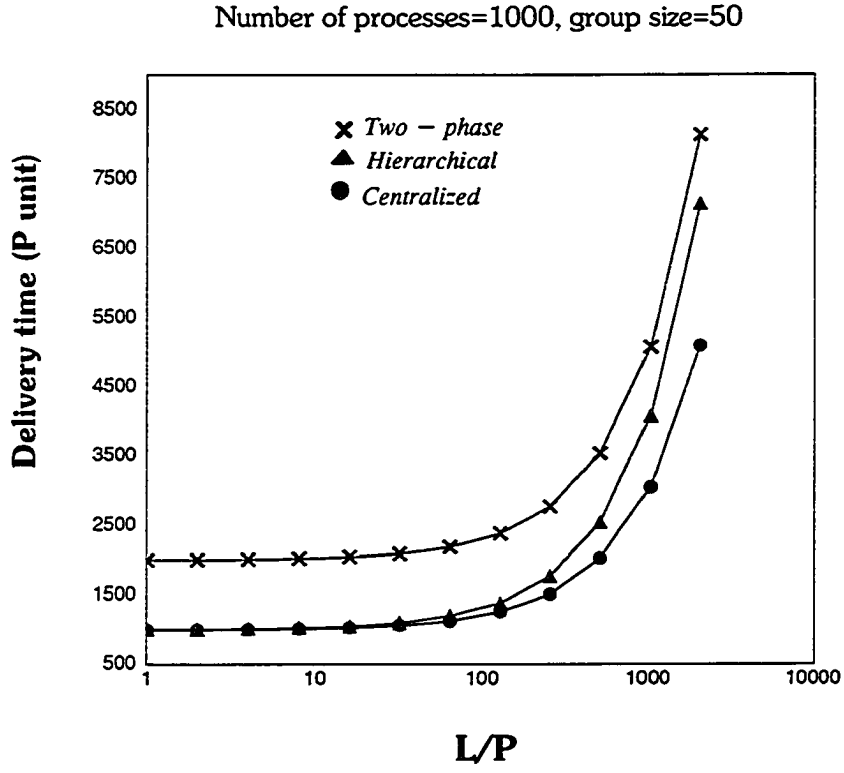


Figure 8.6: Performance curves for group size equal to 50 with point-to-point model.

greater than 25 for an L/P ratio of magnitude of order 3. For group sizes smaller than 25, the two-phase protocol outperforms the hierarchical protocol for higher ratios of L/P. For example, Figure 8.7 shows the outperformance of two-phase protocol over the hierarchical protocol for L/P values greater then 512. As the group size decreases, this intersection shifts toward a lower value of L/P.

Also note that both the model chosen here and the assumption of message direction to all groups favors the centralized and two-phase protocols. Also, the hierarchical protocol requires fewer protocol messages than the two-phase protocol, which provides a lower probability of message loss and, thus, a lower recovery cost. In addition to the result shown above, if the physical and the logical structures are aligned, the performance of the hierarchical protocol will improve to a greater

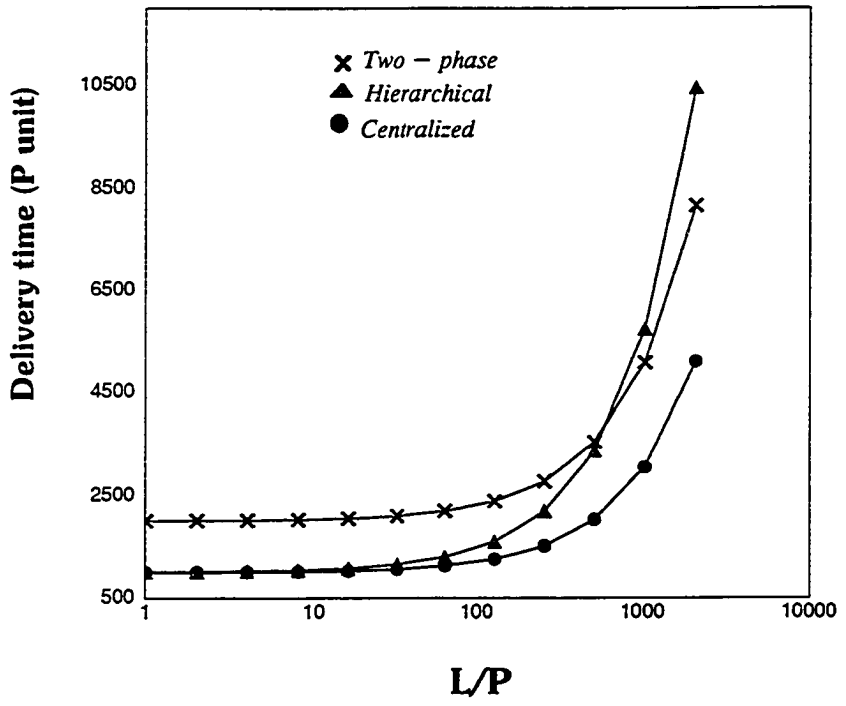


Figure 8.7: Performance curves for group size equal to 20 with point-to-point model.

extent.

8.4 The Multicast Model

The second model assumes a multicast environment in which messages can be multicasted to all group members at once with a multicasting address. The destination sites must verify members of the multicasting address. This verification can be accomplished at the software level; however, some network interfaces provide this flexibility at the hardware level.

We must calculate the same two performance indexes for this model. The number of extra messages N for the two-phase protocol is different than that calculated

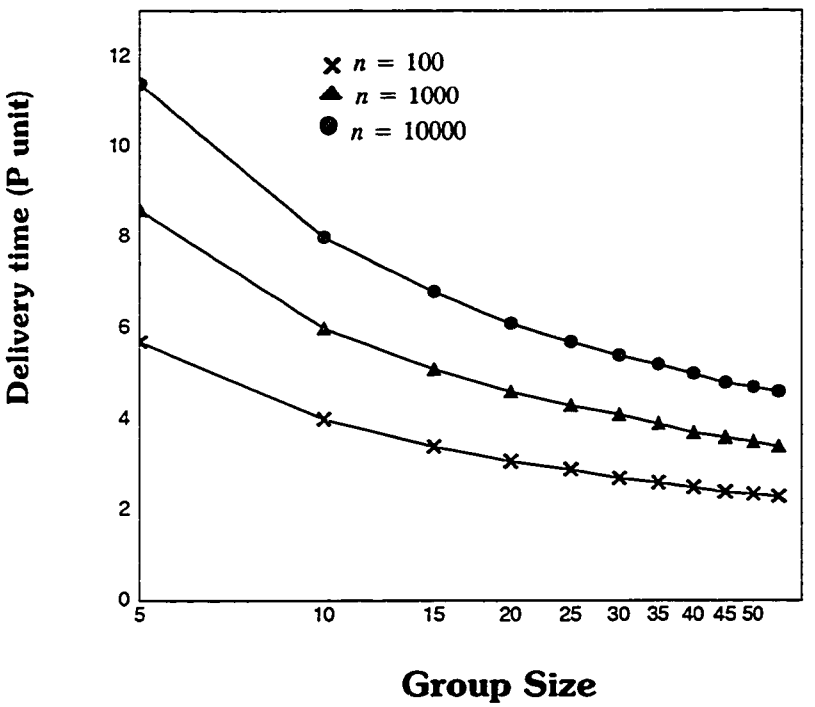


Figure 8.8: Performance curves for different group sizes with point-to-point model.

in the previous section. The sender multicasts one message to n members of the group (recipients), n messages from the recipients to the sender, and one message from the sender to the recipients. The total number of messages in this case is $n + 2$ for the two-phase protocol. The centralized protocol requires one message from the sender to the central site and one multicast message from the central site to the recipients: a total of two messages. The hierarchical protocol forwards the message from the sender to the LCA of the message; a multicast is initiated at each level of the communication structure that the message passes. As it arrives at its LCA, the message is directed down the hierarchy to the remaining recipients. The total number of messages is $3d$, where d is the expected path length from the sender to the LCA.

To calculate the second performance index, we must estimate the delay encoun-

	Centralized	Two-phase	Hierarchical
# of messages	2	$n+2$	$3d$
Delivery time	$2L$	$(n+2)L$	$(d_d + d_u)L$

Table 8.5: Performance Index for Multicast Model

tered in moving the message from its sender to its recipients. Assume that the message requires L units of time to travel from the sender to some of the recipients. This time includes processing time, network delay in delivering the message, and site checks to identify if it is a member of the multicasting address. If we multiply the number of messages calculated previously by L, we can estimate the delay of message delivery for both the centralized and the two-phase protocol.

For the hierarchical protocol, the total delay for message delivery will vary between the time needed for the message to get to its LCA and the time needed for the message to reach those recipients that are farthest from the LCA. After the message arrives at the LCA (it takes a total number of hops d), it can be delivered to any recipients that are directly connected to the message LCA $[(d+1)L]$. As it travels down the hierarchy, the message is delivered to all recipients with a worst-case delay of $(d_u + d_d)L$, where d_d is the path length from the message LCA to the farthest recipient in the targeted group. Both d_d and d_u have an upper limit of $d = log_x(n)$, where x is the order of the hierarchical structure and n is the number of nodes in the structure. Note that d is an upper limit for d_d and d_u because the message is never directed to the root of the tree unless the message is directed to the whole multicasting group; d will be the number of levels in the tree. We adopt a worst-case scenario have and assume that the delay time is $2dL$.

The performance indexes clearly show that the centralized approach performs better than both the two-phase and the hierarchical approach. However, in the centralized approach the central node may become a bottleneck because of high

message traffic. This may result in an increase in the value of L, which, consequently, will increase message delay. In a comparison of both the two-phase and hierarchical protocols, a breakpoint occurs at $d = n/2 + 1$. In other words, if d is greater than $n/2 + 1$, the two-phase protocol will outperform the hierarchical protocol. Because we are considering a communication structure that is based on a tree structure of order x, d is on the order of $log_x(n)$, so that d is smaller than $n/2 + 1$. Therefore, the two-phase protocol is comparable to the hierarchical protocol only when n is small; for larger value of n, the hierarchical protocol will outperform the two-phase protocol.

8.5 Remarks

When the TFM is running at the gateway (although this is not necessary), a better performance results because messages that are leaving the LAN are redirected to the gateway by default. Therefore, if the TFM process runs on the gateway, then the number of hops that the message must make to leave the LAN decreases.

8.6 Conclusion

The hierarchical protocol is a viable option for ordered multicasting in both point-to-point and multicast networks. The simplified models shown here validate this assertion in terms of both extra protocol messages used and message delivery delay. However, the hierarchical protocol has one drawback in the extra cost needed to build the communication structure. This cost can be acceptable if changes in the group membership are not frequent and if the number of messages propagated during the lifetime of the groups is large. Another important issue is the importance of optimizing the communication structure; the performance of the

protocols examined here are highly affected by this structure.

Chapter 9

Reliability and Fault Tolerance

9.1 Introduction

Fault tolerance is concerned with how well the system continues to function in the event of failure. It is a major concern because of the tremendous effect failure has on the functionality of all protocols. A higher probability of failure in an interconnected LAN environment increases the necessity of improving protocol resiliency. We must identify the types of failures that may be encountered before we can address protocol resiliency. Two types of failures are of interest here:

- *Transient failure:* This type of failure causes some messages to be lost because of a buffer overflow or link failure.
- *Persistent failure:* This type of failure causes network partitions, in which a group of hosts is disconnected from the other broadcast groups. This failure results in a total loss of messages multicasted by sites from the other partitions.

Recovery from a persistent failure is more costly than a recovery from a transient failure because it essentially requires message retransmissions to multiple destina-

tions [39].

Another method for classifying failures is based on the way they occur and the level of synchrony in the system. Based on this method, some researchers have classified failures into two types: *omission failures* and *timing failures* [43, 27, 62, 44]. Omission failures consist of crash, send omission, and receive omission failures of processes, as well as link omission failures. Timing failures consist of omission, clock, and performance failures. Certain types of failures are characteristic of certain networks; individual processes and links generally commit failures from the same class. Thus, a network with omission failures is not subject to clock, performance, or arbitrary failures. Similarly, one with timing failures is not subject to arbitrary failures [44].

As presented earlier, distributed systems are either synchronous or asynchronous. Asynchronous systems have recently gained attention for several reasons: they have simple semantics; the applications that are programmed under them are easier to port; and, in practice, variable workloads are sources of asynchrony (thus, assumptions of synchrony are at best probabilistic). Informally, a distributed system is *asynchronous* if no bounds exist on message delay, clock drift, or time necessary to execute a step. Thus, in order to say that a system is asynchronous, no timing assumptions can be made whatsoever.

Although the asynchronous model is attractive, the Atomic Broadcast cannot be solved deterministically on an asynchronous system that is subject to even a single crash failure [37, 34]. Essentially, the impossibility results for Atomic Broadcast stem from the inherent difficulty is determining whether a process has actually crashed or is very slow. To overcome the impossibility results, previous research focused on the use of randomization techniques [16, 8], or the study of several models of *partial synchrony* [34, 35]. Nevertheless, the impossibility of deterministic solutions to many agreement problems such as Atomic Broadcast

remains a major obstacle.

No protocol that currently exists is resilient to network partitioning when messages are lost (i.e., the possibility always exists that some sites will block messages when networks become partitioned [71]). Failure properties of the described network are characterized by the following set of assumptions:

1. *Benign Failures.* Process and link failure are benign. Benign failures are synonymous to omission failures in asynchronous networks and to timing failures in synchronous networks. In a system with only benign failures, processes do not commit arbitrary failures. Thus, a faulty process does not change its state arbitrarily or send a message that it was not supposed to send. Our failures fall under benign failures because of their practicality and the availability of automatic methods that increase the fault tolerance of an algorithm. Fault tolerance is achieved by a set of algorithms that can translate any algorithm that is tolerant to a certain type of failure to an algorithm that can tolerate a more severe type of failure [15, 6].
2. *No partitioning.* Every two correct processes are connected via a path that consists entirely of correct processes and links (partitioning will be discussed later in this chapter along with different methods of handling it).

9.2 The Reliability Model

Our model is similar to the work presented in references [43] and [27] and is an extension to the model presented in reference [44].

9.2.1 Multicast Network

In a multicast network, messages can be multicasted to all group members at once with a multicasting address. The destination sites must verify that they

are members of this multicasting address. Normally, this verification is done at the software level; however, some network interfaces provide this flexibility at the hardware level.

1. **Networks with No Failures**

   A multicast network allows each of its processes to multicast a message m to a group address X, where X represents a set of processes that is the target of all messages multicasted to its address. In such a network, any set of processes that is a member of a group can communicate by multicasting and receiving messages, as described below. In this section, we assume that processes and links do not fail.

   - **Properties of Processes:**

     Each process is capable of executing certain operations, such as writing a local variable or sending a message. The execution of an operation by process p is a step of p. We do not assume that the steps are atomic; a step consists of a sequence of atomic events, indicated by a start and an end event. The fact that steps are not atomic will permit us in the next section to model failures that interrupt the execution of an operation in the middle. Hence, the execution of a process p is modeled as a sequence of events grouped into steps, such that the start event of each step (except the first one) immediately follows the end event of the previous step. If this sequence includes the start event of a step, we say that p has *started* that step; if it includes the end event of a step, we say that p *completed* that step. Associated with each process p is an automaton whose transition relation describes the legal sequences of events for p. We assume that:

     a. **Every process completes an infinite number of steps.**

This implies that every process eventually completes every step that it starts.

- **Properties of Multicast and Receive:**

  Let $X = \{x_1, x_2, ..., x_k\}$ be a group of processes connected by a multicast network. Associated with this network are the communication primitives "multicast" and "receive", which are among the operations that can be executed by members of X. The operation multicast takes a message and a group ID as parameters; receive returns a message. The execution of the multicast primitive with parameter m and X is a step denoted by multicast(m,X); the execution of the receive primitive with return value m is a step denoted as receive(m). We say that x_i *multicasts* m *to group* X starts the step multicast(m,X); we say that x_j *receives* m if x_j completes the step receive(m).

  Associated with each process in the multicast group is an *outgoing message buffer*, denoted $omb(x)$, and an *incoming message buffer*, denoted $imb(x)$. Informally, when x_i multicasts a message m to X, x_i inserts m in $omb(x_i)$; the multicast network transports m to all members of X. Therefore, m will be entered into $imb(x_j)$, where $x_j \in X$, and x_j receives m from $imb(x_j)$. More precisely, the multicast and receive primitives associated with a group X satisfy the following three properties:

  b. **If x_i completes the multicast of m to X, then m is eventually inserted into $omb(x_i)$.**

  c. **If m is inserted into $omb(x_i)$, then m is eventually inserted into $imb(x)$, $\forall\ x \in X$.**

  d. **If m is inserted into $imb(x_j)$, then x_j eventually receives m.**

  These three properties imply that:

– **If x_i multicasts m to X, then $\forall\, x \in X, x$ eventually receives m.**

We also assume that:

b′. **m is inserted into $omb(x_i)$ at most once and only if x_i sends m to X.**

c′. **m is inserted into $imb(x)$, $\forall\, x \in X$, at most once and only if m is in $omb(x_i)$.**

d′. **x_j receives m at most once, and only if m is in $imb(x_j)$.**

Properties (b′)-(d′) imply *uniform integrity*, which means that for any message m, $\forall\, x \in X, x$ receives m at most once from x_i and only if x_i previously sent m to X.

The preceding definition of a multicast network assumes that no failures occur. In the next section, we consider some of the failures that can affect processes and links. These failures will be defined as violations of properties (a) and (b′)-(d′). We will *not* allow the violation of properties (b′)-(d′); thus, uniform integrity holds even in networks with failures. We will also not allow violation of the postulated property in regard to messages sent by a process to itself.

2. Networks with Omission Failures

Failures can be defined as deviations from correct behavior. In networks with omission failures, processes and links may violate properties (a) and (b′)-(d′). The violation of property (a) is described below.

To model a violation of property (a), we introduce a special event called a crash. Every process p can execute a crash at any time, and after doing so it stops executing further events. This is modeled by the addition of a new terminal state to the automaton associated with p, and a transition from

every other state of p to that terminal state. The event associated with such a transition is defined as a crash. We say that p *commits a crash failure* if it executes a crash event.

Because no event can follow a crash, a process that crashes can execute only a finite number of events, and, therefore, completes only a finite number of steps. Thus, a process that crashes violates property (a). We assume, however, that *only* processes that crash violate that property. That is, a process that does not crash completes an infinite number of steps.

The violation of properties (b), (c), and (d) of multicast and receive is described below.

- Process x_i *commits a multicast omission failure on* m if x_i completes $multicast(m, X)$ but m is never inserted into $omb(x_i)$, where $x_i \in X$ (violation of property (b)).
- The multicast network *commits an omission failure on* m if m is inserted into $omb(x_i)$ following a $multicast(m, X)$, but m is never inserted into $imb(x)$, *for any* $x \in X$ (violation of property (c)).
- Process x_j *commits a receive omission failure on* m if m is inserted into $imb(x_j)$ but x_j never receives m and does not crash (violation of property (d)).

If a process or a multicast network commits a failure, we say that it is *faulty*. Recall that in networks with no failures, if x_i multicasts m to X, then $x, \forall x \in X$, eventually receives m. The properties of multicast networks with omission failures imply *validity*, which means that if x_i multicasts m to X and *for any* $x \in X, x$ does not receive m, then one of the following holds:

1. x_i does not complete the multicast of m, or
2. x_i commits a multicast-omission failure on m, or
3. the multicast network commits an omission failure on m, or

4. x, *where* $x \in X$, commits a receive-omission failure on m, or

5. x, *where* $x \in X$, crashes.

9.2.2 Point-to-Point Network

In a point-to-point network, a pair of processes connected by a link can communicate by means of send and receive primitives. From a reliability perspective, the point-to-point network can be seen as a specialization to the multicast network. Here, the multicast group will have a single process as a member.

A point-to-point network can be modeled as a directed graph with nodes that represent processes and edges that represent communication links between processes. In such a network, any two processes that are connected by a link can communicate with each other by sending and receiving messages. The same rules presented in Section 9.2.1 will still apply, with some minor changes because of the simple model represented. Receive omission failures are easier to represent and to recover from because they involve a single process and not a group of processes. We will not repeat the model description here; however, more details about similar models can be found in reference [44].

9.3 Reliability Approaches for our Protocols

We will show in the next several subsections how our protocol tracks these failures and how recovery should be conducted.

9.3.1 Network Omission Failure

Our protocols rely on the assignment of a timestamp to the message m_k at the sender p_{jx}. This timestamp is added to MTS_{m_k}, which accompanies the message on its journey for delivery. The protocols use a set of reliability procedures built

in to the protocol to track send omission, receive omission, and multicast omission failures, as well as out-of-order messages. These tracking mechanisms are made available through the timestamp vector that accompanies the message ($MTS_m[]$) and the timestamp vector that exists at each active process and at the TFMs ($PTS_p[]$). These tracking mechanisms are part of the procedures presented in Chapters 4, 5, 6, and 7.

The message m_k, as it leaves p_{jx} and is directed to its TFM ($\mathcal{T}_x$), will be checked on its arrival at $\mathcal{T}_x$ for lost or delayed messages; $\mathcal{T}_x$ can detect lost or delayed message from p_{jx} because of the entry in $PT_{\mathcal{T}_x}$ that represents the last message that arrived from p_{jx} and was timestamped. Because all p_{jx} messages are directed toward $\mathcal{T}_x$, any lost or delayed messages can be tracked by the arrival of another message that arrives from the same sender. This process makes $\mathcal{T}_x$ aware of any lost or delayed messages; in this case, m_k is held to wait for m_{k-1}. Any send/multicast omission failures and network/link omission failures that concern sender-receiver interaction can be tracked with this approach.

Another timestamp entry that is provided by the protocols to check for network omission between TFMs is the OLDTS. The $OLDTS_{\mathcal{T}_x}$ timestamp is used to provide a reliable delivery scheme between two consecutive level processes while a message is traveling up in the hierarchy. To clarify the functionality of $OLDTS_{\mathcal{T}_x}$, assume that a message m_k is sent from $\mathcal{T}_x$ to $\mathcal{T}_w$ (the TFM of $\mathcal{T}_x$). The message m_k is timestamped at $\mathcal{T}_x$ before it is forwarded to $\mathcal{T}_w$ with $LTS_{\mathcal{T}_x}$. This timestamp from $\mathcal{T}_x$, which is assigned to $MTS_{m_k}[1]$, is used at $\mathcal{T}_w$ to be compared with $PTS_{\mathcal{T}_w}[\mathcal{T}_x]$ to check the message order. In order for this process to be effective, all messages that are timestamped at $\mathcal{T}_x$ would have to be forwarded to $\mathcal{T}_w$ so that $PTS_{\mathcal{T}_w}[\mathcal{T}_x]$ would have the same sequence as $LTS_{\mathcal{T}_x}$. However, this would not normally be the case because the message that has $\mathcal{T}_x$ as its LCA would not forward to $\mathcal{T}_w$; therefore, $PTS_{\mathcal{T}_w}[\mathcal{T}_x]$ would be missing those messages with $\mathcal{T}_x$ as

their LCA. As a result, $LTS_{\mathcal{T}_x}$ cannot be used for this comparison. This problem forces us to introduce the $OLDTS$ as a timestamp variable at each TFM process. The OLDTS keeps the timestamp of the last message forwarded to the next highest level TFM from this process. When message m_k is forwarded up the hierarchy, the first entry in the vector MTS_{m_k} carries a copy of $OLDTS$. This entry is responsible for ensuring the ordered delivery of m_k from the sender $\mathcal{T}_x$ to its TFM $\mathcal{T}_w$. The value assigned to $MTS_{m_k}[0]$ should be in sequence with $PTS_{\mathcal{T}_w}[\mathcal{T}_x]$ unless a delayed message exists. The $OLDTS_{\mathcal{T}_x}$ timestamp is then adjusted by assigning the timestamp value given to m_k by $\mathcal{T}_x$.

The question here is whether or not this tracking and detection mechanism actually accomplishes its purpose. In spite of the fact that this mechanism fits the asynchronous definition for no crash failure, the performance of the system is affected. For example, in the case of message m_k sent from p_{jx} to $\mathcal{T}_x$, where m_k is lost and p_{jx} did not send any other messages after m_k to $\mathcal{T}_x$ for a period of time, a long delay results before this lost message is detected.

Several options are available to resolve this problem. One is the use of a timeout mechanism to detect lost messages and to decrease this delay effect. With this feature, the active processes try to maintain a message exchange with their TFMs. Furthermore, the TFMs in the communication hierarchy try to act similarly between TFMs that report to each other. As a result, if no traffic occurs for a certain period of time between two processes that normally interact with each other, then the *initiator* p_i will send a *Status Information Packet* (SIP) to the *receptor* p_r to check the messages status. The SIP will carry with it the part of the p_i timestamp structure that pertains to p_r; p_r can then decide if any lost messages exist and respond to p_i's SIP message with a *Status Information Packet Reply*(SIPR). The SIPR will be a null message (if no lost messages exist) to simply inform p_i that p_r is alive but has no message to send. If one or more lost messages

exist, then p_r will begin to retransmit the lost messages along with the SIPR.

The possibility exists that p_i will have to send multiple SIP messages to a specific process before an SIPR is returned because the SIP or the SIPR messages may be lost. Therefore, p_i will adopt a timeout scheme before it retransmits another SIP. Another possibility, if p_i receives no reply, is that p_r experienced a crash. A crash can be detected with the SIP algorithm or with a more sophisticated algorithm that uses one of the known failure detector algorithms. This algorithm signals a p_r crash and stops the SIP requests that are traveling to p_i.

The approach we adopt can be initiated in two different ways: with the *TFM* or with the *active process*. The TFM initiation approach relies on the assumption that the TFM is sending the SIP message when a communication link timeout. The active process initiation approach assigns the task of sending the SIP message to the active process. Both approaches will be discussed in more detail in Section 9.3.5 because we use these approaches to track site failures.

9.3.2 Receive Omission Failure

A receive omission failure results when rule (d) is violated. Our timestamp mechanism allows the TFM, as well as the active process, to detect the receive omission failure. A lost message m_k that is traveling from p to q is detected when the next message m_l that comes from p is delivered to q because q will check the timestamp vector added to m_l by p. This vector will reflect a timestamp difference between $MTS_{m_l}[l]$ and $PTS_q[p]$, which indicates a missing message.

This detection is adequate in a high-traffic environment because the period between the time that m_k and m_l are delivered will be short. In an environment with a lower traffic volume, a failure detector or a timeout mechanism (as described in Sections 9.3.1 and 9.3.4) will have to be relied upon. The retransmission request will still be handled as described in Sections 9.3.1 and 9.3.4.

9.3.3 Multicast/Send Omission Failure

A multicast omission failure happens as a violation of rule (b). When the send operation is completed, the timestamp structure at the sender (p) is modified as a sequence to the multicast/send operation. However, the message is not inserted in the $omb(p)$; thus, it will not be inserted in $imb(q)$. This omission failure can be handled as a network/link omission failure, as described in Section 9.3.1.

Multicast omission failures require more than just a single message recovery. Because the multicast operation directs the message to a multicast group, in order to meet the requirements of rule (b) and (c), the message must be delivered to all members of the group. The message timestamp vector is still valid because each member of the group can track a message loss with the same mechanism described in Section 9.3.1.

9.3.4 Retransmission Buffers

The TFM process may be the best candidate for the retransmission site for a number of reasons, such as:

- The TFM process is the center of all group messages, which provides a good pool for piggybacking retransmission requests.
- It can provide a potential low cost garbage-collection algorithm for cleaning retransmission buffers.
- It provides a faster method for honoring retransmission requests for other CUs.

The failure of the TFM may mean the loss of the retransmission buffers. However, under the fail-stop assumptions presented in reference [71], in the event of a failed process crash, all correct processes are informed of the crash and have

access to any information written by the faulty process in its stable storage. This fail-stop assumption provides an inexpensive solution to this problem.

However, if we assume the unavailability of such stable storage or the possibility of network partitions that disconnect the retransmission site, then another approach is necessary.

The possibility of TFM failure, where the TFM is the main retransmission-request server, requires a secondary retransmission repository. The secondary repository is responsible for honoring retransmission requests in case of TFM failures or overload. As a result, our retransmission buffer is resilient up to the main and no secondary or any number of secondary within the same unit. Actually, we developed a distributed secondary retransmission buffer that provides more resiliency and better performance. It also provides a means of granting send omission failure retransmission requests.

The possibility of retransmission requests makes it necessary for each process to keep old messages to honor the retransmission requests. This may need a large storage unless we use a reasonable garbage-collection algorithm. Messages can be eliminated from the retransmission buffers once they are no longer subject to any retransmission requests [29]. A simple garbage-collection protocol is shown below to provide a main retransmission buffer. When a message m_i is sent from process p_{jx} to $\mathcal{T}_j$ of cu_j, the following steps occur:

- message m_i is kept in the sender (p_{jx}) retransmission buffer; the sender does not release it until it receives the multicast version of the message from $\mathcal{T}_j$.
- $\mathcal{T}_j$ is responsible for keeping the copy of the message for retransmission requests. After multicasting it to the group, it will hold m_i in its retransmission buffer (which is better than keeping the retransmission buffer at the sender because the message will not have to be redirected from the TFM process to the sender). Furthermore, it also has a better chance of benefiting from

piggybacking.

- Each process p_{jx} ($where\ p_{jx} \varepsilon cu_j$) will send the last message received m_l with each message to its TFM $\mathcal{T}_j$. This step will allow $\mathcal{T}_j$ to release copies of message $m_y : m_y \leq m_l$ in case they are received by all members of cu_j.

A problem is encountered when messages are not sent; extra messages are held for longer periods of time. To eliminate this problem, *null* messages can be sent to inform the TFM process of the last message received.

The secondary retransmission buffer will be distributed among the members of the group. This gives the retransmission algorithm more resiliency and improves performance because it eliminates overloading one site. In addition, we believe that the use of the sender's retransmission buffer as the main buffer and the TFM as the secondary buffer may provide a better solution in a majority of cases with high traffic. The algorithm relies on holding m_k at p_{jx} (the sender of m_k) until all destinations receive m_k. Only then can p_{jx} remove m_k from its retransmission buffer. The reception of m_k at all destinations can be publicized to the active processes by their TFMs. The TFMs can detect the delivery of m_k at its destination by analyzing $MTS_m[]$, where m is any later message that passes by the TFM. This identification can be done because m carries (in $MTS_m[]$) the timestamp of the last messages received from each process.

Another possible method is the use of a recording site to log all traffic. This site can be used either to rebuild the lost retransmission buffer or to answer the retransmission requests if no buffer is kept.

9.3.5 Site Failure

The detection of a site (process) failure initiates a reformation protocol that depends on whether the failed process is a TFM process or a participant.

- Participant failure

  The failure of a participant is detected by the TFM by a *Still-Alive* protocol that is initiated after a period of silence from the participant. The Still-Alive protocol can be initiated from the participant or the TFM.

  1. Participant Still-Alive initiator

     If the participant has not sent any messages to its TFM for a period of time, then the participant sends a Still-Alive Packet (SAP) to inform the TFM that it is still alive. If the TFM is timed out for a participant message, before it assumes that the participant has failed it will initiate an Alive Information Packet (AIP). The reason for using this message is the possibility of the loss of the SAP. If the TFM is timed out, it will assume the participant failure.

  2. TFM Still-Alive initiator

     In this protocol, if the TFM does not receive a message from any of the CU members for a certain period of time, then the TFM will send an AIP. If the TFM timeout for a participant reply, it will assume that either its message or the participant's message is lost and will send a second request. After a predetermined number of retries, a failure is assumed.

The reformation phase of the process initiates a new group by eliminating the failed process from the CU membership. The TFM of the CU that contains the failed process will multicast a reformation message (R_x) that contains the new membership to the subgroup, and the reformation protocol ends. The reformation message is handled as a normal message from an ordering perspective. In other word, R_x is timestamped at the TFM and is delivered to the members in its order with other reformation and normal messages.

The members use R_x to update the CU membership list (CUML).

- TFM failure

  If the failed process is a TFM, then we face two main problems:

  1. We must create a new TFM process.
  2. We must restore the lost information that is kept in the TFM structure, including primarily the timestamp values and the retransmission buffers.

  - The new TFM process sends a request to each member of the group for status information. Membership information is already available at the process from the CUML. It also sends a message to the higher level TFM for its election to request the last sequence number sent from the previous TFM and the timestamp vector for the higher level.
  - Upon delivery of the information that declares the TFM failure, the CU members discard all messages sent from the old TFM.
  - The CU members reply assists the new TFM in restoring the sequence numbers for its communication with the members. The replies contain the messages that the members have forwarded to the TFM for broadcasting that have not yet been multicasted by the TFM. The members also send their retransmission buffers to help build the new TFM retransmission buffer.
  - The TFM, upon receiving these messages, begins to build its structure. First, the timestamp vector of the higher level TFMs is restored from the upper subgroup TFM. Second, the subgroup timestamp is restored by using the largest timestamp value received from the members of the group. That is, the new TFM assigns a value for the timestamp by obtaining the maximum timestamp known to the group members.

- The TFM, once its structure is complete, will retransmit all the messages received from its CU members. The timestamp vector sent from each member will identify the messages that need retransmission. These messages are retransmitted because some messages may have been ready to be sent to their destinations when the TFM went down. The processes are assumed to reject any duplicates.
- Any messages that were directed to the failed TFM for a higher level process (if the failure occurred before the messages were multicasted) are retransmitted. These messages will be recovered because retransmission occurs for all messages with a timestamp greater than the timestamp of the upper level node.
- In order to rebuild the retransmission buffer, the new TFM needs the messages from backup sites. As discussed earlier, each process has a secondary retransmission buffer for the messages that originate from it as a backup for its TFM retransmission buffer. These messages are released after the TFM release, which allows the new TFM to build its retransmission buffer with these backups.
- The TFM takes on its normal tasks by multicasting the new group formation.

9.3.6 Network Partitions

Our work on partitions relies on the use of a failure-detector algorithm that reports to a set of membership servers, which provides members with an updated membership list. The membership server's task is extended to monitor the operational status of group members. For group members, those that are operational are registered with the membership servers; in a failed process, they are then unregistered.

Failure detectors have been the target of various research efforts to overcome the impossibility result reported in reference [37]. The conclusion drawn in this study is that the consensus problem cannot be solved in an asynchronous system that is subject to process failures. This conclusion is often taken to mean that software must operate with some risk of inconsistent failure detection. A related result exists for the database commit problem in the presence of partitions [71].

In our work here, we will not introduce a new failure detector algorithm, but we will use a combination of the currently available failure-detectors algorithms, namely, the combination reported in references [66, 18]. The algorithm relies on a membership service that monitors the status of group processes and excludes any process that is *suspected* to have failed. All communications that come from processes that are not registered with the servers will be discarded. If the excluded process did not crash and was wrongly suspected, we have two approaches that can be adopted:

- When a suspected process is mistakenly assumed to have failed and is excluded from the membership list, it can be added again when discovered to be alive. A similar failure-detector is presented in reference [17].

- When a suspected process is mistakenly assumed to have failed and is excluded from the membership list, it cannot be added back to the group view; it will be assumed to have failed. The process can similarly join the group back by using a normal join operation. Ricciardi [66] presents a similar failure detector.

The failed process will be informed of its exclusion from the group membership as soon as it recovers.

When physical partitions occur, the membership service will keep the system logically connected. However, the members will function under some stricter rules.

The membership service will identify two types of partitions: the main and the secondary partitions. The main partition will be the one that assumes full working capability (as the original system); the secondary partition will function under a restricted execution. This restricted execution allows the algorithm to provide consistent system behavior even when partitions exist. If no partition can be assumed, as a main partition, then all partitions will be assumed to be secondary partitions and will be allowed a restricted execution. Actually, some applications do not require similar strict correctness and can allow all partitions to proceed with full execution after the reformation phase (e.g., conferencing systems, air-traffic-control monitor updates, and stock exchange screens). We allow applications to identify what level of correctness they require in similar failures, and the protocol will allow either strict or full execution to the partitions. Also, some applications may require a single partition execution and a halt state to the secondary partition. An example of similar applications is replica control. This approach allows several partitions to operate simultaneously. Here, we adopt the approach presented by Ricciardi in reference [66] to provide the partitions with a consistent scenario.

Another approach [2] can be adopted here that provides a less restricted execution. This approach allows multiple membership views to exist simultaneously and requires neither atomicity nor uniformity in committing new views. This approach maintains more replicated data availability while it provides a weaker consistency.

We have adopted the first approach, with some modification, to cope with network partitions. For the remainder of this section, we describe the algorithm but do not provide a proof. Readers are referred to reference [65] for a similar proof.

The Group Management Protocol

With this algorithm, we attempt to provide a virtual, centralized site that can provide a membership service to the group. This site is the main authority in defining operational sites that still belong to the group. We define a set of processes called the *View Maintainers* (VM). This set cooperates in defining the *Global View* (GV) of the group. The VMs must create the illusion of a single fault-tolerant process that requires them to agree on the group membership, as well as the *VM list* (VML). The VML contains all the VMs of the group.

When a process p suspects a failure of a process q (because of a timeout or by running a failure-detector algorithm), it will multicast a $faulty_p(q)$ message to the VML members. After the $faulty_p(q)$ message is sent, p multicasts the $remove_p(q)$ message to the VML members; the members perform the actual removal of q from the views of all VML members that p believes to be operational. Similar to $faulty_p(q)$ and $remove_p(q)$, p can execute $operating_p(q)$ (p believes q is functional) and $add_p(q)$ (p adds q to the view at the VM in the VML).

The protocol tries to create a single process illusion in regard to the VM in order to guarantee a global consistency. The GV is defined if, and only if, local views of all its functional members agree.

The algorithm works as follows:

- An elected VML member, denoted *mgr*, coordinates updates among all the local views of the VML members.
- When *mgr* suspects an outer members' failure (or that a subset of the outer members has failed), it initiates a two-phase update algorithm that operates as follows:
  - **Phase I**

    The *mgr* proposes q's removal by multicasting $remove-req(-q)$ to the

members of its local view. The *mgr* then waits for each member to respond or to decide if a member is faulty. In this way, at the end of phase I, all core members that *mgr* does not believe to be faulty know that q may be faulty.

- **Phase II**

  If *mgr* receives responses from a majority subset of its current local view, then it multicasts a *commit* message $commit(-q)$. If the *mgr* does not receive a majority response, a minority partition may exist. If this is the case and the protocol allows the secondary partition to resume with a restricted or full execution, then the algorithm identifies this grouping as a secondary partition and informs the core members of their membership in this partition. The core members inform their active processes of the new secondary partition and execution is resumed. If the protocol does not allow execution in the secondary partitions, then the *mgr* must block if it does not receive a majority response. If the local views are identical at the beginning of this protocol, because *mgr* is a single process local views are identical at the end of it.

The remove message coordinates belief among the core that q may be faulty; the commit message tells outer members that the group has reached agreement on q's failure and that they should now remove q from their local views. However, because *mgr* does not receive a response from outer members it believes to be faulty, it cannot know whether these members received its removal message. From *mgr*'s perspective, these members may not be aware of the current update to the group view, which would render core-wide agreement on the new view, contingent upon the subsequent removal of these faulty members. The *gossip* approach used ensures that operational outer processes become aware of such contingencies.

When the group view is being added, *mgr* sends the new process(es) p a $state - transfer$ message that gives p permission to join and informs p of all relevant system states. The *mgr* awaits a reply and then multicasts the commit message to the entire new group.

Reconfiguration Algorithm

When *mgr* is believed to have failed, the outer members execute a *reconfiguration algorithm* to select a new coordinator and, if necessary, reestablish the group view. Local view agreement may be lost, for example, when *mgr* fails in the middle of a *commit* multicast. The local views will differ, which will result in an undefined group view.

Successful reconfiguration involves the solution to two problems:

1. Determination of which process(es) should initiate the reconfiguration and which should assume the *mgr* role.

2. Determination of which update should a reconfiguration initiator propose to resolve core member inconsistencies.

A reconfigurer must be able to determine the last defined group view and propagate the correct proposal for the succeeding group view. The most difficult aspect of reconfiguration involves *invisible commits.* An invisible commit occurs when the only processes that receive a commit message fail or are believed to be faulty by the rest of the group. This is significant for reconfiguration: although no subsequent reconfigurer will know whether these processes committed the change to their local views, we require that if an invisible commit occurs, the remaining core members must behave consistently. So, every invisibly committed update must be detectable by every configurer. We can ensure this only if all initiators (whether a *mgr* or a reconfigurer) attempt to install the x^{th} group view for the requisite

majority responses from among the same set of processes.

The Algorithm

The reconfiguration algorithm requires three phases in the worst-case scenario.

- **Phase I**

  The initiator I multicasts a reconfiguration message $reconfig(view(I))$ to its local view. The reconfigurer then awaits responses from the outer processes. Upon receipt of $reconfig(view(I))$, a core member that is lagging behind I adopts I's local view as its own. Every core member, whether or not it has updated its local view, responds to the reconfigurer with its current local view.

  If a majority of core members respond, then I uses the information it receives to determine an updated value (v) and version number (x), whose execution would result in a new group view.

- **Phase II**

  The reconfigurer multicasts the predetermined values as a *reconfiguration submit* message $recsubmit(< v, x >)$. The core members acknowledge the $recsubmit(< v, x >)$; a majority reply is required.

- **Phase III**

  After the $recsubmit(< v, x >)$ acknowledgment is received, I multicasts a *reconfiguration commit* message $reccommit(< v, x >)$.

The Initiator Selection

One way to select the initiator and the new *mgr* is to use a deterministic approach, based on seniority in the group view. For example, older core members can be

ranked higher. Whenever a process is removed from the group view, the ranks of all higher ranked processes are decreased by one. A process initiates a reconfiguration when it believes all other higher ranked processes are faulty.

The Secondary Partition

As described earlier, a network partition may result in primary and secondary partitions or just secondary partitions. The application correctness criteria will determine if any form of execution can exist in the secondary partition. If any similar execution would be permitted, then additional steps be performed.

In the group management update algorithm, we have described how the algorithm will inform the members wether the partition is a primary or a secondary partition. The partition type indicates wether the members should proceed with a full or a restricted execution. The reconfiguration algorithm must also be modified because the *mgr* failure detection may be the result of a network partition that does not contain the *mgr* as one of its members. When the reconfigurer discovers that it does not have a majority of processes, it assumes a secondary partition and try to create a new core quorum. The new quorum will be marked as a secondary quorum and after the reconfiguration algorithm is executed the members are informed that they are in a secondary partition under a restricted execution.

This approach allows different partitions to resume execution if this execution does not violate the correctness criteria of the applications.

9.4 Conclusion

In this chapter, we have discussed the different failures that our protocol should expect. We have presented different approaches to tolerate such failures. Our failure model assumes that all failure are benign. Among the failures discussed

are multicast/send omission, receive omission, and network omission failures. We have shown how the protocols detect these failures through the timestamp scheme used. We have also presented different approaches to detect failures by using either simple timeout procedures or failure detection algorithms. We have also presented an approach to handle retransmission requests and retransmission buffers. Our approach assumes the possibility of both TFM and active processes. Recover from a TFM failure involves a reformation phase for the group and a new TFM election. Our fault-tolerance module is also resilient to network partitioning. We have presented an approach that allows execution to continue in different partitions based on the application's correctness criteria. Our protocol is resilient to a failure of $(n-1)/2$ active processes, where n is the number of processes.

Chapter 10

Conclusion

Ordered reliable multicasting is a common activity in distributed computing. Groups of processes that perform a distributed application interact by multicasting intra-group and inter-group messages. If the physical communication layer is comprised of a set of interconnected LANs, then members of one process group may not belong to a single LAN. Different process groups may adopt different ordering criteria for delivering messages to group members. Also, due to performance and reliability constraints, an application carried out by a given process group may dictate a specific multicasting protocol for intra-group messages. Unfortunately, protocols that are capable of handling all of these problems are beyond the current state of the art. In this thesis, we have investigated the subject of reliable ordered multicasting in a heterogeneous interconnected group of LANs, in which both intra-group and inter-group messages bridge several LANs. Our research efforts have resulted in the development of a protocol suite that supports a reliable ordered delivery service for both local and global messages. Characteristic to our protocol is a communication structure that can be aligned with the actual routing topology, which largely minimizes the number of protocol messages that need to be sent. The protocols depend on forcing the communication between the processes

to follow the communication structure. The benefits of this structure are twofold. First, it enables the communication structure to be potentially aligned with the internet routing topology, which minimizes the number of protocol messages. Second, because of this alignment, the protocol can exercise control over its routing scheme, which decreases the actual number of multicasted messages. The protocol suite developed honors a multiorder message delivery, which allows each group to select its own ordering criteria. It also provides an interoperability framework that allows the interchange of messages with local multicasting protocols while it honors a predetermined order between global and external messages.

Protocol	Causal	Total	Different	Multiple	Interoperability
Token		✔			
ISIS	✔	✔	✔		
Propagation	✔				
Consensus		✔			
Psync	✔				
Trans−Total		✔			
INTER	✔	✔		✔	✔

Figure 10.1: Comparison between existing reliable multicasting protocols.

In this chapter, we review the different protocols developed and summarize the main contributions. We also outline several directions for future work.

10.1 Multicasting Protocols

In this section, we briefly describe the developed protocols and summarize the key features of each.

10.1.1 BUS and BUS-TO Protocols

The Bottom-Up Stamping (BUS) protocol is a reliable ordered multicasting protocol developed to target the needs of distributed applications that are executed in an interconnected network. The BUS protocol ensures causal-order delivery among multicasted messages and relies on the communication structure presented earlier. This protocol is useful for many distributed applications that do not require total order.

The Bottom-Up Stamping - Total Order (BUS-TO) protocol is a reliable ordered multicasting protocol that is based on an idea similar to the BUS protocol. The BUS-TO protocol guarantees a total order for message delivery that honors the potential causality of the messages. The protocol is a tool for distributed applications that require total ordering to achieve correct execution. Among these applications are replica control and stock exchange, in which a total order is required to relax some of the design constraints on the distributed system. The BUS-TO protocol is subject to a higher message delivery delay than the BUS protocol. It is also subject to delays in the delivery of local messages because earlier global messages have traveled to their LCAs for timestamping and have not returned.

The third protocol, the Top-Down Stamping (TDS) protocol, eliminates the message blocking in the BUS-TO protocol and guarantees both total and causal order. The protocol improves the message delivery time of both local and global messages. The TDS protocol uses the same communication structure presented

earlier but uses a different approach by directing the message immediately to its LCA; the message is then multicasted down the hierarchy.

The TDS protocol clearly achieves a lower delay than the BUS-TO protocol and is best suited to applications in which total order is required. However, in cases for which the total order is not needed, the BUS protocol can be used. This protocol ensures a lower delivery time, and fewer messages are required for delivery. The BUS-TO protocol can be used with the BUS protocol within the same communication structure among a group of processes. The combination of the BUS and BUS-TO protocols allows a multiorder among multicasted messages, based on the recipient's needs. The allowance of both total and causal order within the same group is useful for those applications in which total order is necessary for some but not all of the subgroups. The possibility of interfacing both protocols to achieve this ordering scheme may help maintain the site autonomy that is involved in multicasting.

The preliminary performance study conducted showed that the BUS and BUS-TO protocols provide a viable option for ordered multicasting in both point-to-point and multicast networks. The simplified models that are provided in this work clarify this assertion by comparing the extra protocol messages that are required and the message delivery delay incurred for a number of protocol types. The one drawback to our approach is the extra cost needed to build the communication structure.

10.1.2 The MLMO Protocol

The Multi-LAN Multi-Order (MLMO) protocol is developed to support multicasting in heterogeneous distributed systems. Our protocol insists on a hierarchical structure in the communication topology. The protocol uses the same communication model outlined in Chapter 3, with the communication hierarchy shown

in Figure 6.1. Members of one group can be individual processes and/or other groups. The protocol does not restrict the members of a group to the same LAN. Additionally, the protocol allows each group to determine a causal or total ordering criterion. Therefore, our multicasting environment contains two types of groups: the *causal groups* that enforce a causal order and the *total groups* that enforce a total order. Our protocol can circulate messages that have some destinations in total groups and other addressees in causal groups yet can still observe the particular ordering criterion for each addressee's group. Note that a given group's ordering criterion pertains to members that are individual processes and not members that are groups because the later would, by definition, have their own criteria.

The protocol relies on a modified version of the BUS and BUS-TO protocols. It achieves a degree of latency, depending on the ordering criteria adopted. For example, groups that adopt causal order under MLMO do not incur unnecessary delay in message delivery because other groups adopt total order. The protocol performance is affected by the ratio of intra-group to inter-group traffic; the protocol performs better for larger ratios.

10.1.3 The INTER Layer

The INTER layer is a new approach for allowing interoperability between our protocol suite and existing multicasting protocols. The INTER interface is built around the assumption that different CUs, each with a different ordering criterion, can coexist. It is added as a layer between the applications and the multicasting layer. The added layer achieves an order among messages going to and from a CU, independent of the particular multicasting protocol that is running. These local protocols effectively handle all messages they receive as local messages to their CU. Therefore, a local protocol can function autonomously in performing multicasting in its own CU.

The INTER layer provides an interoperability framework that allows message interchange with local multicasting protocols, while it honors a predetermined order between global and external messages. Whether this predetermined order is causal or total is selected based on the needs of the application. This INTER layer can accommodate the coexistence of multiple heterogeneous intra-group multicasting protocols. Specifically, it is an encapsulation protocol that effectively connects any protocol that is performing multicasting in a process group to the MLMO. An added feature with INTER is the elimination of the necessity to alter the local multicasting protocols. This essentially enables MLMO to achieve interoperability of multiple intra-group multicasting protocols, such that full autonomy remains upheld.

A comparison between INTER and the relevant atomic broadcast protocols reported in the literature presented in Section 2.4 is shown in Figure 10.1. In the figure, the major features of seven protocols are compared, including order achieved (total or causal), the capability of achieving different orders for independent message streams, the possibility of interacting with other reliable multicasting protocols, and capability of multiple-order coexistence between dependent messages based on recipient groups. The figure shows the features provided by INTER in achieving multiple order and interoperability. These features although required by our environment, are not provided by the existing protocols.

10.2 Future Extension

Our research has uncovered a number of important areas for future work in multicasting in interconnected LANs. In this section, we outline some of these areas.

10.2.1 Building the Communication Structure

An algorithm that builds the communication structure by considering the cost and frequency of communication between processes needs to be devised. The dynamic characteristics of the communication need to be considered. This will require a dynamic algorithm that can reconfigure the CU membership, as well as the hierarchical structure, during execution.

We also see a need for some development and improvement in the area of building the communication hierarchy. We believe that improvement can be achieved by studying the communication patterns between the groups and allowing this factor to influence the construction of the structure. The algorithm must be able to restructure the communication hierarchy to provide adaptability that can cope with the variations in communication pattern.

We have introduced a solution to multicasting to intersected groups by changing the definition of the LCA (the LCA is the node that manages all the nodes in the intersecting targeted groups). Additional research may be required to identify whether or not dynamic restructuring will introduce a more effective approach to solve this problem.

10.2.2 Prototype of the Protocols

Among the chief issues still to be addressed as a follow-on to our work is the implementation. The development of prototype protocols is a major task that would provide a new platform for the development of applications and for extending the applicability domains of existing ones. During the different phases of our research, we considered several implementation issues that would ease some of the pre-implementation tasks. Prototyping will provide a testbed for the proposed approach and will enable conduct of an accurate performance study to test the

applicability of the protocols.

10.2.3 Multi-Order Support

An interesting question we received in regard to our research [80, 79] was why we didn't incorporate more order to the MLMO protocol (i.e., why didn't we provide for *no order*?) Specific distributed applications require the relaxation of the ordering restrictions and the use of the multicasting criteria of our protocol to interact with existing groups. We realize that several other orders, such as no order, FIFO, and FIFO Atomic, maybe required to co-exist along with total and causal order. This diversity is dictated by the needs of the distributed applications and the autonomy of the system and will require the development of more protocols that can use our communication structure and can provide an interface to fit within the MLMO environment.

10.2.4 Interoperability and the Interface

An investigation of the use of the MLMO protocol with its interface as a platform to provide interoperability between several external protocols is called for. The INTER interface in its current form can provide this interoperability service. However, we believe that if the combination of INTER and MLMO were dedicated to providing an interoperability framework, then the design of a special interface for each local protocol could provide a more efficient interface than the general protocol we provided. The interoperability framework we have provided will allow a similar interface to be incorporated. These interfaces could, by utilizing the knowledge about the local protocol and the data structure, be capable of achieving a higher performance than our general interface.

10.2.5 Network Partitions

The management of network partitions is one of the major areas that needs more research. We believe that the definition of atomicity can be modified based on specific application needs. The research in this area has resulted in a set of multicasting protocols that either stop in the event of partitions or allow execution to proceed in one partition and halt activity in the remaining partitions. These approaches may be adequate for applications that require a consensus among the whole group to achieve a correct execution, such as replica control. The approach we adopted to handle partitions allows the larger partition to proceed while all minor partitions resume a restricted execution after group reformation. A large number of distributed applications do not require similar correctness criteria. In this case, reforming the group membership in each partition and allowing an unrestricted execution will allow a similar operability in the event of multiple partitions. This approach will be acceptable if a majority is not required over the original group membership to achieve correct execution, such as group chat and e-mail. Some of these applications require notification that the group has been reformed and notification of new membership.

Bibliography

[1] A. E. Abbadi, D. Skeen, and F. Cristian. "An Efficient Fault Tolerant Protocol for Replicated Data Management". *Proc. of the ACM 4th Annu. Conf. Principles Database Syst.*, 1985.

[2] A. E. Abbadi and S. Toueg. "Maintaining Availability in Partitioned Replicated Databases". *ACM Trans. on Database Systems*, 14(2):264–290, Jun. 1989.

[3] H. Abdel-Wahab and M. Feit. "XTV: A Framework for Sharing X Window Clients in Remote Synchronous Collaboration". *Proceedings of the IEEE Tri-Comm 91*, Apr. 1991.

[4] L. Aguillar. "Datagram Routing For Internet Multicasting". *ACM Computer Communications Review*, 14(2):58–63, 1984.

[5] C. Alaettinoğlu, K. Dussa-Zieger, I. Matta, and A.U. Shankar. "MaRS (Maryland Routing Simulator) - Version 1.0 User's Manual". Technical Report UMIACS-TR-91-80, CS-TR-2687, Department of Computer Science, University of Maryland, College Park, Jun. 1991.

[6] R. Bazzi and G. Neiger. "Simulating Crash Failures with many Faulty Processors ". *A. Segal and S. Zaks, editors, Proceedings Proceedings of the Sixth International Workshop on Distributed Algorithms*, Nov. 1992.

[7] P. A. Bernestein and N. Goodman. "Multiversion Concurrency Control". *ACM Trans. on Database Systems*, 8(4), Dec. 1983.

[8] O. Biran, S. Moran, and S. Zaks. "A Combinatorial Characterization of the Distributed Tasks that are Solvable in the Presence of one Faulty Processor". *Proceedings of the Seventh ACM Symposium on Principles of Distributed Computing*, pages 263–275, Aug. 1988.

[9] K. Birman. "The Process Group Approach to Reliable Distributed Computing". Technical Report 1216, Dept. of Computer Science, Cornell Univ., 1993.

[10] K. Birman. Private communication. Nov. 1990.

[11] K. P. Birman and T. A. Joseph. "Exploiting Virtual Synchrony in Distributed Systems". *Proceedings of ACM Symposium on Operating System Principles*, pages 123–138, 1987.

[12] K. P. Birman and T. A. Joseph. "Reliable Communication in the Presence of Failures". *ACM Transactions on Computer Systems*, 5(1):47–76, Feb. 1987.

[13] K. P. Birman and T. A. Joseph. "Exploiting Replication in Distributed Systems". *S. Mullender (editor), Distributed Systems, ACM press*, pages 319–368, 1989.

[14] A. D. Birrell, R. Levin, R. M. Needham, and M. Schroede r. "Experience with Grapevine: The Growth of a Distributed System". *ACM Transactions on Computer Systems*, pages 2–23, Feb. 1984.

[15] G. Bracha. "Asynchronous Byzantine Agreement Protocols ". *Information and Computation*, 75(2):130–143, Nov. 1987.

[16] M. Bridgland and R. Watro. "Fault-Tolerant Decision Making in Totally Asynchronous Distributed Systems". *Proceedings of the Sixth ACM Symposium on Principles of Distributed Computing*, 1987.

[17] T. Chandra. "Unreliable Failure Detectors For Asynchronous Distributed Systems.". Technical report, Dept. of Computer Science, Cornell Univ., May 1993. Ph.D. Thesis.

[18] T. Chandra, V. Hadzilacos, and S. Toueg. "The Weakest Failure Detector for Solving Consensus". *Submitted for Publication*, Apr. 1994.

[19] J. M. Chang and N. F. Maxemchuck. "Reliable Broadcast Protocols". *Reliable Broadcast Protocols. ACM Transactions on Computer Systems*, 2(3):251–273, Aug. 1984.

[20] D. Cheriton. Private communication. Dec. 1990.

[21] B. Chor and C. Dwork. "Randomization in Byzantine Agreement". *Advances in Computer Research*, 5:443–497, 1989.

[22] D. Comer. *"Internetworking with TCP/IP Principles, Protocols, and Architecture"*. Prentice Hall., 1988.

[23] D. Comer. *"Internetworking with TCP/IP"*, volume II. Prentice Hall, second edition, 1990.

[24] John Corbin. *"The Art of Distributed Applications"*. Springer-Verlag, 1990.

[25] P. J. Courtois. *"Decomposability: Queueing and Computer System Applications"*. Academic Press, New York, 1977.

[26] F. Cristian. "Reaching Agreement on Processor Group Membership in Synchronous Distributed Systems". Technical Report RJ5964, IBM Research Report, Mar. 1988.

[27] F. Cristian, H Aghili, H. Raymond, and D. Dolev. "Atomic Broadcast: From Simple Message Diffusion to Byzantine Agreement". *Proceedings of the Fifteenth International Symposium on Fault-Tolerant Computing*, Jun. 1985.

[28] F. Cristian, H. Aghili, R. Strong, and D. Dolev. "Atomic Broadcast: From Simple Message Diffusion to Byzantine Agreement". Technical Report RJ5244, IBM Research Report, Jul. 1986.

[29] P. Danzig. "Finite Buffers and Fast Multicast". *Proceedings of the ACM Conference on Measurement and Modelling of Computer Systems*, Aug. 1989.

[30] S. B. Davidson. "Optimism and Consistency in Partitioned Distributed Database Systems". *ACM Transactions on Database Systems*, 9(3):456–481, May 1984.

[31] S. Deering and D. Cheriton. "Multicast Routing in Datagram Internetworks and Extended LANs". *ACM Trans. on Comp. Sys.*, pages 85–110, May 1990.

[32] S. E. Deering. "Multicast Routing in Internetworks and Extended LANs ". *ACM Computer Communications Review*, 18(4):55–64, 1988.

[33] R. Dirvin and A. Miller. "The MC68824 Token Bus Controller: VLSI for the Factory LAN". *IEEE Micro Magazine*, 6:15–25, Jun. 1986.

[34] D. Dolev, C. Dwork, and L. Stockmeyer. "On the Minimal Synchronizm Needed for Distributed Consensus". *J. ACM*, 34(1):77–97, Apr. 1987.

[35] C. Dwork, N. Lynch, and L. Stockmeyer. "Consensus in the Presence of Partial Synchrony". *J. ACM*, 35(2):288–323, Aug. 1988.

[36] A. D. Fekete. "Asynchronous Approximate Agreement". *Information and Computation*, To Appear, 1994.

[37] M. Fischer, N. Lynch, and M. Paterson. "Impossibility of Distributed Consensus with One Faulty Process". *J. ACM*, 32(2):374–382, Apr. 1985.

[38] A. Frank, L. Wittle, and A. Bernstein. "Multicast Communication on Network Computers". *IEEE Software*, 2(3):49–61, May 1985.

[39] H. Garcia-Molina and B. Kogan. "An Implementation of Reliable Broadcast Using an Unreliable Multicast Facility". Technical Report CS-170-88, Princeton Univ., 1988.

[40] H. Garcia-Molina and B. Kogan. "Node Autonomy in Distributed Systems". *Proceedings of the IEEE International Symposium on Databases in Parallel and Distributed Systems*, pages 158–166, Dec. 1988.

[41] H. Garcia-Molina and A. Spauster. "Ordered and Reliable Multicast Communications". *ACM Transactions on Computer Systems*, 9(3), Aug. 1991.

[42] I. Greif. *"Computer-Supported Cooperative Work: A Book of Readings"*. Morgan Kaufman Pub. Co., 1988.

[43] V. Hadzilacos. "Issues of Fault Tolerance in Concurrent Computations". Technical Report TR11-84, Dept. of Computer Sc., Harvard University, Jun. 1984.

[44] V. Hadzilacos and S. Toueg. "A Modular Approach to Fault-Tolerant Broadcasts and Related Problems". *Submitted for Publication*, June 1994.

[45] M. Herlihy. "Optimistic Concurrency Control for Abstract Data Types". *Proceedings of the 5th Symposium on Principles of Distributed Computings*, pages 206–217, 1986.

[46] A. Heybey. "The Network Simulator". Lab. of Comp. Sc., Massachusetts Institute of Technology, Oct. 1989.

[47] H. Ishida and L. Landweber. "Introduction, Special Issue on Internetworking". *ACM Communications*, pages 28–30, Aug. 1993.

[48] L. Kleinrock. "Scheduling, Queueing, and Delays in Time-Shared Systems and Computer Networks". *N. Abramson and F. F. Kuo (eds.), Computer Communications Network, Prentice-Hall*, pages 95–141, 1973.

[49] Hisashi Kobayashi. *"Modeling and Analysis: An Introduction to System Performance Evaluation Methodology"*. Addison-Wesley, 1979.

[50] L. Lamport. "Using Time Instead of Timeout for Fault-Tolerant Distributed Systems". *ACM Transactions on Computer Systems*, 6(2):254–280, Apr. 1984.

[51] T. LeBlanc and R. Cook. "High-Level Broadcast for Local Area Networks". *IEEE Software*, 2(3):40–48, May 1985.

[52] B. Liskov, D. Curtis, P. Johnson, and R. Scheifler. "Implementation of Argus". *Proceedings of the 11th Symposium on Operating System Principles*, pages 111–122, Nov. 1987.

[53] S. Luan and V. Gligor. "A Fault-Tolerant Protocol for Atomic Broadcast". *IEEE Transactions on Parallel and Distributed Systems*, 1(3):271–285, Jul. 1990.

[54] N. Lynch, B. Blaustein, and M. Siegel. "Correctness Conditions for Highly Available Replicated Databases". *Proceedings of 5th ACM Symposium on Principles of Distributed Computing*, pages 11–28, Aug. 1986.

[55] E. Mayer. "An Evaluation Framework for Multicast Ordering Protocols". In *SIGCOMM '92*, pages 177–187, Baltimore, Maryland, Aug. 1992.

[56] P. M. Melliar-Smith, L. E. Moser, and V. Agrawala. "Broadcast Protocols for Distributed Systems". *IEEE Transaction on Parallel and Dist. systems*, pages 17–25, Jan. 1990.

[57] R. Metcalf and D. Boggs. "Ethernet: Distributed Packet Switching for Local Computer Networks ". *Communication ACM*, 19:395–404, Jul. 1976.

[58] L. E. Moser, P. Melliar-Smith, and V. Agrawala. "On the Impossibility of Broadcast Agreement Protocols". submitted for publication.

[59] L. E. Moser, P. M. Melliar-Smith, and V. Agrawala. "Asynchronous Fault-Tolerant Total Ordering Algorithms". *Submitted for Publications.*

[60] S. Mullender, editor. *"Distributed Systems"*. ACM press, second edition, 1993.

[61] C. H. Papadimitrio. "The Serializability of Concurrent Database Updates". *J. ACM*, 26(4):631–633, Oct. 1979.

[62] K. J. Perry and S. Toueg. "Distributed Agreement in the Presence of Processor and Communication Faults". *IEEE Transaction on Software Eng.*, SE-12(3):447–482, Mar. 1986.

[63] L. Peterson, N. C. Buchholz, and R. D. Schilichting. "Preserving and Using Context Information in Interprocess Communication". *ACM Transactions on Computer Systems*, 7(3):217–246, Aug. 1989.

[64] B. Rajagopalan. "Reliability and Scaling Issues in Multicast Communication". In *SIGCOMM '92*, pages 188–198, Baltimore, Maryland, Aug. 1992.

[65] A. Ricciardi. "The Group Membership Probelm in Asynchronous Systems". (Ph.D. Thesis, 92-1313), 1992.

[66] A. Ricciardi and K. Birman. " Process Membership in Asynchronous Environments". Technical Report 93-1328, Dept. of Computer Science, Cornell Univ., 1993.

[67] A. Ricciardi, A. Schiper, and K. Birman. "Understanding Partitions and the No Partitions Assumptions". Technical Report 93-1355, Dept. of Computer Science, Cornell Univ., 1993.

[68] F. B. Schmuck. "The Use of Efficient Broadcast Protocols in Asynchronous Distributed Systems". (Ph.D. Thesis, TR88-928), 1988.

[69] M. D. Schroeder. "A State-of-the-Art Distributed System: Xomputing with BOB". *S. Mullender (editor), Distributed Systems, ACM press*, pages 1–16, 1993.

[70] A. Sheth and J. Larson. "Federated Database Systems for Managing Distributed Heterogeneous and Autonomous Databases". *ACM Computing Surveys*, 22(3), Sep. 1990.

[71] D. Skeen. "A Formal Model for Crash Recovery in a Distributed System.". *IEEE Trans. Software Eng.*, SE-9(3):219–228, May 1983.

[72] A. Tanenbaum. *"Computer Networks"*. Prentice Hall, second edition, 1988.

[73] R.H. Thomas. "A Majority Consensus Approach to concurrency Control for Multiple Copy Databases". *ACM Trans. on Database Systems*, 4:180–209, Jun. 1979.

[74] K. S. Trivedi. *"Probability & Statistics with Reliability, Queuing, and Computer Science Applications"*. Prentice-Hall, 1982.

[75] V. L. Wallace and R. S. Rosenberg. "Markovian Models and Numerical Analysis of Computer Systems Behavior". *Proceedings of AFIPS Spring Joint Computer Conference*, pages 141–148, 1966.

[76] M. Willet. "Token-Ring Local Area Networks - An Introduction". *IEEE Network Magazine*, 1:8–9, Jan. 1987.

[77] O. ZeinElDine and H. Abdel-Wahab. "A Multicasting Protocol Suite for Interconnected LANs". *in preparation.*

[78] O. ZeinElDine, M. Eltoweissy, and H. Abdel-Wahab. "BUS: A Multicasting Protocol for Interconnected LANs". *IEEE Proceedings of the 5th International Conference on Computing and Information*, May, 1993.

[79] O. ZeinElDine, A. Wadaa, and H. Abdel-Wahab. "A New Approach for Ordering Multicasted Messages in Heterogeneous Distributed Systems". *Proceedings of the 6th International Conference on Computing and Information. Also to appear in Journal of Computing and Information*, May 1994.

[80] O. ZeinElDine, A. Wadaa, and H. Abdel-Wahab. "MLMO: A Multi-LANs Multi-Order Multicasting Protocol". *Proceedings of the 32nd ACM Southeast Conference*, Mar. 1994.

GLOSSARY

- *Deliver Queue (DQ):* The queue at which the messages are buffered for delivery to the process.
- *Local Wait Queue (LWQ):* The queue at which the local messages are kept waiting for the messages that are assigned smaller timestamps to arrive if any are missing.
- *Global Wait Queue (GWQ):* The queue at which the global messages are kept waiting for the messages that are assigned smaller timestamps to arrive if any are missing.
- *Out-of-Order Queue (OOQ):* The queue at which the messages that arrive out of order from the same process are kept until the late or lost message(s) arrive. In the case of a TFM process, the messages kept in this queue have not been assigned a timestamp from the receiving TFM yet.
- $MTS_{m_x}[\ ]$: The timestamp vector that accompanies the message m_x and carries the timestamps assigned to it by the sender and the different TFM processes it passes.
- $PTS_{p_x}[\ ]$: The timestamp vector that is used by the process p_x to keep track of the message timestamp last delivered from the different TFMs.
- LTS_{p_x}: The local timestamp variable used by process p_x to stamp the messages sent, received, or passing by.
- $OLDTS_{p_x}$: The timestamp variable that records the timestamp of the last message sent from p_x to its TFM $\mathcal{T}_x$.

- $TWL_{\mathcal{T}_x}$: A list that contains the messages timestamped by the TOCU-TFM before they gain their LCA's timestamp.

- $LCAM_{\mathcal{T}_x}$ and $CLCAM_{\mathcal{T}_x}$: The Least Common Ancestor Messages ($LCAM_{\mathcal{T}_x}$) contains all messages for which $LCAM_{\mathcal{T}_x}$ acts as their LCA. The LCAM is a temporary list on which messages reside until they are committed for delivery. The messages in $LCAM_{\mathcal{T}_x}$ are waiting for messages that have received smaller timestamps from $\mathcal{T}_x$ and have not come back with LCA timestamps. The Committed LCAM list ($CLCAM_{\mathcal{T}_x}$) contains those messages of the LCAM that have been committed by $\mathcal{T}_x$; The $CLCAM_{\mathcal{T}_x}$ is carried with any message traveling down its one-way path.

- $TSUL_{\mathcal{T}_x}$: The *Timestamp Updater List* (TSUL) is added at each TOCU-TFM. Any message on its way down the hierarchy as it passes by any of its $\mathcal{OWA}$ paths, is assigned a timestamp. The message adds an entry to the TSUL after it has been timestamped. This list is used for messages traveling along their TW or $\mathcal{OWB}$ paths to adjust $PTS_{p_x}[\,]$.

- **SENDER**(m_k): The protocol procedure that is activated when a process sends a message m_k.

- $\mathcal{T}_i$: The Timestamping, Multicasting, and Forwarding (TFM) process.

- **RECEIVE**(m_k): The protocol procedure that is activated when a message is received at any of the processes involved in multicasting.

- **TFM**(m_k): The protocol procedure that is activated when a message is sent or received at any of the TFM processes.

- **VM**: The *View Maintainer* is a process that maintains a view of the group membership. This process along with the other VMs cooperate in defining the *Global View* of the group.

- GV: The *Global View* is the view agreed upon by all VMs to describe the membership of the group at a certain point in time.
- VML: The *VM list* contains all the VMs of the group.