Totem: A Reliable Ordered Delivery Protocol for Interconnected Local-Area Networks

Deborah A. Agarwal

UNIVERSITY OF CALIFORNIA
Santa Barbara

Totem: A Reliable Ordered Delivery Protocol for Interconnected Local-Area Networks

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

Doctor of Philosophy
in
Electrical and Computer Engineering
by

Deborah A. Agarwal

Committee in charge:
Professor Louise E. Moser, Chairperson
Professor P. M. Melliar-Smith
Professor Roger Wood
Professor Amr El Abbadi

August 1994

The Dissertation of Deborah A. Agarwal
is approved:

Committee Chairperson

August 1994

29 August, 1994

ACKNOWLEDGMENTS

The road to completing my doctorate has been a long one with a multitude of hazards and turnoffs along the way. I have many people to thank for helping me down that road.

Firstly, I would like to thank my advisors Dr. Moser and Dr. Melliar-Smith for their guidance, support and roadmaps. I would also like to thank my committee members for their time and patience.

The students in my research lab have been excellent passengers and can ride with me any time. I particularly want to thank Marcos and Ravi for navigating and for sharing in the driving. I appreciate the companionship provided by Amitabha, Ramki, George, and Amy on the road. There will always be room in the car for them.

Thanks also to Hugh, Paul, Yair, Michael, Ravi, Tom, Amy and Wes for the mechanical support, supercharging the engine and detailing the car. People stop and stare in every town.

I appreciate the time and effort of the people I met at Lawrence Berkeley Laboratory during my internship there. In particular, I would like to thank Sally and Carol. It was a most excellent adventure.

The National Park Service receives my special gratitude for providing the much needed rest stops along the highway.

Most importantly, I would like to thank my friends particularly Kathy and Ed who stood by me through thick and thin and even pushed the car when it got stuck in the mud. A roadtrip will never be the same without them.

Last but not least, I would like to thank my family for keeping me between the lines.

This work was supported by the National Science Foundation, Grant No. NCR-9016361 and by the Advanced Research Projects Agency, Contract No. N00174-93-K-0097. Financial support for this research was also provided by the Dissertation Year Fellowship and the Graduate Research Mentorship Program at the University of California, Santa Barbara.

VITA

1985	B.S., Mechanical Engineering, Purdue University.
1985-88	Project Engineer, General Motors Technical Center Warren, Michigan.
1988-90	Teaching Assistant, Department of Mechanical Engineering Department of Electrical and Computer Engineering, University of California, Santa Barbara.
1991	M.S., Electrical and Computer Engineering University of California, Santa Barbara.
1992	Lecturer, Department of Electrical and Computer Engineering, University of California, Santa Barbara.
1989-94	Researcher, Department of Electrical and Computer Engineering, University of California, Santa Barbara.
1993-94	Instructor, Technical Research Associates, Camarillo, California.
1993-94	President's Dissertation Year Fellowship, University of California.

PUBLICATIONS

- "The Totem Multiple-Ring Ordering and Topology Maintenance Protocol," with L. E. Moser, P. M. Melliar-Smith and R. Budhia, in preparation.

- "Reliable Ordered Delivery in Interconnected Local-Area Networks," with L. E. Moser, P. M. Melliar-Smith and R. Budhia, submitted for publication.

- "The Totem Single-Ring Ordering and Membership Protocol," with Y. Amir, L. E. Moser, P. M. Melliar-Smith and P. Ciarfella, submitted for publication.

- "The Totem Protocol Development Environment," with P. Ciarfella, L. E. Moser and P. M. Melliar-Smith, *Proceedings of the 1994 International Conference on Network Protocols*, Boston, MA (October 1994), 168-177.

- "Extended Virtual Synchrony," with L. E. Moser, Y. Amir and P. M. Melliar-Smith, *Proceedings of the 14th IEEE International Conference on Distributed Computing Systems*, Poznan, Poland (June 1994), 56-65.

- "Debugging Internet Multicast," with S. Floyd, *Proceedings of the 22nd ACM Computer Science Conference*, Phoenix, AZ (March 1994), 22-29.

- "Fast Message Ordering and Membership Using a Logical Token-Passing Ring," with Y. Amir, L. E. Moser, P. M. Melliar-Smith and P. Ciarfella, *Proceedings of the 13th International Conference on Distributed Computing Systems*, Pittsburgh, PA (May 1993), 551-560.

- "Totem: A Protocol for Message Ordering in a Wide-Area Network," with P. M. Melliar-Smith and L. E. Moser, *Proceedings of the First International Conference on Computer Communications and Networks*, San Diego, CA (June 1992), 1-5.

- "A Graphical Interface for Analysis of Communication Protocols," with L. E. Moser, *Proceedings of the 20th ACM Computer Science Conference*, Kansas City, MO (March 1992), 149-156.

- "Ring-Based Ordering Protocols," with P. M. Melliar-Smith and L. E. Moser, *Proceedings of the International Conference on Information Engineering*, Singapore (December 1991), 882-891.

FIELDS OF STUDY

Major Field: Computer Engineering

Studies in Fault-Tolerant Distributed Systems
Professors L. E. Moser and P. M. Melliar-Smith

ABSTRACT

Totem: A Reliable Ordered Delivery Protocol
for Interconnected Local-Area Networks

by

Deborah A. Agarwal

Many recent computer applications have been designed to execute in a distributed computer system because a distributed system has the potential to provide high availability and excellent performance at a low price. However, due to the need to coordinate tasks and share data among processors, programming the application is often difficult and, thus, this potential is not often realized. A communication protocol that provides reliable totally ordered delivery of messages in a distributed system can greatly simplify the application programmer's task.

This dissertation describes a reliable delivery and total ordering protocol, called Totem, that models a communication network as broadcast domains connected by gateways. Processors within a broadcast domain communicate by broadcasting messages. Access to the communication medium is controlled by a token, and reliable delivery is achieved by the use of sequence numbers in messages. Gateways forward messages between rings. Timestamps in messages provide totally ordered message delivery that respects causality and is consistent across the entire network.

A membership algorithm provides recovery from processor failure and network partitioning, as well as loss of all copies of the token. When a failure occurs, the membership algorithm forms a new membership and regenerates the token so that normal operation can resume. The gateways maintain network topology information that is updated when a membership change occurs. Changes in the membership and in the topology are provided to the application in order with respect to the other messages in the system.

Contents

List of Figures

Chapter 1

Introduction

1.1 Motivation

Many recent computer applications have been designed to execute in a distributed computer system because a distributed system has the potential to provide high availability and excellent performance at a low price. Application tasks can be divided among the processors in the system, and data can be replicated to protect against failures. However, due to the difficulty of coordinating tasks and sharing data among processors, the potential provided by a distributed system is not often realized.

Traditionally, distributed systems have been designed to be synchronous, because synchronous systems allow simple consensus protocols to be used to maintain the consistency of replicated data in the presence of faults. The Mars system provides one example of a synchronous distributed system that has been built [34, 35]. Unfortunately, large systems are necessarily partially or entirely asynchronous. Existing protocols to maintain consistency in such systems are inefficient. A reliable ordered delivery protocol can provide efficient fault-tolerant consensus and simplify the application programmer's task.

Applications developed for distributed systems include process control systems, database systems, and cooperative work tools. One application is manufacturing process control. Modern factories consist of automated workcells connected by conveyors or automated guided vehicles (AGVs). Each workcell is capable of performing several tasks and each part visits workcells as needed to complete the processing. The individual tasks performed by a workcell are controlled by one or more computers in the workcell. A scheduling system determines how parts should be scheduled based on the availability of workcells and raw materials. The computers in the workcells, the conveyor or AGVs, and the scheduler must coordinate their activities by passing messages on a communication network. The distributed sites can coordinate tasks more easily if the communication protocol provides reliable ordered delivery of messages.

Another distributed system application, a multiparty whiteboard, allows multiple users to hold a meeting and interact on a virtual whiteboard. The whiteboard, available on the Internet, uses unreliable multicast packets to communicate between sites running the whiteboard. The unreliable multicast is an experimental capability recently added to the Internet [15]. Coherent interaction on the shared whiteboard requires reliable ordered communication of changes to the whiteboard.

Database applications also use distributed systems to provide fault tolerance by allowing redundant processing and replicas of the data. Replicated databases typically employ a client/server paradigm in which multiple servers serve a client and each of the servers holds a copy of the data. When the data are updated, a copy of the update message must be sent to each of the servers. These update messages must be processed in a consistent serializable order; otherwise, inconsistencies in the data can arise. It is particularly critical that all of the servers commit a transaction, or none do. Existing replicated databases use a two-phase commit protocol that blocks if any server fails. A reliable ordered multicast protocol has the potential to achieve reliable commit for fault-tolerant distributed databases that very seldom blocks.

Although it is impossible to provide a reliable ordered delivery that is guaranteed to terminate in a purely asynchronous system (see [45] for one proof), several protocols have been developed for reliable ordered delivery in a partially asynchronous system that have asymptotic termination properties [7, 8, 13, 49].

1.2 The Totem Protocol

The main contribution of this dissertation is a reliable delivery and total ordering protocol, called Totem, that models a communication network as broadcast domains connected by gateways. Processors within a broadcast domain communicate by broadcasting messages. On each broadcast domain is superimposed a logical token-passing ring. Access to the broadcast medium is controlled by a token-passing protocol, and reliable delivery is achieved by the use of sequence numbers in messages. Gateways forward messages between rings. Timestamps in messages provide totally ordered message delivery that respects causality and is consistent across all rings.

Reliable ordered message delivery protocols can provide different levels of message ordering services. We categorize these levels as fifo, partial and total ordering. The fifo level of service provides ordered delivery of messages between any pair of processors but does not place any restrictions on the interleaving of the messages from two different sites at a destination. The partial order service is usually based on the causal order defined by Lamport [36] which ensures that if a message m can have causally affected another message m', then it is ordered before that message. The total order is a partial order in which every message is ordered with respect to every other message.

In the past, the protocol efficiency decreased going from delivering messages in partial order to delivering messages in total order. This was a result of the perceived need to construct the total order from the partial order. The objective of the Totem protocol is to deliver totally ordered messages efficiently, thus making partial ordering unnecessary.

In Totem, messages are broadcast with enough information to establish their position in the total order immediately, and token transfer is not delayed except as needed for flow control. This eliminates the need for acknowledgments of individual messages by each of the processors. The Totem protocol provides virtual synchrony [13] and introduces extended virtual synchrony [44]. Extended virtual synchrony extends the concept of virtual synchrony to systems in which the components of a partitioned system must continue to operate and may subsequently remerge, and also to systems in which failed processors can be

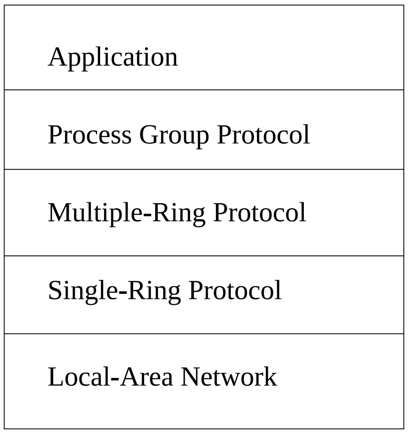


Figure 1.1: The Totem system hierarchy.

repaired and can rejoin the system with stable storage intact. In extended virtual synchrony, two processors may deliver different sets of messages, when one of them has failed or when they are members of different components, but they must deliver messages consistently.

The Totem system is composed of a hierarchy of protocols at each processor and provides reliable ordered delivery of messages network wide (see Figure 1.1). The bottom layer is the local-area network itself which provides message passing between processors. The single-ring protocol provides reliable delivery and ordering of messages within the broadcast domain. The multiple-ring protocol provides reliable delivery and ordering across the entire network. The process group interface to the application is described in [37].

1.3 The Single-Ring Protocol

The single-ring protocol uses a logical token ring to provide reliable ordered delivery of messages to processors in a broadcast domain. The token circulates around the ring as a point-to-point message, and a processor must be in possession of the token to broadcast a message to the processors on the ring. Each message header contains a sequence number derived from a sequence number field in the token. The sequence number field is incremented as each message is broadcast, and the token passes from processor to processor as it circulates around the ring, thereby providing a single sequence of monotonically increasing sequence numbers for the messages broadcast on the ring.

Processors recognize missed messages by detecting gaps in the sequence numbers. When a processor receives the token, it requests retransmissions by inserting the sequence numbers of missing messages into the token's retransmission request field. On receiving the token, a processor that possesses a requested message retransmits the message and removes the request from the retransmission request field in the token. In this way, the single-ring protocol provides the local mechanism for reliable ordered delivery of messages.

On the local ring, a message can be totally ordered as soon as it and all prior messages, as defined by the sequence numbers, have been received. A message can be discarded when it has been received by all processors on the ring.

To provide fault-tolerance, the single-ring protocol is integrated with a membership algorithm that provides recovery from token loss, from processor failure and restart, and from ring partitioning and remerging. A time-out determines token loss, processor failure, and ring partitioning when the token is passed to a failed or disconnected processor, resulting in loss of the token. New or restarted processors and partitioned rings that have recently regained access to the local-area network are detected by the appearance of messages on the communication medium from processors that are not members of the current ring.

The membership algorithm is activated by a token loss timeout or receipt of a message from a processor that is not a member of the current membership. The membership algorithm contains four states. The first state is the Gather state in which the processors attempt to obtain agreement on the membership of the new token ring. If they agree on the membership, they proceed to the Commit state where they establish the ring identifier and rotation pattern for the new token. After the new token has begun circulating, in the Recover state the processors deliver old messages from the previous ring(s) to ensure extended virtual synchrony and then proceed to the normal Operational state. The membership algorithm is a significantly modified version of the algorithm developed for Transis [6].

A new or restarting processor starts in the Gather state. Other processors on the same local-area network also proceed to the Gather state and the membership algorithm normally merges the new processor and the existing ring into a single new ring. In the base case, the processor forms a ring containing only itself.

Each single-ring membership change is delineated by two Configuration Change messages which are ordered with respect to the messages on the ring. The first Configuration Change message lists the processors that are in the intersection of the old and new membership. Residual messages from the old configuration that cannot be delivered in the old configuration are delivered within this configuration. The second Configuration Change message marks the beginning of the new configuration formed by the membership algorithm and lists the membership of the new configuration.

1.4 The Multiple-Ring Protocol

The multiple-ring protocol is executed by each processor and gateway. A gateway interconnects two broadcast domains, and is responsible for forwarding messages between rings, maintaining network topology information, and disseminating local-area failure and join information. Messages are timestamped on generation using Lamport clocks [36] to preserve causality.

As the single-ring protocol delivers messages in total order to a gateway, the gateway forwards the messages in order onto the other ring. When forwarded, a message is given a new sequence number appropriate to the new ring but retains its original timestamp. The single-ring protocol and forwarding mechanisms combine to ensure that messages originating on any one ring are forwarded throughout the network in order. When a membership change occurs on a ring, the resulting Configuration Change messages are forwarded as regular messages. When a processor receives a Configuration Change message, it begins ordering messages received from the new ring. The messages that were originated before the membership change precede the Configuration Change message, and the messages originating after the membership change follow the Configuration Change message.

Since messages are forwarded through the network in order, a processor can determine which messages have been forwarded by observing the messages on its own ring. A processor is guaranteed to have received all messages preceding a message generated on the message's source ring. By keeping track of a timestamp for each ring in the network, a processor can determine a network-wide total order.

Each gateway maintains a view of the network topology. When a Configuration Change message is received, a gateway notes the change in its topology information. When a Configuration Change message for a locally attached ring is the message with the lowest timestamp, each gateway on the ring sends a Network Topology message indicating its current topology information. The Network Topology messages are necessary to ensure that all of the gateways on the new ring start with the same view of the topology of the network. A Topology Change message is sent by a gateway after receipt of Network Topology messages or when a gateway determines that a ring has become disconnected from the topology. Topology Change messages forwarded through the network are used by processors and gateways to learn of the changes in the network topology due to a configuration change.

1.5 Related Issues

Although the Totem protocol is designed to continue despite network partitions, reliable delivery of messages can be provided only between processors in the same partition. Thus, it is desirable to design the network to increase the probability that the network will remain connected despite failures. If we represent the network as a graph where each local-area network is a node and each gateway is an edge, then network connectivity can be analyzed as a graph partitioning problem.

We have investigated graphs of networks that were constructed by adding random edges to the graph and robust graphs that were constructed to be resilient to edge deletion [41]. The data indicate that, as gateways fail, partitioning of the network is of more concern than the increased length of the routes in the network.

To provide efficient broadcast communication, effective flow control is required. Broadcast communication can proceed only at the rate of the slowest processor or communication link. If messages are broadcast at a higher rate for an extended period of time, the slowest participant will experience buffer overflow and begin to drop messages. This results in higher retransmission rates and lower throughput.

The single-ring protocol contains a simple token-based flow-control mechanism which provides good performance. Throughput is higher than achieved by other algorithms and comparable to that achieved by TCP/IP for point-to-point communication. The multiple-ring protocol introduces back-pressure mechanisms across the network to alleviate congestion. Any processor or gateway that is running out of buffer space attempts to block generation of new messages throughout the network for long enough to free buffer space.

The Totem single-ring and multiple-ring protocols have been implemented. The effects of the network layout on fault-tolerance have also been studied through simulations.

The remainder of this dissertation is organized as follows. The related work is presented and discussed in Chapter 2. The environment in which Totem operates and the services provided by Totem are described in Chapter 3. A detailed description of the single-ring protocol can be found in Chapter 4. The multiple-ring protocol and the issues related to interconnecting broadcast domains are covered in Chapter 5. Conclusions and recommendations are given in Chapter 6. Finally, Appendix A contains a guide to compiling and using the Totem implementation and simulation.

Chapter 2

Background

The Totem protocol is designed to provide message passing support for distributed applications. One of the target applications, distributed databases, is primarily concerned with consistent updates of replicated data and consistent operation during partitioning of the system. Replica control protocols orchestrate the reading and writing of copies of a data item in a distributed database.

The task of designing a replica control protocol is particularly difficult if the system might partition leaving some of the copies inaccessible. Several replica control protocols have been developed to support distributed databases on networks that might partition [24, 26, 48]. All of these protocols would benefit from having reliable message delivery, and some of these protocols assume that reliable ordered message delivery is already provided by a communication layer. Some replica control protocols also assume that process failure and partition detection are provided by underlying failure detectors [48, 53].

Algorithms have been developed for maintenance of replicated data in a distributed database, such as [2, 3, 4], that implement mechanisms for causal ordering of messages. If the message order provided by the reliable ordered delivery protocol incorporates causality constraints, these database algorithms can be built using relatively simple application level mechanisms.

Not all distributed application designers accept that totally ordered delivery of messages to a database will simplify the task of maintaining consistency [18]. Some application designers would argue that reliable ordered delivery protocols are inappropriate because, external and semantic causality constraints are not represented in the message order, the database transaction model is not directly implemented, and the protocols are inefficient. This view is relatively narrow since reliable ordered delivery does not preclude the inclusion of external and semantic ordering constraints in the messages; it does, however, eliminate the need to include causality constraints between messages. Although reliable ordered delivery does not directly implement the database transaction model, it has been used successfully to support this model and many recent reliable ordered delivery protocols provide high throughput and low latency of message delivery [11].

The need for an underlying protocol to provide failure detection to the database is apparent; the current database solutions to replica control tend to be complicated and are not often implemented. Reliable ordered delivery protocols that can operate despite network partitions should ease the task of designing and implementing replica control protocols. Since the Totem protocol provides notification of membership changes in order with respect to the messages in the system, the application can focus on other issues [55].

Totem is not the first protocol to be developed in support of distributed applications. Several other protocols provide fault-tolerance and reliable ordered delivery of messages. Many of these protocols have been developed to operate in a synchronous system [22, 34, 35]. Synchronous protocols assume that the maximum processing and communication times can be predicted. Such constraints on processing and communication times are needed by real-time systems with very short deadlines. However, timing constraints can be unnecessarily restrictive and unrealistic. Operating systems such as Unix have unpredictable response times, and communication involves buffering of messages which can cause unpredictable delays.

Reliable ordered delivery protocols require consensus decisions for message ordering and membership decisions; unfortunately, the impossibility of reaching consensus in an asynchronous distributed system has been shown in [28]. Additional proofs are given in [27, 39] and results indicating what is achievable appear in [9]. Necessary and sufficient conditions for broadcast consensus protocols are given in [46].

Some protocols for reaching consensus in asynchronous systems are asymptotic in the sense that the probability of reaching consensus asymptotically approaches one as time increases. An example of an asymptotic atomic broadcast protocol can be found in [38].

Much of the difficulty in reaching consensus in an asynchronous system is caused by the difficulty of distinguishing between a failed processor and a slow processor. Many protocols use timeouts to detect failure; these failure detectors are termed "unreliable" because they may sometimes eliminate working processors. Chandra and Toueg investigate the properties of unreliable failure detectors in [16].

Despite the difficulty of designing communication protocols for asynchronous environments, the benefits of not assuming an upper bound on communication and processing time generally outweigh the disadvantages. The following protocols were all designed to operate in asynchronous environments.

2.1 TCP

The most widely used protocol for reliable ordered delivery today is the Internet TCP protocol. The TCP protocol provides reliable ordered delivery of packets between a pair of processors; it does not order messages arriving from different sites. A sliding window mechanism provides flow control and packet ordering and recovery. Each packet is acknowledged by the sender upon receipt allowing additional packets to be sent. Sequence numbers on the packets allow ordering and recognition of lost packets. A group multicast to N processes using TCP requires 2N packets: N transmissions of the packet (one for each destination) and N acknowledgments. Although multicast services are now available on the Internet, these services provide only unreliable delivery and do not provide for ordering of packets consistently across an entire multicast group.

2.2 Chang and Maxemchuk

The Chang and Maxemchuk protocol [17] provides a total order on messages by using a token-passing protocol. This protocol does not use the token to restrict access to the medium. Instead, all processors can broadcast messages at any time, and the processor in possession of the token broadcasts acknowledgments that determine the total order of messages. The token site delays acknowledgment of new messages until it has received all messages already acknowledged by the previous token holders. In a network subject to k or fewer failures, messages can be discarded after the token has visited k+1 sites. The acknowledgments in this protocol are additional broadcast messages that increase the message traffic on the network.

The Chang and Maxemchuk protocol does not provide flow control because the broadcasting of new messages is not restricted. It does provide a membership mechanism to recover from token loss, but the membership algorithm is not resilient to further failures during reconfiguration. The protocol will normally broadcast two messages for each message ordered in a lossless system if the token is transferred with each acknowledgment. This overhead can be reduced in high-load situations by grouping several acknowledgments into a single message or by passing the token via the acknowledgment.

2.3 Isis

The Isis distributed programming system [12, 13] has been used in a wide variety of applications to provide partial and total ordering on messages within groups of processes. Four types of broadcast messages are allowed: GBCAST (group broadcast), ABCAST (atomic total order broadcast), CBCAST (causal partial order broadcast), and BCAST (unordered broadcast).

Isis has been upgraded recently with new protocols to enhance its performance. The current version of Isis includes timestamp vectors in messages multicast within a group to preserve causality relationships. Each processor maintains a timestamp vector which has an entry for each member of the group. The entries indicate which messages have been delivered and sent by this processor. When sending a message, a processor increments the value in its position of the timestamp vector and then appends the timestamp vector to the outgoing message. These timestamp vectors are then used to determine causal ordering of messages within a group. Causality between groups is maintained by passing current versions of timestamp vectors of other groups as needed. This mechanism does not, however, ensure proper ordering of messages connected by causal chains through series of process groups [13].

The total order on messages within a group is determined using a protocol similar to that of Chang and Maxemchuk. A token is circulated around the group, and the current holder of the token imposes an order on concurrent messages in the partial order for the group. The token holder then sends a message indicating the results of its ordering decision to the other processors. ABCAST messages sent to multiple process groups are not guaranteed to be ordered the same since the total ordering is only within a process group.

The Isis protocol also provides in the ABCAST and CBCAST broadcast functions the ability for the sender to request a "stability" for the message. A processor delays delivery of a message until the requested number of processors have acknowledged receipt of the message. If a processor is unable to achieve the requested level of stability due to a membership change, the message is delivered and mechanisms to determine the known stability are provided.

Group membership changes are implemented using GBCAST which provides a system-wide total order; the ABCAST mechanism only provides message ordering within a group. A GBCAST message causes a temporary halt to message ordering and a flush of all pending messages to ensure system-wide ordering consistency for the message. Each new group membership is referred to as a view, and the current view is appended to the timestamp vector in each message.

In the interest of higher performance, the Isis protocols were designed for a network in which partitioning and rejoining of a processor with stable storage after failure are not allowed. These restrictions in the model are required for Isis to maintain consistency, because the ABCAST ordering mechanism allows a partitioned or failed processor to order and deliver messages in a different order than processors in the rest of the system. Such ordering inconsistencies can lead to alternate decisions and can become a problem if the components of the partition later remerge. A processor that rejoins the membership after having been removed is considered a new processor and is forced to obtain state information from the other processors. The Isis membership mechanisms are described in [51].

The Isis system includes a comprehensive suite of message ordering and process group membership services. A key concept introduced in the Isis protocol is virtual synchrony. Virtual synchrony ensures that, if two processors are members of the same two consecutive configurations, then they will deliver the same set of messages in the first configuration. This level of consistency, although adequate in a network without partitioning and remerging, must be extended and further restrictions applied when partitioning is considered. This topic is discussed further in Chapter 3 of this dissertation.

2.4 Trans and Total

The Trans and Total protocols [40, 42, 45, 47] provide partial and total orders on messages broadcast on a local-area network. The Trans protocol uses positive and negative acknowledgments piggybacked on messages to create the partial order. A processor uses transitivity of acknowledgments to reduce the number of acknowledgments in a message; the processor places, in its next message, positive acknowledgments for messages it has received if the processor has not already received an acknowledgment for the message from another processor. The processor includes negative acknowledgments for messages that it has failed to receive. The partial order of the messages is computed from the acknowledgments requiring a graph processing operation.

The Total protocol converts the partial order created by the Trans protocol into a total order and involves no further exchange of messages. The Total protocol is rare in that it is fault tolerant and can continue to order messages without detecting failures; most other protocols block on processor failure. Because the Total protocol is fault tolerant, its membership protocol can be built on top of the ordering algorithm. Failure of a processor is determined by the processor's failure to broadcast or failure to receive as determined by the messages in the total order; the removal of the processor requires no additional messages. Several alternative algorithms are provided for addition of processors.

2.5 Transis

Reliable ordered delivery of messages and membership are provided in Transis by the Lansis and Toto [6, 7, 25] protocols. The Lansis protocol is used to provide delivery of messages in a partial order and is derived from the Trans protocol. The primary difference between Lansis and Trans is in the acknowledgments. A processor executing the Lansis protocol waits to acknowledge messages until they can be delivered in causal order. The advantage of waiting to acknowledge a message is that the causal order is directly defined by the acknowledgments.

A processor executing the Toto protocol computes the total order of the messages using the underlying causal order generated within Lansis with exchange of additional messages for majority voting on the message order. Since Toto is not fault-tolerant, failure detection and reconfiguration are provided by mechanisms within the Lansis protocol. Failure detection is by timeout, and reconfiguration requires several rounds of message passing to reach consensus on the new membership. Messages can be broadcast during reconfiguration and the causal order can be constructed, but the Toto total ordering protocol is stopped.

2.6 Psync

The Psync protocol [49] provides a partial order on messages broadcast between processes participating in a group. Each processor maintains a context graph that determines the partial order. When a message is broadcast, its header contains information defining the messages that it follows in the context graph. The context graph information contained in the message headers is similar to the acknowledgments used in Lansis. A processor executing Psync sends retransmission requests as separate messages. Gaps in the partial order due to failures are handled by discarding messages that follow a missing message in the context graph.

The Psync protocol has been implemented as part of the x-kernel operating system [29, 30]. The responsibility for providing a total order and for consistency between groups is left to the application program. Several services such as membership have been added to the Psync protocol [43].

2.7 TPM

The token-passing multicast protocol (TPM) uses a token to provide reliable ordered multicast communication within process groups [50]. A processor can only broadcast a message if it is in possession of the token. Each message is given a sequence number derived from the token. TPM proceeds by first circulating the token to send a set of messages. The token is then used to determine which messages of the set processors are missing. Missed messages are retransmitted until the set of messages can be delivered. During this time, processors are allowed to start sending a new set of messages which will be considered for delivery after the current set.

TPM provides a mechanism for recovery from failures. If the network partitons, the component with the majority of the members of the group is the only component that is allowed to operate. Token regeneration is carried out by iteratively trying to pass the token to the members of the old token list until a processor accepts the token. Token regeneration is successful only if the new token list has a majority of the group members.

2.8 The Amoeba System

In Amoeba [32], messages are sent point-to-point to a central coordinator that assigns the message a sequence number and then broadcasts the message. A process acknowledges receipt of messages by placing the highest message sequence number received without gaps in its next message. The processes are organized into groups and messages are broadcast within a group. Each group has its own independent central site. The Amoeba system does not order messages for different groups with respect to each other.

In the Amoeba approach, each reliable broadcast requires a minumum of one point-to-point message and one broadcast message for each broadcast message. It has potential to reduce the storage requirements across the system as a whole since the central coordinator is the only site responsible for keeping copies of messages until the broadcast becomes stable.

2.9 Summary

Although many reliable ordered delivery protocols have been developed, many of these protocols have high overhead. In TCP each point-to-point conversation has its own sliding window and acknowledgments. All of the causal order protocols have to maintain the partial order information. These overheads, although reasonable in a local-area network, become a dominant factor when several local-area networks are participating in the protocol.

A problem that has been identified by the database community for distributed applications is partitioning. The only prior system that has begun to address the problem of partitioning and remerging of the network is the Transis system. The other protocols either assume that partitioning does not occur or only allow a primary component to continue and do not allow remerging unless the processors rejoin without stable storage intact.

Message throughput of a reliable delivery protocol is seriously degraded as retransmissions increase; retransmissions require bandwidth and processing time. Despite this, many protocols allow unrestricted access to the communication medium, and use external flow control mechanisms. These external mechanisms depend on heuristics to determine the current traffic load and are often inaccurate, leading to input buffer overflow, message loss and throughput degradation.

The Totem system has been designed specifically to address the above mentioned problems.

Chapter 3

The Model and Services

3.1 Environment

We model a network as a finite number of broadcast domains that are interconnected by gateways. A *broadcast domain* consists of a finite number of processors that communicate by broadcasting messages; each processor has a unique identifier. The broadcast domain has the following characteristics.

A broadcast message is received immediately (without excessive delay) or not at all by each processor or gateway in the broadcast domain, i.e. it may be received by only a subset of the processors or gateways. A processor or gateway receives all of its own broadcast messages. Messages can be rebroadcast to achieve reliable delivery.

Imposed on the broadcast domain is a logical token-passing ring. Each ring has a *representative*, chosen deterministically from the membership when the ring is formed, that initiates the token for the ring, and an identifier that consists of a ring sequence number and the identifier of the representative. To ensure that ring sequence numbers and hence ring identifiers are unique, each processor stores its ring sequence number in stable storage.

The gateways interconnecting the broadcast domains forward messages between rings and perform topology maintenance functions across the entire network. Other than these functions, a gateway behaves exactly the same as a processor. Each processor or gateway has stable storage, and communication is bi-directional. Messages are timestamped when they are first broadcast and messages are ordered by timestamp to ensure consistent ordering across the entire network. To ensure that timestamps on messages generated by a processor after the processor failed are larger than the timestamp on any message previously generated by the processor, the current timestamp is stored periodically in stable storage.

We use the term *configuration* to define a particular membership or network topology view provided to the application. The *membership* of a single-ring protocol configuration is a set of processor identifiers. In the single-ring protocol, a minimum configuration consists of the processor itself. A *regular* configuration has the same membership and identifier as its corresponding ring. The *transitional* configuration consists of processors that are transitioning from the same old ring to the new ring. A transitional configuration also has an identifier that consist of a "ring" sequence number and the identifier of a representative.

The network *topology* of the multiple-ring protocol consists of a set of single-ring configuration identifiers. The minimum configuration in the multiple-ring protocol consists of a single ring identifier.

We distinguish between receipt and delivery of a message. A message is *received* from the next lower layer in the protocol hierarchy and a message is *delivered* to the next higher layer. Delivery may be delayed to achieve ordering properties requested by a message. We use the term *originate* to refer to the generation of a message by the application when it is broadcast the first time.

Two types of messages are delivered to the application. *Regular* messages are originated by the application for delivery to the application. A *configuration change* message contains notification of a membership change reported by the single-ring membership algorithm or a topology change reported by the multiple-ring membership algorithm. The configuration change message terminates one configuration and initiates another.

We assume that the network, processors and gateways are asynchronous and provide unreliable failure detectors using timeouts [16]. We implement flow control within the protocol so that messages are not dropped within a processor or gateway.

3.2 Fault Model

Processors can incur fail stop [52], timing, or omission faults [23]. A processor that is excessively slow, or that fails to receive a message an excessive number of times, can be regarded as having failed. A processor's identifier does not change when the processor fails and restarts. A repaired processor may have retained all or part of its data in stable storage. There are no malicious faults.

The network may become partitioned so that processors in one component of the partitioned network are unable to communicate with processors in another component. Communication among separated components can subsequently be reestablished. Messages can be dropped by the communication medium, and corrupted messages are detected.

No distinction is made in the single-ring protocol between loss of all copies of the token and processor failure or network partitioning because a failed or disconnected processor cannot forward the token to the next processor on the ring. Thus, the consequence of processor failure or network partitioning is loss of all copies of the token. Loss of all copies of the token results in invocation of the single-ring membership algorithm and formation of a new token-passing ring.

3.3 Consensus

We define *consensus* with respect to a particular configuration C as follows:

- If processor p reaches a decision value x in configuration C, then every processor in C reaches decision value x or fails.
- If processor p reaches a decision value x in configuration C, then p does not reach a different decision value y.
- The decision value reached by p is not pre-determined.

The main difference between this definition of consensus and the traditional definition is that traditional consensus is not tied to a particular configuration but instead applies across the entire system over all time [28].

3.4 Membership Services

The following services are provided by the single-ring membership algorithm and by the multiple-ring membership algorithm.

- **Delivery of Configuration Change Messages.** Each configuration change is signalled by delivery of a Configuration Change message by the membership algorithm. The Configuration Change message contains the configuration identifier and the membership of the new configuration.

- **Uniqueness of Configurations.** Each configuration identifier is unique; moreover, at any time a processor is a member of at most one configuration.

- **Termination.** If a configuration ceases to exist for any reason, such as processor failure or network partitioning, then every processor of that configuration either installs a new configuration, or fails before doing so.

- **Configuration Change Consistency.** Processors that are members of the same configuration C_1 deliver the same Initiate Configuration C_1 message to begin the configuration. Furthermore, if two processors install a configuration C_2 directly after C_1, then the processors deliver the same Configuration Change message to terminate C_1 and initiate C_2.

Within the multiple-ring membership algorithm a configuration is a topology and the Configuration Change message is replaced by the Topology Change message. The Initiate Configuration C_2 message and the Terminate Configuration C_1 message are replaced by the Initiate Topology C_2 message and the Terminate Topology C_1 message, respectively.

3.5 Reliable Ordered Delivery Services

The service of reliable totally ordered message delivery is provided by both the single-ring protocol for messages in the broadcast domain and the multiple-ring protocol for messages network-wide.

We define a causal order that is similar to Lamport's definition [36], but we define the causal order in terms of messages rather than events and with respect to a particular configuration rather than across all configurations. This allows rejoining of failed processors and remerging of partitioned networks without requiring all messages from all components of a partition to be delivered. For this definition, we split the Configuration Change message into a Terminate Configuration C_1 message and an Initiate Configuration C_2 message. The Initiate Configuration C_2 message that starts configuration C_2 lists the membership of configuration C_2 and is the same for every processor p in C_2. A processor that transitions directly from configuration C_1 to configuration C_2 delivers a Terminate Configuration C_1 message and an Initiate Configuration C_2 message together as a Configuration Change message. A processor always delivers a Terminate Configuration message and an Initiate Configuration message together; one is never delivered without the other. There is, however, no implied causal relationship between the Initiate Configuration message and the Terminate Configuration message for different configurations.

Causal Order for a Configuration

For a given configuration C and for all processors p that are members of C, the causal order for C is the reflexive transitive closure of the "precedes" relation defined as follows:

- The Initiate Configuration C message delivered by p precedes every message originated by p in C.
- For each message m_1 delivered by p in C and each message m_2 originated in C by p, if m_1 is delivered by p before message m_2 is originated, then m_1 precedes m_2.
- For each message m_1 originated in C by p and each message m_2 originated in C by p, if m_1 is originated by p before m_2, then m_1 precedes m_2.
- Each message delivered in C by p precedes the Terminate Configuration message delivered by p to terminate C.

This definition of causal order allows processors to deliver messages after the network partitions by limiting the causal relationships to the configuration in which the message is originated. This allows processors in different components of the partitioned network to remerge and deliver messages without having to deliver the messages delivered in the other component. Past systems have defined the causal order to be across all messages and all configurations [13].

The processors in a configuration do not necessarily deliver the same last few messages in a configuration. Partitioning of the network can result in different sets of messages being delivered in different components of the network and therefore in different configurations. Each message is delivered according to its timestamp so that the relative order of any two messages can be established deterministically by processors that deliver both messages. We define the message delivery order within a configuration and across the entire network in the following manner:

Delivery Order for Configuration C

The reflexive transitive closure of the "precedes" relation defined on the union of the sets of regular messages delivered in C by all processors p in C, as follows:

- Message m_1 precedes message m_2 if processor p delivers m_1 in C before p delivers m_2 in C.

We prove in Sections 4.5 and 5.5 that the Delivery Order for Configuration C is a total order. Note that some processors in configuration C may not deliver all messages of the Delivery Order for Configuration C.

Global Delivery Order

The reflexive transitive closure of the union of the Delivery Orders for all Configurations and of the "precedes" relation defined on the set of Configuration Change messages and regular messages as follows:

- Message m_1 precedes message m_2 if a processor p delivers m_1 before p delivers m_2.

We prove in Sections 4.5 and 5.5 that the Global Delivery Order is a total order. In the past, the protocol efficiency decreased as the level of ordering increased. This was a result of the perceived need to construct the total order from the causal order. The objective of the Totem protocol is to deliver totally ordered messages efficiently and with less overhead than can be achieved by other protocols for partial ordering on a local-area network. The message ordering services provided by the Totem single-ring and multiple-ring protocols, defined below, are for all configurations C and all processors $p \in C$.

- **Reliable Delivery for Configuration** C
  - Each regular message m has a unique message identifier.
  - If a processor p delivers message m, then p delivers m only once.
  - A processor p delivers its own messages unless it fails.
  - If processor p delivers two different messages, then p does not deliver them simultaneously.
  - A processor p delivers all of the messages originated in its current configuration C unless a configuration change occurs.
  - If processors p and q are both members of consecutive configurations C_1 and C_2, then p and q deliver the same set of messages in C_1 before delivering the Configuration Change message that terminates C_1 and initiates C_2.
- **Delivery in Causal Order for Configuration** C
  - Reliable delivery for Configuration C.
  - If processor p delivers both messages m_1 and m_2, and m_1 precedes m_2 in the Lamport causal order, then p delivers m_1 before p delivers m_2.
- **Delivery in Agreed Order for Configuration** C
  - Delivery in causal order for Configuration C.
  - If processor p delivers message m_2 in configuration C and m_1 is any message that precedes m_2 in the Delivery Order for Configuration C, then p delivers m_1 in C before p delivers m_2.

- **Delivery in Safe Order for Configuration C**
  - Delivery in agreed order for Configuration C.
  - If processor p delivers message m in configuration C and the originator of m requested safe delivery, then p has determined that each processor in C has received m and will deliver m or will fail.

- **Extended Virtual Synchrony**
  - Delivery in agreed or safe order as requested by the originator of the message.
  - If processor p delivers messages m_1 and m_2, and m_1 precedes m_2 in the Global Delivery Order, then p delivers m_1 before p delivers m_2.

Reliable delivery defines which messages a processor must deliver and basic consistency constraints on that delivery. Agreed order goes further by defining delivery in total order. When a processor delivers a message in agreed order in a configuration, the processor has delivered all preceding messages in the total order and the processors in the configuration have reached consensus regarding the delivery order of this message.

When a processor delivers a message in safe order in a configuration, the processors in the configuration have reached consensus regarding the delivery order of the message. Consensus is reached through acknowledgments from the other processors in the configuration that have received the message and all preceding messages. The algorithm is designed to guarantee that once a processor has acknowledged a message and its predecessors, the processor will deliver the message unless the processor fails. There is no requirement defining the configuration in which the processor delivers the message.

Extended virtual synchrony ensures that messages are delivered in a consistent order system-wide, even if processors fail and restart or the network partitions and remerges. In contrast, virtual synchrony only constrains delivery of messages in a single component of the network, even if processors in other components have received the messages. Protocols that use virtual synchrony as their consistency constraint are forced to ensure that at most one component continues to operate after partitioning occurs. Virtual synchrony allows the components that are halted to deliver messages inconsistently before halting. Extended virtual synchrony has been defined to provide consistent message delivery despite partitioning and remerging. The Totem protocol uses a Configuration Change message or a Topology Change message to notify the application of the membership of the configuration or topology within which delivery is guaranteed before delivering a message as safe.

Some distributed applications require that, if a partition occurs, at most one of the resulting components is allowed to continue to deliver messages. The component allowed to continue is referred to as the primary component. We show in [44] that a primary partition system can be built on top of a protocol that provides extended virtual synchrony.

Chapter 4

The Single-Ring Protocol

The single-ring protocol provides membership and agreed and safe delivery of messages within a broadcast domain. The membership algorithm gathers the processors into a ring and begins circulation of the token. A processor must be in possession of the token to broadcast a message. Each message header contains a sequence number derived from a field in the token; there is a single sequence of monotonically increasing sequence numbers for the ring. Delivery in sequence number order is agreed delivery. Safe delivery uses an additional field in the token to determine when all processors on the ring have received a message.

4.1 The Total Ordering Algorithm

First we describe the algorithm with the assumptions that the token is never lost, that processor failures do not occur, and that the ring does not become partitioned; however, messages may be lost. In Section 4.2 we describe the membership algorithm which handles token loss, processor failure and restart, and partitioning and remerging of the ring.

The Data Structures

Regular Message

Each regular message contains the following fields:

- *sender_id*: The identifier of the processor originating the message.
- *ring_id*: The identifier of the ring on which the message was originated, consisting of the representative's identifier and a ring sequence number.
- *seq*: A message sequence number.
- *conf_id*: 0.
- *contents*: The contents of the message.

The *ring_id*, *seq* and *conf_id* fields constitute the identifier of the message.

Regular Token

To broadcast a message on the ring, a processor must hold the token. The token contains the following fields:

- *type*: Regular.
- *ring_id*: The identifier of the ring on which the token is circulating, consisting of the representative's identifier and a ring sequence number.
- *token_seq*: A sequence number which allows recognition of redundant copies of the token.
- *seq*: The largest sequence number of any message that has been broadcast on the ring, *i.e.* a high-water mark.
- *aru*: A sequence number (all-received-up-to) used to determine which messages processors on the ring have received, *i.e.* a low-water mark. The *aru* controls the discarding of messages that have been received by all processors on the ring.
- *aru_id*: The identifier of the processor that set the *aru* to a value less than the *seq*. The *aru_id* is used to choose a processor to blame for failure to receive.
- *rtr*: A retransmission request list, containing one or more retransmission requests.

Local Variables

Each processor maintains the following local variables:

- *my_aru*: The sequence number of a message such that the processor has received all messages with sequence numbers less than or equal to this sequence number.

- *my_token_seq*: The value of the *token_seq* when the processor forwarded the token last.

- *my_high_delivered*: The sequence number in the most recently delivered message.

The *my_aru* and *my_token_seq* are initialized to zero by the processor in the membership algorithm during the formation of the ring. *My_aru* is updated by the processor as it receives messages. *My_token_seq* is updated by the processor as it receives tokens. *My_high_delivered* is initialized to zero on installation of a new token ring and is updated by the processor as it delivers messages.

The Algorithm

On reception of the token, a processor completes processing of all the messages in its input buffer, so that the processor has an empty input buffer at the start of the next token rotation. After emptying the messages from its input buffer, the processor broadcasts requested retransmissions and new messages, updates the token, and transmits the token to the next processor on the ring. For each new message that it broadcasts, a processor increments the *seq* field of the token and sets the sequence number of the message to the value in the *seq* field.

Each time a processor receives the token, the processor compares the *aru* field of the token with *my_aru* and, if *my_aru* is smaller, replaces *aru* with *my_aru* and sets *aru_id* to its identifier. If the processor receives the token and the *aru_id* field of the token equals its identifier, it then sets *aru* to *my_aru*. If *seq* and *aru* are equal, it increments *aru* and *my_aru* in step with *seq* and sets the *aru_id* field to negative one (an invalid processor identifier).

If the *seq* field of the token is higher than *my_aru*, there are messages that this processor has not received so the processor sets or augments the *rtr* field. If the processor has received messages that appear in the *rtr* field then, for each such message, it retransmits that message before broadcasting new messages. When it retransmits a message, the processor removes the sequence number of that message from the *rtr* field. The pseudocode executed by a processor on receipt of a token is shown in Figure 4.1.

If a processor has received message m and has received and delivered every message with sequence number less than that of m and if the originator of m requested agreed delivery, then the processor delivers m in agreed order. If, in addition, the processor has forwarded the token with an *aru* greater than or equal to the sequence number of m on two successive rotations and if the originator of m requested safe delivery, then m can be delivered by the processor in safe order. When a message becomes safe, it no longer needs to be retained for future retransmission. The pseudocode executed by a processor on receipt of a regular message is given in Figure 4.2.

The total ordering algorithm is unable to continue when the token is lost; token retransmission has been added to reduce the probability that the token is lost. Each time a processor forwards the token, it sets a Token Retransmission timeout. A processor cancels the Token Retransmission timeout if it receives a regular message or the token. Receipt of a regular message indicates that the token is not lost. On a Token Retransmission timeout, the processor retransmits the token to the next processor on the ring and then resets the Token Retransmission timeout.

The *token_seq* field provides recognition of redundant tokens. A processor accepts the token only if the *token_seq* field of the token is greater than or equal to *my_token_seq*; otherwise, the token is discarded as redundant. If the token is accepted, the processor increments *token_seq* and sets *my_token_seq* to the new value of *token_seq*. Token retransmission increases the probability that the token will be received at the next site and incurs minimal overhead. The membership algorithm described in Section 4.2 handles the loss of all copies of the token.

```
Regular token received:
    if token.ring_id ≠ my_ring_id or token.token_seq < my_token_seq then
        discard token
    else
        cancel Token Retransmission timeout
        determine how many msgs I'm allowed_to_broadcast
            by flow control
        update retransmission requests
        broadcast requested retransmissions
        subtract retransmissions from allowed_to_broadcast
        for allowed_to_broadcast iterations do
            get message from new_message_queue
            increment token.seq
            set message header fields and broadcast message
        endfor
        update my_aru
        if my_aru < token.aru or my_id = token.aru_id or
                token.aru_id = invalid then
            token.aru := my_aru
            if token.aru = token.seq then
                token.aru_id = invalid
            else token.aru_id = my_id
            endif
        endif
        Determine failure to receive (detailed in Figure 4.4)
        update token rtr and flow control fields
        last_aru_seen := token.aru
        increment token.token_seq
        my_token_seq := token.token_seq
        forward token
        set Token Retransmission timeout
        deliver messages that satisfy their delivery criteria
    endif
```

Figure 4.1: Algorithm executed by a processor or a gateway on receipt of a token.

```
Regular message received:
    add message to receive_message_queue
    update retransmission request list
    update my_aru
    deliver messages that satisfy their delivery criteria
```

Figure 4.2: Algorithm executed by a processor or a gateway on receipt of a regular message.

4.2 The Membership Algorithm

The Totem single-ring ordering algorithm is optimized for high performance under failure-free conditions, but depends on a membership algorithm to handle token loss, processor failure, and network partitioning. The membership algorithm detects these events and constructs a new ring on which the Totem single-ring ordering algorithm can resume operation. The objective of the membership algorithm is to reach consensus on the membership of the new ring, to generate a new token, and to recover messages that had not been delivered by some of the processors when the failure occurred.

Termination of the membership algorithm is achieved by only allowing the set of processors considered for membership and the set of processors regarded as failed to increase monotonically, by bounding with timeouts the time that a processor spends in each of the states, and by forcing an additional failure rather than repeating a proposed membership.

The States of the Membership Algorithm

The membership algorithm is described by four states illustrated in Figure 4.3, we list the states here before defining the algorithm below.

- **Operational State.** In the Operational state messages are broadcast and delivered in agreed or safe order, as described in Section 4.1.

- **Gather State.** In the Gather state processors exchange Join messages with one another to reach consensus on a ring membership. Each Join message contains a set of processors being considered for membership in

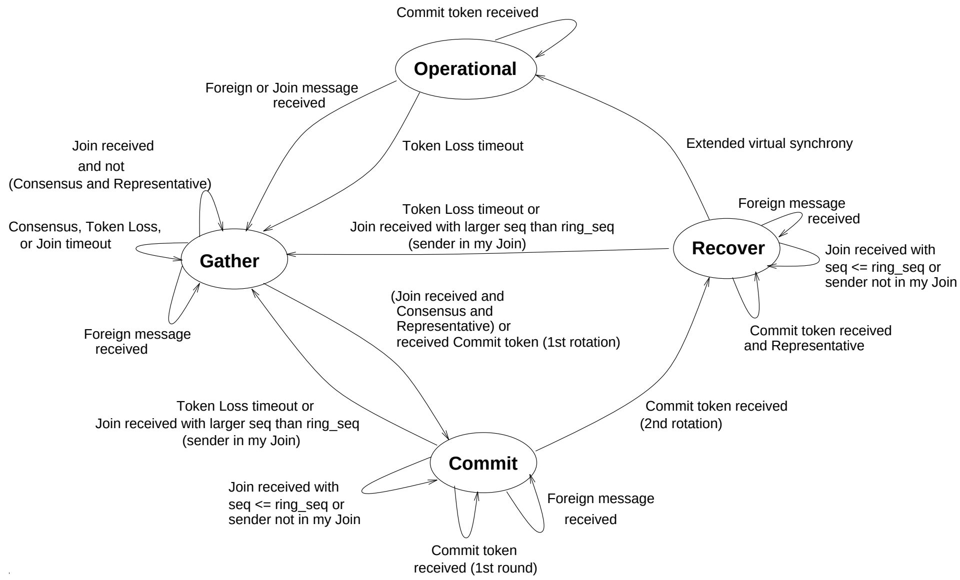


Figure 4.3: The finite state machine for the membership algorithm.

the new ring by the processor broadcasting the Join message and also a set of processors regarded as failed.

- **Commit State.** On reaching consensus, the representative constructs an identifier for the ring and launches a Commit token. Circulation of the Commit token confirms that all members of the ring agree on the membership, and collects information needed to determine correct handling of the messages from the old ring that had not been delivered by some of the processors when the membership algorithm started.

- **Recover State.** In the Recover state processors use the new ring to retransmit messages from their old rings.

The Events of the Membership Algorithm

There are seven membership events, namely:

- Receiving a *foreign* message broadcast by a processor that is not a member of the ring. A foreign message activates the membership algorithm in the processor that receives it.

- Receiving a Join message, which informs the receiver of the sender's proposed membership and may cause the receiver to enlarge its *my_proc_set* or *my_fail_set*.

- Receiving a Commit token. On the first reception of the Commit token a member of the proposed new ring updates the Commit token. On the second reception it obtains the updated information that the other members have supplied.

- Token Loss timeout. This timeout indicates that a processor did not receive the token or regular messages from other processors on the ring in the required amount of time and activates the membership algorithm.

- Join timeout. This timeout is used to determine the interval after which a Join message is rebroadcast in the Gather or Commit states.

- Consensus timeout. This timeout indicates that a processor participating in the formation of a new ring failed to reach consensus in the required amount of time.

- Recognizing failure to receive. If the *aru* has not advanced in several rotations of the token, a processor determines that the processor that set this *aru* has repeatedly failed to receive a message.

The Data Structures

Local Variables

Each processor maintains the following local variables:

- *my_ring_id*: The ring identifier in the most recent Commit token that the processor has accepted.

- *my_old_ring_id*: The ring identifier of the last ring this processor installed.

- *my_rotation_count*: The number of times that the processor has forwarded the token.

- *my_last_aru*: The value of the *aru* when the processor last forwarded the token.
- *my_aru_count*: The number of times that the processor has received the token with an unchanged *aru* where the aru is not equal to *seq*.
- *my_memb*: The set of identifiers of processors on the current ring.
- *my_trans_memb*: The set of identifiers of processors that are transitioning from the processor's old ring to its new ring.
- *my_new_memb*: The set of identifiers of processors on the new ring.
- *my_proc_set*: The set of identifiers of processors that are under consideration for the membership of a new ring.
- *my_fail_set*: The set of identifiers of processors that the processor has determined to have failed during execution of the membership algorithm (a subset of *my_proc_set*).
- *consensus*: A boolean array indexed by processors and indicating whether each processor is committed to the processor's *my_proc_set* and *my_fail_set*.
- *my_deliver_memb*: The set of identifiers of processors whose messages the processor must deliver in the transitional configuration.
- *my_received_flg*: A flag that indicates whether the processor has received all messages from processors in *my_deliver_memb*.
- *my_last*: A list of Join messages that have been received from other processors.
- *discard_regular_token*: A boolean to decide whether the regular token should be discarded.
- *high_ring_delivered*: The largest old ring sequence number of any message delivered on the old ring by the processors in *my_deliver_memb*.

- *low_ring_aru*: The lowest aru value of any of the processors in *my_deliver_memb*.

When a processor first comes up, it initializes the representative identifier in *my_ring_id* to its processor identifier and the sequence number to the value contained in stable storage. When a processor installs a new ring, it initializes *my_memb* and *my_proc_set* to the set of identifiers of processors on the new ring, *my_fail_set* and *my_last* to the empty set, *my_old_ring_id* to *my_ring_id*, and *my_received_flg* and *discard_regular_token* to false. A processor initializes *my_last_aru*, *my_aru_count*, and *my_rotation_count* to zero in the membership algorithm during the formation of the ring, and updates these variables each time it forwards the token.

The Join Message

Each time a processor in the Gather state modifies *my_proc_set* or *my_fail_set* it broadcasts a special type of message, the *Join* message. Join messages differ from regular messages in that a processor may broadcast a Join message without holding the token; moreover, Join messages are not retransmitted or delivered to the next higher layer. A Join message contains the following fields:

- *type*: Join.

- *sender_id*: The processor identifier of the sender.

- *ring_seq*: The largest sequence number of a *ring_id* known to the sender.

- *proc_set*: The set of identifiers of processors that are under consideration for membership in a new ring.

- *fail_set*: The set of identifiers of processors that the sender has determined to have failed during execution of the membership algorithm (a subset of *proc_set*).

- *rotation_count*: The number of times the sender has forwarded the token since reaching consensus.

The processor broadcasting the Join message sets the *proc_set*, *fail_set*, and *rotation_count* fields of the Join message to the values of its local variables *my_proc_set*, *my_fail_set*, and *my_rotation_count*, respectively. It also sets the *ring_seq* field of the Join message to the ring sequence number in *my_ring_id*.

Each time a processor broadcasts a Join message, it is trying to achieve consensus on the *proc_set* and *fail_set* in the Join message. The *ring_seq* field allows the receiver of a Join message to determine if the sender has abandoned a past round of consensus and is now attempting to form a new membership. It is also used to create unique transitional ring identifiers.

The Configuration Change Message

The membership algorithm uses another special type of message, the *Configuration Change* message, which contains the following fields:

- *ring_id*: The identifier of the regular configuration that this message initiates if the message initiates a regular configuration or the identifier of the preceding regular configuration if this message initiates a transitional configuration.

- *seq*: 0 if this message initiates a regular configuration or the largest sequence number of a message delivered in the preceding regular configuration if this message initiates a transitional configuration.

- *conf_id*: The identifier of the old transitional configuration from which the processor is transitioning if this message initiates a regular configuration or the identifier of the transitional configuration to which the processor is transitioning if this message initiates a transitional configuration.

- *memb*: The membership of the configuration that this message initiates.

The *ring_id*, *seq* and *conf_id* fields comprise the identifier of the message. A Configuration Change message may describe a change from an old configuration to a transitional configuration or from a transitional configuration to a new configuration. Configuration Change messages differ from regular messages in that they are generated locally at each processor and are delivered directly to the application without being broadcast. They are used to inform the next higher layer when membership changes occur.

The Commit Token

Each new ring is initiated by one of its members, the representative, a processor chosen deterministically from the members of the ring. The representative generates a Commit token that differs from the regular token in that its *type* field is set to *Commit* and it contains the following fields in place of the *rtr* field:

- *memb_list*: A list containing a processor identifier, *my_old_ring_id*, *old_ring_my_aru*, *my_received_flg*, and *my_high_delivered* fields for each member of the new ring. The *my_received_flg* field indicates whether the member has received all messages from the processors in its *my_deliver_memb* in a previous failed pass through the Recover state in which the new ring was not installed. The *my_high_delivered* field is the largest sequence number of a message that the processor has delivered on the old ring. This list is ordered according to the positions of the members on the new ring.
- *memb_index*: The index of the processor in the *memb_list* that last forwarded the Commit token.

On the first rotation of the Commit token around the new ring, each processor sets its *my_old_ring_id*, *old_ring_my_aru*, *my_received_flg*, and *my_high_delivered* fields in the token. It also updates *memb_index*. The remaining fields are set by the representative when it creates the Commit token.

The Algorithm

A processor that starts or restarts first forms and installs a singleton ring containing only itself; it then broadcasts a Join message containing the value of *my_ring_id.seq* from its stable storage and proceeds to the Gather state.

The Operational State

The total ordering algorithm described in Section 4.1 is executed by a processor while in the Operational state. When the Token Loss timeout expires or when a Join or foreign message is received by a processor on the ring, the algorithm for the formation of a new ring is invoked in the Operational state. Join and

```
Regular token received
    in addition to the actions listed in Figure 4.1
    increment my_rotation_count and
    Determine failure to receive (described below)
    if token.aru = last_aru_seen and token.aru_id ≠ invalid then
        increment my_aru_count
    else
        my_aru_count := 0
    endif
    if my_aru_count > fail_rcv_const and token.aru_id ≠ my_id then
        add token.aru_id to my_fail_set
        Call Shift_to_Gather
    endif
Token Loss timeout expired
    discard_regular_token := true
    call Shift_to_Gather
Foreign message from processor q received:
    add message.sender_id to my_proc_set
    call Shift_to_Gather
Join message from processor q received:
    same as in Gather state (Figure 4.6) except always call Shift_to_Gather
    before returning regardless of content of Join message
Commit token received:
    discard the Commit token
```

Figure 4.4: Algorithm executed by a processor or a gateway on occurrence of a membership event in the Operational state.

Consensus timeouts cannot occur in the Operational state. If a processor receives a Commit token, it discards that token. A description of the actions taken by a processor in the Operational state when a membership change event occurs is given below. The pseudocode executed by a processor or gateway in the Operational state is shown in Figure 4.4.

Token Loss Timeout

On expiration of the Token Loss timeout in the Operational state, a processor broadcasts a Join message, sets the Join and Consensus timeouts, and shifts to the Gather state. The pseudocode executed by a processor when it shifts to the Gather state is given in Figure 4.5.

Receiving a Foreign Message

If a processor receives a foreign message in the Operational state that is not a message retransmitted in the Recover state, it sets *my_proc_set* to the union of its current *my_proc_set* and the singleton set containing the identifier of the sender of the foreign message. It then shifts to the Gather state (Figure 4.5).

Receiving a Join Message

If a processor receives a Join message in the Operational state and if the receiver's identifier is in the Join message's *fail_set* or if the sender's identifier is in the receiver's *my_proc_set* and the Join message's *ring_seq* is less than the receiver's ring sequence number, then it ignores the Join message. Otherwise, the processor updates its *my_proc_set* and *my_fail_set* as in the Gather state described below and shifts to the Gather state (Figure 4.5).

Recognizing Failure to Receive

A processor buffers a message for retransmission until receipt of the message has been acknowledged by the other processors on the ring. If a processor repeatedly fails to receive a particular message, then the other processors will buffer that message and all subsequent messages until that message is received. A processor cannot be allowed to fail to receive messages indefinitely because that failure might impose excessive buffering requirements, and prevent other processors from delivering messages in safe order.

When its local variable *my_aru_count* reaches a predetermined constant, a processor determines that some other processor has failed, namely the processor whose identifier is in the *aru_id* field of the token. The processor then discards the token, updates *my_fail_set* to include the processor identifier in *aru_id* and shifts to the Gather state (Figure 4.5).

```
Shift_to_Gather
    broadcast Join message containing my_proc_set, my_fail_set,
        my_rotation_count, and seq := my_ring_id.seq
    cancel Token Loss timeout and Token Retransmission timeout
    reset Join and Consensus timeouts
    discard_regular_token := true
    for all q in my_proc_set do consensus[q] := false endfor
    consensus[my_id] := true
    for all messages in my_last do
        if message.proc_set = my_proc_set and
                message.fail_set = my_fail_set then
            consensus[q] := true
        endif
    endfor
    state := Gather
    call Try_to_Form
```

Figure 4.5: Algorithm executed by a processor or a gateway to shift to the Gather state.

The Gather State

The objective of the Gather state is to achieve a membership that is as large as possible, while ensuring that the membership algorithm terminates. A membership is a set of processor identifiers on which the processors agree and in which every processor can communicate with every other processor. The actions on receiving a regular token or a foreign message in the Gather state and on detecting failure to receive in the Gather state are similar to the actions in the Operational state. In the Gather state a processor collects information about operational processors and failed processors. This information is broadcast in Join messages. The pseudocode executed by a processor or gateway in the Gather state is shown in Figures 4.6 and 4.7.

Receiving a Join Message

When a processor receives a Join message in the Gather state, it updates its *my_proc_set* and *my_fail_set* as described below. If its *my_proc_set* and *my_fail_set* have changed, it abandons its previous consensus, broadcasts a new Join message containing the updated sets, and resets the Join and Consensus timeouts. The processor remains in the Gather state.

```
Regular token or regular message received:
    same as in Operational state
Foreign message from processor q received:
    if q not in my_proc_set then
        add message.sender_id to my_proc_set
        call Shift_to_Gather
    endif
Join message from processor q received:
    if my_proc_set = message.proc_set and
            my_fail_set = message.fail_set then
        consensus[q] := true
        call Try_to_Form
        return
    else if message.proc_set is a subset of my_proc_set and
            message.fail_set is a subset of my_fail_set then
        return
    else if q in my_fail_set then return
    else/* there is something in this Join message not in mine */
            add message.proc_set to my_proc_set
        if my_id in message.fail_set then
            add message.sender_id to my_fail_set
        else
            if q in my_memb then
                if q not in my_fail_set then add message.fail_set to my_fail_set
            else
                add message.fail_set − my_memb to my_fail_set
            endif
        endif
        add message to my_last
        call Shift_to_Gather
    endif
```

Figure 4.6: Algorithm executed by a processor or a gateway in the Gather state on receipt of a regular message, regular token or Join message.

```
Commit token received:
    if my_proc_set − my_fail_set = token.memb and
            token.seq > my_ring_id.seq then
        call Shift_to_Commit
    endif
Join timeout expired:
    broadcast Join message with my_proc_set, my_fail_set,
        my_rotation_count and seq = my_ring_id.seq
    set Join timeout
Consensus timeout expired:
    empty my_last
    if consensus not reached then
        for each processor q where consensus[q] ≠ true do
            add q to my_fail_set
        endfor
        call Shift_to_Gather
    else
        for all q do consensus[q] := false
        consensus[my_id] := true
        set Token Loss timeout
    endif
Token Loss timeout expired:
    if reached consensus on same membership second time then
        add processor with lowest my_rotation_count to my_fail_set
    else
        execute code for Consensus timeout expired in Gather state
    endif
    call Shift_to_Gather
```

Figure 4.7: Algorithm executed by a processor or a gateway in the Gather state on receipt of a Commit token, Join timeout, Consensus timeout or Token Loss timeout.

Updating the Membership on Reception of a Join Message

If a processor receives a Join message with a *proc_set* and *fail_set* identical to its *my_proc_set* and *my_fail_set*, respectively, the processor records the sender of the Join message as participating in the consensus on those sets.

If a processor receives a Join message such that the sender's identifier is in the receiver's *my_fail_set*, it ignores that Join message. This is appropriate because a processor never declares itself failed. If a processor receives a Join message such that the receiver's identifier is in the Join message's *fail_set*, the receiver updates both *my_proc_set* and *my_fail_set* to include the identifier of the sender of the Join message. This is appropriate because those two processors will not be able to reach consensus on a membership that excludes one of them.

If a processor receives a Join message such that (1) the receiver's identifier is not in the Join message's *fail_set*, (2) the sender's identifier is not in the receiver's *my_fail_set*, and (3) the Join message's *proc_set* or *fail_set* contains at least one identifier that is not in the receiver's *my_proc_set* or *my_fail_set*, respectively, then the receiver adds to its *my_proc_set* and *my_fail_set* the identifiers in the Join message's *proc_set* and *fail_set*, respectively, with the following exception. If the sender is not a member of the receiver's old ring, then the receiver does not add an identifier of a member of its own old ring to its *my_fail_set*. This exception is intended to bias the membership algorithm towards preserving existing rings by preventing an outsider from breaking up an existing ring.

The Join Timeout and Rebroadcasting Join Messages

Each time a processor broadcasts a Join message, it sets or resets the Join timeout. When the Join timeout expires, the processor rebroadcasts the Join message. The Join timeout is shorter than the Consensus timeout and is used to increase the probability that Join messages from all currently working processors are received during a single round of consensus.

Reaching Consensus

A processor has reached consensus when it has received Join messages with *proc_set* and *fail_set* equal to its *my_proc_set* and *my_fail_set*, respectively, from every processor in the difference of those sets, *i.e.* *my_proc_set* − *my_fail_set*. A processor is also considered to have reached consensus when it has received a Commit token with the same membership as *my_proc_set* − *my_fail_set*. The processors in that difference constitute the membership of the proposed new ring.

When a processor has reached consensus, it determines whether it is the representative of the proposed new ring. If it is not the representative and has not received the Commit token, the processor sets the Token Loss timeout, cancels the Consensus timeout and continues in the Gather state, waiting for the Commit token.

If it is the representative, the processor generates a Commit token. It determines the *ring_id* of the new ring, which is composed of the representative's identifier and a ring sequence number equal to four plus the largest ring sequence number in any of the Join messages used to reach consensus (the sequence number two less than that of the new ring is used to create a unique transitional ring identifier). It also determines the *memb_list* of the Commit token, which specifies the membership of the new ring and the order in which the token will circulate, with the representative placed first. In addition, the representative sets the *type* field of the Commit token to *Commit*, the *token_seq* field to 0, the *seq* field to 0, the *aru* field to 0, the *fcc* field to 0 and the *retrans_flg* field to false. It also sets the *my_old_ring_id*, old ring *my_aru*, *my_received_flg* and *my_high_delivered* fields in its entry of the *memb_list* field of the token and sets the *memb_index* field to 1. The representative cancels the Consensus timeout, sets the Token Loss timeout, transmits the Commit token, and shifts to the Commit state. The pseudocode executed by the representative on reaching consensus is given in Figure 4.8.

```
Try_to_Form
    if for all q in my_proc_set − my_fail_set, consensus[q] = true and
            my_id = smallest_id of my_proc_set − my_fail_set then
        token.ring_id.seq := maximum of my_ring_id.seq and Join ring_seqs + 4
        token_memb := my_proc_set − my_fail_set
        call Shift_to_Commit
    endif
```

Figure 4.8: Algorithm executed by a processor or a gateway on reaching consensus.

Receiving a Commit Token

On receiving a Commit token, there are several acceptance tests performed by a processor. The processor first ensures that the ring sequence number in the *ring_id* field is greater than the ring sequence number in *my_ring_id* and that the *token_seq* field is less than the cardinality of *my_proc_set - my_fail_set*. The processor next compares the proposed membership, given by the *memb_list* in the Commit token, with the difference of its *my_proc_set* and *my_fail_set*. If they differ, the processor discards the Commit token. If they agree, it extracts the *ring_id* for the new ring, sets the *my_old_ring_id*, old ring *my_aru*, *my_received_flg*, and *my_high_delivered* fields in its entry of the *memb_list* field of the token, and increments the *memb_index* field of the token. It also initializes its *my_rotation_count* to 1 and increments the *token_seq* field of the token and sets *my_token_seq* equal to *token_seq*. The processor then cancels the Consensus timeout, resets the Token Loss timeout, forwards the Commit token, and shifts to the Commit state. The pseudocode executed by a processor when it shifts to the Commit state is given in Figure 4.9.

The Consensus Timeout

If the Consensus timeout expires before a processor has reached consensus, it adds to *my_fail_set* all of the processors in *my_proc_set* from which it has not received a Join message with *proc_set* and *fail_set* equal to its own sets. It then shifts to the Gather state (Figure 4.5).

```
Shift_to_Commit
    update memb_list in Commit_token with my_old_ring_id, my_aru,
        my_received_flg, and my_high_delivered
    my_ring_id := Commit token ring_id
    my_rotation_count := 1
    forward Commit token
    empty my_last
    cancel Join and Consensus timeouts
    reset Token Loss timeout and Token Retransmission timeout
    state := Commit
```

Figure 4.9: Algorithm executed by a processor or a gateway to shift to the Commit state.

The Token Loss Timeout

When a processor enters the Gather state, it cancels the Token Loss timeout. On reaching consensus the processor sets the Token Loss timeout and awaits the Commit token. If the Token Loss timeout expires, it remains in the Gather state and tries to reach consensus again. If it then reaches consensus on the same *my_proc_set* and *my_fail_set* as it had previously, it adds one processor to its *my_fail_set* and broadcasts another Join message. The processor added to *my_fail_set* is the first processor on the token rotation path that forwarded the token the fewest times, as determined by the *my_rotation_count* fields of the Join messages. If this is the processor itself, it forms a singleton ring. The same mechanism applies if the processor returns to the Gather state on a Token Loss timeout from the Commit or Recover state and reaches consensus on the same *my_proc_set* and *my_fail_set* as it had previously.

If the Commit token subsequently reaches a processor that has already determined that the token is lost because its Token Loss timeout expired, the processor discards the token.

The Commit State

The objective of the Commit state is to establish that all members of the proposed new ring agree on the membership and to collect information needed for the recovery algorithm. In the Commit state regular tokens are discarded, for-

```
Regular token received:
    discard token
Regular message received:
    same as in Operational state
Foreign message received:
    discard message
Join message from processor q received:
    if q in my_new_memb and message.ring_seq ≥ my_ring_id.seq then
        execute code for receipt of Join message in Gather state
        call Shift_to_Gather
    endif
Commit token received:
    if token.seq = my_ring_id.seq then call Shift_to_Recover
Join timeout expired:
    same as in Gather state
Token Loss timeout expired:
    call Shift_to_Gather
```

Figure 4.10: Algorithm executed by a processor or a gateway in the Commit state.

eign messages are ignored, Token Loss and Join timeouts are handled as in the Gather state, and Consensus timeout and recognition of failure to receive do not occur. The pseudocode executed by a processor or gateway when it shifts to the Commit state is given in Figure 4.9. The pseudocode executed by a processor or gateway in the Commit state is shown in Figure 4.10.

Receiving a Commit Token

On receiving a Commit token during its second rotation on the proposed new ring, a processor obtains, for each processor on that ring, the *my_old_ring_id* and *old_ring_my_aru* of that processor's old ring. From this information the processor calculates *my_trans_memb*, consisting of the processors transitioning from its same old ring. The processor then writes *my_ring_id.seq* to stable storage and forwards the token. If some processor in *my_trans_memb* has its *my_received_flg* set to false, the processor shifts to the Recover state (Figure 4.11), sets *my_deliver_memb*, *low_ring_aru*, *my_received_flg* (to false), and *high_ring_delivered* fields, and then executes the recovery algorithm. If each of

```
Shift_to_Recover
    forward Commit token                 /* second time */
    write my_ring_id.seq to stable storage
    my_aru_count := 0
    increment my_rotation_count
    discard_regular_token := false
    my_new_memb := membership in Commit token
    my_trans_memb := members on old ring transitioning to new ring
    if for some processor in my_trans_memb my_received_flg = false then
        my_deliver_memb := my_trans_memb
        my_received_flg := false
        low_ring_aru := lowest aru for old ring for processors in my_deliver_memb
        high_ring_delivered := highest sequence number of message delivered
            for old ring by a processor in my_deliver_memb
        copy all messages from old ring with sequence number >
            low_ring_aru into retrans_message_queue
    endif
    my_aru := 0
    last_aru_seen := 0
    my_retrans_flg_count := 0
    reset Token Loss timeout and Token Retransmission timeout
    state := Recover
```

Figure 4.11: Algorithm executed by a processor or a gateway to shift to the Recover state.

the processors in *my_trans_memb* has its *my_received_flg* set to true, the processor likewise shifts to the Recover state, and executes the recovery algorithm, but in this case no messages for the old ring need to be retransmitted.

Receiving a Join Message

If a processor receives a Join message in the Commit state from a member of the proposed new ring and that Join message contains a *ring_seq* greater than the ring sequence number of the proposed new ring, the processor abandons its current consensus, updates *my_proc_set* and *my_fail_set* as in the Gather state described above and shifts to the Gather state (Figure 4.5).

This is necessary because some processor has determined that either the Commit token or the regular token for the new ring has been lost.

A processor discards a token with a ring sequence number less than or equal to its own ring sequence number. Such a token must be the token of an old or abandoned ring.

The Recover State

On receiving the Commit token after its second rotation, the representative of the new ring converts the Commit token into the regular token for the new ring, replacing the *memb_list* and *memb_index* fields by the *rtr* field. At this point the new ring is formed but not yet installed, and the recovery operation begins. The recovery algorithm is described in Section 4.3.

In the Recover state failure to receive is handled exactly as in the Operational state, and Join messages are handled exactly as in the Commit state. Foreign messages are ignored, and Join and Consensus timeouts do not occur. Expiration of the Token Loss timeout in the Recover state results in a processor's returning to the Gather state, where token loss is handled as though the processor were a member of the old ring in the Gather state.

4.3 The Recovery Algorithm

The objective of the recovery algorithm is to recover the messages that had not been delivered by some of the processors when the membership algorithm was invoked, and to enable the processors transitioning from the same old configuration to the same new configuration to deliver the same set of messages from the old configuration. The recovery algorithm also maintains message delivery guarantees to the application during recovery from failures. Maintenance of these guarantees is essential to applications such as fault-tolerant distributed databases. The pseudocode executed by a processor in the Recover state is given in Figures 4.12 and 4.13.

```
Regular token received:
    same as in Operational state (Figures 4.1 and 4.4) except get messages from
    retrans_message_queue instead of new_message_queue
    and before forwarding the token execute:
    if retrans_message_queue is not empty then
        if token.retrans_flg = false then token.retrans_flg := true
    else
        if token.retrans_flg = true and I set it then token.retrans_flg := false
    endif
    if token.retrans_flg = false then increment my_retrans_flg_count
    else my_retrans_flg_count := 0
    endif
    if my_retrans_flg_count = 2 then my_install_seq := token.seq
    if my_retrans_flg_count ≥ 2 and my_aru ≥ my_install_seq
            and my_received_flg = false then
        my_received_flg := true
        my_deliver_memb := my_trans_memb
    endif
    if my_retrans_flg_count ≥ 3 and token.aru ≥ my_install_seq
            on last two rotations then
        call Install_Ring
    endif
Regular message received:
    reset Token Loss timeout
    add message to receive_message_queue
    update my_aru
    if retransmitted message from my_old_ring_id then
        add to receive_message_queue for old ring
        remove message from retrans_message_queue for old ring
    endif
Foreign message from processor q received: discard message
Join message from processor q received:
    if q in my_new_memb
            and message.ring_seq ≥ my_ring_id.seq then
        execute code for receipt of Join message in Commit state
        execute code for Token Loss in Recover state
    endif
```

Figure 4.12: Algorithm executed by a processor or a gateway in the Recover state on receipt of a regular message, regular token, foreign message or Join message.

```
Commit token received:
    /* new representative, i.e. smallest processor id */
    convert Commit token to regular token
    if retrans_message_queue is not empty then
        token.retrans_flg := true
    else
        token.retrans_flg := false
    endif
    forward regular token
    reset Token Loss timeout
    increment my_rotation_count
Token Loss timeout expired:
    discard_regular_token := true
    discard all new messages received on the new ring
    empty retrans_message_queue
    determine current old ring aru (it may have increased)
    call Shift_to_Gather
```

Figure 4.13: Algorithm executed by a processor or a gateway in the Recover state on receipt of a Commit token or on a Token Loss timeout.

The Data Structures

The recovery algorithm uses the following data structures in addition to those already introduced.

Token Field

The recovery algorithm depends on the following field of the token:

- *retrans_flg*: A flag that is used to determine whether there are any additional old ring messages that must be rebroadcast on the new ring.

The *retrans_flg* field of the token is initialized to false by the representative of the new ring when it generates the Commit token.

Local Variables

The recovery algorithm also depends on the following local variables:

- *my_install_seq*: The largest new ring sequence number of any old ring message transmitted on the new ring. The value of this variable is determined locally, but has the same value for all processors that install the new ring.

- *my_retrans_count*: The number of successive token rotations on which the processor has received the token with *retrans_flg* false.

The Algorithm

A processor executing the recovery algorithm takes the following steps:

1. Exchange messages with the other processors that were members of the same old ring to ensure that they have the same set of messages broadcast on the old ring but not yet delivered.

2. Deliver to the application those messages that can be delivered on the old ring according to the requirements for agreed or safe ordering, including all messages with old ring sequence numbers less than or equal to *high_ring_delivered*.

3. Deliver the first Configuration Change message changing to the transitional configuration.

4. Deliver messages that could not be delivered in agreed or safe order on the old ring because delivery might violate the requirements for agreed or safe delivery, but that can be delivered in agreed or safe order in the smaller transitional configuration.

5. Deliver a second Configuration Change message, changing to the new configuration.

6. Shift to the Operational state.

Steps 2 through 6 involve no communication with other processors and are performed as one atomic action. The pseudocode executed by a processor to complete these steps is given in Figure 4.14. In the Operational state the processor broadcasts and delivers messages for the new ring.

Exchange of Messages from the Old Ring (Step 1)

To implement the first step of the recovery algorithm, each processor that is a member of the new ring determines the lowest *my_aru* of any processor from its

```
Install_Ring
    deliver messages deliverable on old ring
        (at least up through high_ring_delivered)
    deliver membership change for transitional configuration
    deliver remaining messages from processors in my_deliver_memb
        in transitional configuration
    deliver membership change for new ring
    my_memb := my_new_memb
    my_proc_set := my_memb
    my_old_ring_id := my_ring_id
    my_fail_set := empty set
    my_received_flg := false
    state := Operational
```

Figure 4.14: Algorithm executed by a processor or a gateway to install a new ring.

old ring that is also a member of the new ring. The processor then broadcasts on the new ring every message for the old ring that it has received and that has a sequence number greater than the lowest *my_aru*. This ensures that each processor receives as many messages as possible from the old ring.

Each message is broadcast with a new ring identifier and a new ring sequence number, and encapsulates the old ring message with its old ring identifier and old ring sequence number. The new ring sequence numbers are used to ensure that messages are received; the old ring sequence numbers are used to order messages as messages of the old ring. Messages from an old ring retransmitted on the new ring are not delivered to the application by any processor that was not a member of that old ring. No new messages are broadcast by a processor in the Recover state.

Completion of the message exchange is determined by the *retrans_flg* field in the token and by the local variable *my_install_seq*. The *retrans_flg* is initially set to false, and a processor changes *retrans_flg* from false to true if it has more old ring messages to retransmit when it forwards the token. A processor changes *retrans_flg* from true to false if it set *retrans_flg* to true and now has no further old ring messages to retransmit.

When a processor has received the token on two successive rotations with *retrans_flg* set to false, it knows that all of the old ring messages have been retransmitted on the new ring. The processor then sets *my_install_seq* to the value of the *seq* field in the token; thus, *my_install_seq* is the largest new ring sequence number of any old ring message transmitted on the new ring. When *my_aru* is at least equal to *my_install_seq*, the processor has received all of the messages of the old ring that have been broadcast on the new ring. If *my_received_flg* equals false, the processor then sets *my_received_flg* to true and *my_deliver_memb* to *my_trans_memb*. The processor now has the complete set of messages rebroadcast for the old ring by the processors in *my_trans_memb*.

If a processor has forwarded the token with *retrans_flg* set to false on two successive token rotations and with the *aru* at least equal to *my_install_seq* on the last of those rotations, the processor provides a guarantee to deliver messages with sequence numbers at most equal to *my_install_seq* that were originated by processors in *my_deliver_memb* unless it fails, by setting its *my_received_flg* to true.

When a processor has received the token with *retrans_flg* set to false on three successive token rotations and with the *aru* at least equal to *my_install_seq* on the last two of those rotations, it determines that all processors on the ring have the value for *my_install_seq* and have received all messages with sequence numbers up to and including *my_install_seq*. The processor then proceeds to the delivery of messages on the old ring without further message exchange.

Delivery of Messages on the Old Ring (Steps 2-3)

For each message, the processor must determine the appropriate membership in which to deliver the message. A processor can deliver a message in agreed order for the old ring if it is in sequence number order and all the messages with lower sequence numbers have been delivered. A processor can deliver a message in safe order for the old ring if it received the old ring token with the *aru* field at least equal to the sequence number in the message twice in succession or if some other processor already delivered the message on the old ring as indicated by *high_ring_delivered*.

The processor sorts the messages for the old ring that were broadcast on the new ring into the order of their sequence numbers on the old ring, and delivers messages in order until it encounters a gap in the message sequence numbers or a message requiring safe delivery with a sequence number greater than *high_ring_delivered*. The variable *high_ring_delivered* provides the information that some processor delivered that message as safe on the old ring and therefore that the message is deliverable on the old ring.

The processor then delivers the first Configuration Change message, which contains the identifier of the old configuration, the identifier of the transitional configuration, and the membership of the transitional configuration. The membership of the transitional configuration is *my_trans_memb*. The identifier of the transitional configuration has sequence number two less than the sequence number of *my_ring_id*, and the representative's identifier is chosen deterministically from *my_trans_memb*.

Delivery of Messages in the Transitional Configuration (Steps 4-6)

Following the first Configuration Change message, the processor delivers in order all remaining messages that were originated on the old ring by processors in *my_deliver_memb*. The processor then delivers a second Configuration Change message, which contains the identifier of the transitional configuration, the identifier of the new configuration (*my_ring_id*), and the membership of the new configuration (*my_new_memb*). The processor then shifts to the Operational state (Figure 4.14).

Note that some messages cannot be delivered on the old ring or even in the transitional configuration because delivery of those messages might violate agreed or safe order. Such messages follow a gap in the message sequence. For example, if processor p originates or delivers message m_1 before it originates m_2 and processor q received m_2 but did not receive m_1 in the message exchange, then processor q cannot deliver m_2 because causality would be violated. Here p is not in the same transitional configuration as q because, if p had been in the same transitional configuration, then q would have received all of the messages originated by p before or during the message exchange.

Note also that a processor delivers messages for the old ring before it broadcasts or delivers any new message for the new ring. The decision to shift to the Operational state and the set of old ring messages to be delivered is a local decision. Some processors may be in the Operational state broadcasting messages for the new ring, while others are still in the Recover state and will install the new ring if the token completes its next rotation. Note, however, that no safe message is delivered on the new ring before all processors on the new ring install that ring.

Failure of Recovery

If the recovery fails while the recovery algorithm is being executed (for example, because the token is lost), some processors may have installed the new ring while others have not. Prior to installation, a processor's old ring is the ring of which it was a member when it was last in the Operational state. Each processor must preserve its old ring identifier until it installs a new ring.

When a processor delivers a message in safe order in a transitional configuration, it must have received a guarantee from all of the other members of the configuration that they will deliver the message unless they fail. If the token is lost in the Recover state, some processors may not install the new ring. Each such processor will proceed in due course to install a different new ring with a corresponding transitional configuration. It must deliver the message in that transitional configuration in order to honor the guarantee.

Thus, if a processor has set its *my_received_flg* in the Commit token to true, but the token is lost before this processor delivers those messages and installs the new ring, then another processor in the transitional configuration, relying on the guarantee, may have delivered messages from the old ring in safe order and installed the new ring. Consequently, if a processor finds the *my_received_flg* in the Commit token set to true for every processor in *my_trans_memb*, it must retain the old ring messages originated by members of *my_deliver_memb* and deliver them in the transitional configuration for the new ring that it actually installs. Note that *my_trans_memb* can only decrease on successive passes through the Recover state before a new ring is installed.

Examples

Consider the simple example shown in Figure 4.15. Here a ring containing processors p, q, r, s and t undergoes a partition in which p becomes isolated

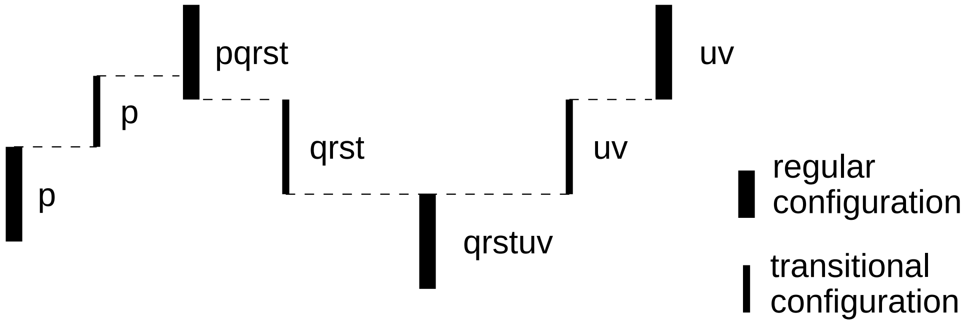


Figure 4.15: Regular and transitional configurations. The vertical lines represent the total orders of messages that have been delivered in the indicated configuration, and the dashed horizontal lines represent Configuration Change messages.

while q, r, s and t merge into a new ring with u and v. Processors q, r, s and t deliver two Configuration Change messages, one to switch from the regular configuration $\{p, q, r, s, t\}$ to the transitional configuration $\{q, r, s, t\}$ and one to switch from the transitional configuration $\{q, r, s, t\}$ to the regular configuration $\{q, r, s, t, u, v\}$. It may not be possible for processors q, r, s and t to deliver all messages originated in the regular configuration $\{p, q, r, s, t\}$, since some of these messages from p may not have been received before p became isolated; however, it can be guaranteed that in the transitional configuration $\{q, r, s, t\}$ all messages originated by a processor of that configuration have been delivered. Similarly, it may not be possible to deliver a message safe in the regular configuration $\{p, q, r, s, t\}$ because no information is available as to whether p had received that message before it became isolated, but it is possible to deliver the message safe in the transitional configuration $\{q, r, s, t\}$. The first Configuration Change message separates the messages that are delivered in the old configuration $\{p, q, r, s, t\}$ from the messages that are delivered in the reduced transitional configuration $\{q, r, s, t\}$.

Next consider the example in Figure 4.16, a modification of the example in Figure 4.15. Here, a further problem occurs late in the membership algorithm so that processors q and r do not complete the recovery algorithm steps 2-6, while processors s, t, u and v do complete the steps and install the regular configuration $\{q, r, s, t, u, v\}$. It is impossible to guarantee that a processor will install a configuration only if it determines that all other members of that con-

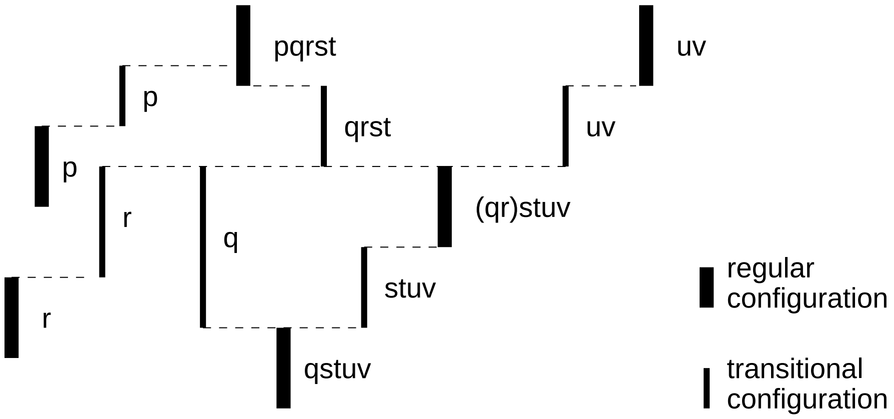


Figure 4.16: Regular and transitional configurations. The vertical lines represent the total orders of messages that have been delivered in the indicated configuration, and the dashed horizontal lines represent Configuration Change messages. The parentheses around q and r indicate that these processors did not actually install the regular configuration $\{q, r, s, t, u, v\}$.

figuration will install it, because that would require common knowledge which is impossible to achieve.

In this example, processor q is subsequently able to reinitiate the membership algorithm to form and install the new ring $\{q, s, t, u, v\}$. During this second attempt to form that ring, processors s, t, u and v are transitioning from the regular configuration $\{q, r, s, t, u, v\}$ to the regular configuration $\{q, s, t, u, v\}$ but processor q is still transitioning from the regular configuration $\{p, q, r, s, t\}$ to the regular configuration $\{q, s, t, u, v\}$, so processor q's transitional configuration is $\{q\}$.

Although processors q and r did not install transitional configuration $\{q, r, s, t\}$, processors s and t have accepted the guarantees of q and r that they have received messages from s and t and may have delivered those messages as safe in the transitional configuration $\{q, r, s, t\}$. Consequently, since processors q and r have received all of the messages needed for the transitional configuration $\{q, r, s, t\}$ and have acknowledged reception of those messages, they must deliver those messages in the transitional configuration they install even if, as in this example, that configuration is smaller.

After the second Configuration Change message, p and r are both members of singleton configurations $\{p\}$ and $\{r\}$ respectively and messages from other processors are not delivered by p or r. When p or r rejoins the other processors in some subsequent configuration, the application programs must update their states, using application-specific algorithms, to reflect activities that were not communicated while the system was partitioned. The Configuration Change messages warn the application that a membership change has occurred, so that the application programs can take appropriate action based on the membership change. Extended virtual synchrony guarantees a consistent order of message delivery across a partition, which is essential if the application programs are to be able to reconcile their states following repair of a failed processor or remerging of the partitioned network.

4.4 Performance

The Totem protocol is designed to provide high performance. Reliable ordered delivery is of little value if the throughput of the protocol is low or the latency to delivery is high. Effective flow control is required to achieve desired performance characteristics.

Flow Control

With point-to-point communication, positive acknowledgment protocols, such as the sliding-window protocol [10], have been refined to provide excellent flow control. However, with broadcast communication, positive acknowledgment protocols result in an excessive number of acknowledgments. Rate-controlled protocols have attracted attention recently [54], but have the disadvantage when used with broadcast protocols that the aggregate rate of all transmitters must be controlled. The maximum transmission rate for each processor must be set to a value that is unacceptably low for applications with bursty communication patterns.

A basic characteristic of reliable broadcast and multicast protocols is that the rate of broadcasting messages cannot exceed the rate at which the slowest processor can receive and process messages. At higher rates of broadcasting, the input buffer of the slowest processor will become full and messages will be lost. Retransmission of those messages will increase the message traffic and reduce the effective transmission rate.

Effective flow control, capable of preventing message loss due to buffer overflow at high transmission rates, is essential to the attainment of high throughput, since retransmissions reduce the available bandwidth and increase the latency. Existing broadcast protocols must be throttled at relatively low rates of broadcasting to avoid high rates of message loss and, thus, exhibit poor performance when the traffic is bursty.

The Data Structures

Regular Token

The following field is added to the regular token:

- *retrans_round*: The number of retransmissions sent in the last round of the token.

Local Variables

Each processor maintains the following local variables:

- *last_token_seq*: The message sequence number on the token ($token.seq$) last time it was received.

- *window_size*: The number of messages which can be sent by all processors during a rotation of the token.

- *each_time*: The maximum number of messages which can be sent by this processor during any visit of the token.

- *last_retrans*: The number of retransmissions by this processor on the last visit of the token.

- *allowed_to_send*: The number of messages this processor can send on this token visit.

The *retrans_round* field of the token is initialized to zero when the token is generated. The *last_token_seq* and *last_retrans* fields are intitialized to zero by the processor in the membership algorithm during the formation of the ring. The values for *window_size* and *each_time* are determined heuristically.

The Algorithm

The Totem protocol uses a simple flow-control algorithm to control the number of messages broadcast during one rotation of the token. If a processor is unable to process messages at the rate at which they are broadcast, one or more messages will be in its input buffer when the token arrives. Before processing the token and broadcasting messages, a processor must empty its input buffer. Thus, the rate of broadcasting messages is reduced to the rate at which messages can be processed by that processor. If the *window_size* is limited by the size of each processor's input buffer, buffer overflow cannot occur.

In practice, it is possible to increase the *window_size* to a larger number of messages than the input buffer can contain. As the token rotates around the ring, a processor can receive and process messages, freeing buffer space for subsequent messages in the same rotation. An appropriate value for *window_size* can be found by experimentation, but care must be taken to allow for processors that may occasionally be heavily loaded and for other uncontrolled traffic on the network.

There is also a limit on the number of messages an individual processor is allowed to send during one visit of the token. This limit contributes to fairness in distributing *window_size* among the processors on the ring and is specified by *each_time*. Retransmissions are included in the number of messages sent. The pseudocode executed by a processor to calculate the number of messages this processor is allowed to send on this token visit is given in Figure 4.17.

Before forwarding the token a processor updates the token *retrans_round* field by subtracting *last_retrans* and adding the number of messages retransmit this token visit. The processor then sets *last_retrans* to equal the number of messages retransmit this token visit. The processor also set *last_token_seq* to equal the *seq* field in the token.

```
Flow_Control
    allowed_to_send := window_size + each_time −
            (token.seq − last_token_seq + token.retrans_round)
    if ( allowed_to_send > each_time ) then
        allowed_to_send := each_time
    endif
    if ( allowed_to_send < 0 ) then
        allowed_to_send := 0
    endif
```

Figure 4.17: Algorithm executed by a processor or a gateway to determine how many messages can be sent on this token visit.

Although more sophisticated flow control schemes can be used, they have not proven necessary.

Analytical Model

The latency to order a message in the Totem single-ring protocol is a function of the token rotation time. Under low loads, where the probability is small that a processor is prevented by the flow-control mechanism from broadcasting all of its pending messages during each visit of the token, the latency to agreed delivery is approximately one-half of a token rotation time and the latency to safe delivery is approximately two token rotation times. Assuming that no messages are lost, we now calculate the token rotation time. We use the following denotations:

N	Number of processors on the ring
T	Token rotation time
ρ	Utilization of the communication medium
a	Mean time to broadcast one message
r	Ratio of the mean time to process and broadcast one message to the mean time to broadcast
b	Token processing and transmission time for one processor
m	Mean number of messages broadcast by one processor during one visit of the token

M Maximum number of messages that all processors are allowed to broadcast in any token rotation, *i.e. window_size*

In the calculation we assume that the time to receive and process a message is approximately equal to the time to process and broadcast the message. If the receive time is substantially greater than the broadcast time, then a and r should be derived from the time to receive messages rather than the time to broadcast messages.

The useful utilization of the communication medium is given by

$$\rho = \frac{Nma}{Nb + Nmra} = \frac{ma}{b + mra}$$

from which it follows that

$$a = \frac{b\rho}{m(1 - r\rho)}$$

Thus, the token rotation time is

$$T = Nb + Nmra = \frac{Nb}{1 - r\rho}$$

Furthermore, the maximum token rotation time is $Nb + Mra$, while the maximum utilization of the communication medium is $Ma/(Nb + Mra)$.

Simulation

A simulator has been built using the C programming language to allow study and debugging of the Totem single-ring protocol [20, 21]. In the simulator, the object code of the Totem single-ring protocol implementation is linked to a simulated communication medium. The simulated communication medium allows the injection of faults, partitions, and merges. The distributed nature and high performance of the Totem single-ring protocol make it difficult to study and debug when running on an actual network. The simulator can instead be run step-by-step and protocol behavior during failures can be studied in-depth. The simulator also provides an environment in which systems with more processors than are physically available can be studied.

A graphical interface for the simulator has also been developed [33]. The graphical interface displays the current ring membership and delivery characteristics for each of the processors. In particular, the monitor shows which messages

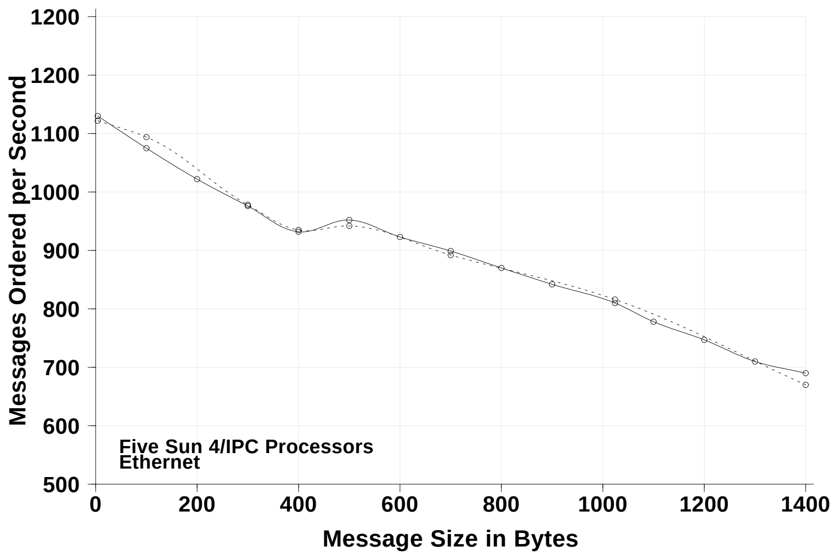


Figure 4.18: The number of ordered broadcast messages per second for various message sizes for a network of five Sun 4/IPC processors running the Totem single-ring protocol. The solid line indicates throughput when the traffic is generated by all the processors, and the dotted line shows the throughput when the messages are generated by a single processor.

were delivered in the old, transitional, and new configurations respectively for each processor.

Implementation

The Totem single-ring reliable ordering and membership protocol has been implemented using the C programming language on a network of Sun 4/IPC workstations connected by an Ethernet. The implementation uses the standard UDP broadcast interface within the Unix operating system (SunOS 4.1.1). One UDP socket is used for all broadcast messages, and a separate UDP socket is used by each processor to receive the token from its predecessor on the ring.

Measurements from the implementation show excellent performance. Figure 4.18 shows the number of messages ordered per second for messages of various sizes. These measurements were made on a network of five Sun 4/IPC workstations with the *window_size* set to the maximum value for which message loss is negligible in order to maximize throughput. Note that with 1024 byte mes-

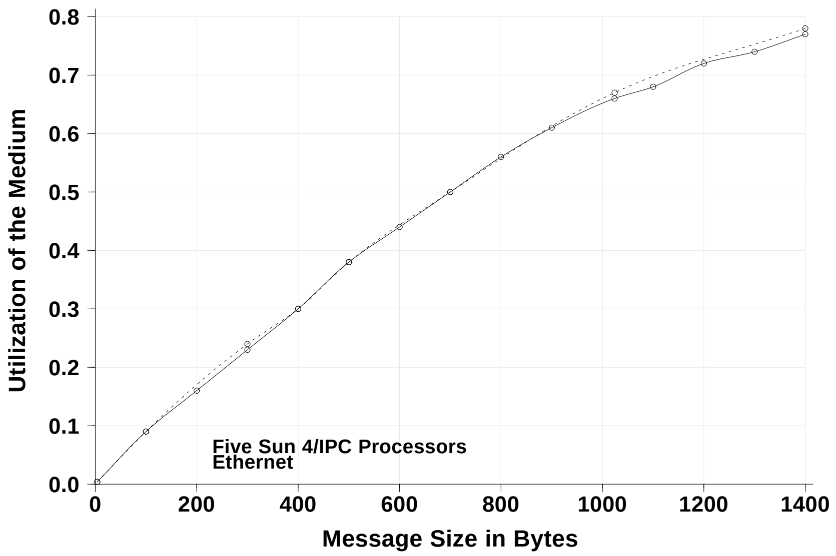


Figure 4.19: The utilization of the Ethernet for various message sizes for a network of five Sun 4/IPC processors running the Totem protocol. The solid line indicates utilization when the traffic is evenly distributed across all five processors, and the dotted line shows the utilization when the messages are generated by a single processor.

sages ($window\_size = 70$ when all 5 processors are sending), 810 messages are ordered per second. Reducing the message size to 600 bytes results in over 900 ordered messages per second. Further reducing the message size to 4 bytes to measure the protocol overhead results in 1130 messages ordered per second. As can also be seen from Figure 4.18 the number of messages ordered per second is relatively unaffected by the number of senders; similar throughput figures are obtained when all messages are broadcast by a single processor on the ring. The highest prior rates for asynchronous fault-tolerant ordered broadcast messages known to us for 1024 byte messages are about 300 messages per second for the Transis protocol using the same equipment and for the Amoeba protocol [31] using equipment of similar performance.

A concern about token-passing protocols, such as Totem, is that the token-passing overhead reduces the transmission rate available for messages. Figure 4.19 depicts the useful utilization (excluding transmission of the token and message headers) of the Ethernet achieved by the Totem protocol. With large

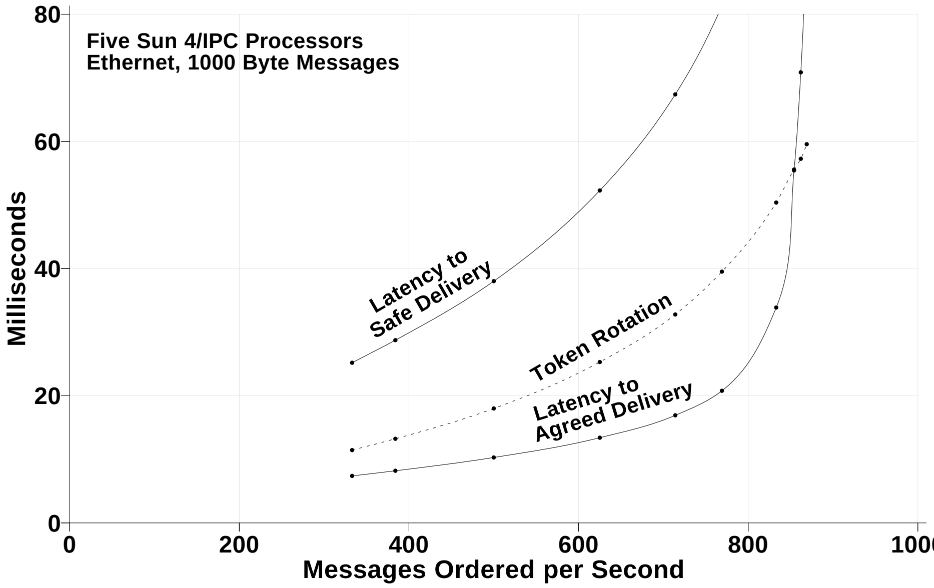


Figure 4.20: The mean token rotation time and the mean latency from generation of a message until it is delivered in agreed order and in safe order by all processors on the ring as a function of load.

messages, a utilization of about 70% is achieved; this may be compared to the approximately 65% utilization that can be achieved by TCP transmitting messages point-to-point from a single source to a single destination with the same equipment. This high utilization is achieved by the single-ring protocol regardless of whether the traffic is distributed equally across the processors or concentrated at a single source.

While Figure 4.18 depicts the maximum transmission rates measured using the Totem protocol over the Ethernet, Figure 4.20 considers performance at lower, more typical loads with Poisson arrivals of messages and 1000 byte messages. It shows the token rotation time and the latency from generation of a message until it is delivered in agreed order by all processors on the ring. At low loads (*e.g.*, 400 ordered messages per second which is much more than the maximum throughput for prior protocols), the latency achieved by Totem is under 10 milliseconds. Even at 50% useful utilization of the Ethernet (625 messages per second), the latency is still only about 13 milliseconds. Note that the latency to agreed delivery is slightly more than half the token rotation time and the latency to safe delivery is approximately twice the token rotation time, except at very high loads where queueing delays dominate. Also note that the latency increases linearly with the size of the membership. These latency results were measured in the simulator and verified on the implementation.

Also a concern with token passing protocols is the time to reach consensus on membership and to begin rotation of the token after loss of the token. We measured the time to membership by disabling the token retransmission mechanism and intentionally losing the token. In a test with the processors sending 1024 byte messages the time to complete the membership process, generate a new token, and return to normal operation is on average 40 milliseconds plus the token loss timeout; for these experiments the token loss timeout was 100 milliseconds. With the token retransmission mechanism reactivated, the average time to return to normal operation after loss of the token is 16 milliseconds.

4.5 Proof of Correctness

Membership

Uniqueness of Configurations

Theorem 4.1 *Each configuration identifier is unique; moreover, at any time a processor is a member of at most one configuration.*

Proof. On startup a processor forms a singleton ring containing only itself. On forwarding the Commit token for a proposed new ring on its first rotation, a processor abandons its current ring and becomes committed to the new ring. The processor then forms a transitional configuration consisting of members of its old ring and new ring. If the processor installs the new ring, it delivers two Configuration Change messages. The first initiates a transitional configuration and terminates the processor's regular configuration (old ring), and the second initiates the regular configuration (new ring) and terminates the transitional configuration. Thus, a processor is a member of at most one configuration at a time.

The identifier of a configuration (either regular or transitional) consists of a "ring" sequence number (which is stored in stable storage) and the processor identifier of the representative (which is chosen deterministically from the members). We now show that, if there is a processor that is a member of two different configurations C_1 and C_2 with the same representative identifier, then C_1 and C_2 have different ring sequence numbers. Since processor identifiers are unique and do not change in time even if a processor fails and restarts, there is a processor p that is the representative of both C_1 and C_2. By the above, p is a member of at most one configuration at a time, say p is a member of C_1 before p is a member of C_2. Then, since the ring sequence number of C_2 is greater than the maximum of the ring sequence numbers of the immediately prior configurations of the members of C_2 when C_2 was formed, the ring sequence number of C_2 is greater than the ring sequence number of C_1. □

Consensus

Theorem 4.2 *All of the processors that install a configuration determine that the members of the configuration have reached consensus on the membership.*

Proof. The configuration that a processor installs when it delivers a Configuration Change message is based on the ring that the processor installs. If a processor installs a ring, then that processor was previously in the Commit state and forwarded the Commit token twice. A processor forwards the Commit token the first time only if the membership in the token is equal to the difference of its *my_proc_set* and *my_fail_set*; otherwise, it discards the Commit token. When a processor forwards the Commit token the second time, it determines that all members of the ring have reached consensus on the membership. □

Termination

Theorem 4.3 *If a configuration ceases to exist for any reason, such as processor failure, network partitioning or token loss, then within a bounded time every processor of that configuration will install a new configuration or will fail before doing so.*

Proof. If a configuration ceases to exist, then the token has either been lost or discarded by a processor and each processor in the Operational state either incurs a Token Loss timeout and shifts to the Gather state, or receives a Join message and shifts to the Gather state.

Because of the Consensus and Token Loss timeouts, a processor can spend only a bounded time in the Gather state without increasing either *my_proc_set* or *my_fail_set*. Because of the Token Loss timeout, a processor can spend only a bounded time in the Commit state. Because of the Token Loss timeout and failure-to-receive mechanism, a processor can spend only a bounded time in the Recover state. Each time a processor returns to the Gather state from the Commit or Recover state it increases either *my_proc_set* or *my_fail_set* before leaving the Gather state. In an n-processor system, a processor can increase one or the other of these sets at most $2n-2$ times before it reaches consensus on a singleton membership with n processors in *my_proc_set* and $n-1$ processors other than itself in *my_fail_set*. If a processor reduces the membership to a singleton set containing only itself, then it will necessarily install the singleton ring; otherwise, it will install a new ring with a larger membership. In either case, the processor will install a transitional configuration and a new regular configuration. □

Configuration Change Consistency

Theorem 4.4 *Processors that are members of the same configuration C_1 deliver the same Initiate Configuration C_1 message to begin the configuration. Furthermore, if two processors install a configuration C_2 directly after C_1, then the processors deliver the same Configuration Change message to terminate C_1 and initiate C_2.*

Proof. If a processor installs configuration C_1, it delivers a Configuration Change message containing the Initiate Configuration C_1 message and the membership of C_1. Thus, if p and q are both members of C_1, they have delivered the same Initiate Configuration C_1 message.

If a processor installs configuration C_2 directly from C_1, it delivers a Configuration Change message containing the Terminate Configuration C_1 message and the Initiate Configuration C_2 message.

Thus, if p and q both install C_2, they deliver the same Configuration Change message to terminate C_1 and initiate C_2. □

Ordering

Reliable Delivery

Theorem 4.5 *Each message m has a unique identifier.*

Proof. The identifier of a regular message m consists of the *ring_id* of the regular configuration in which m was originated, the message sequence number of m (which is greater than 0), and the *conf_id* 0. By Theorem 4.1, the configuration identifiers are unique. By the single-ring ordering protocol, the processor that originates m increments the *seq* field of the token and sets the sequence number of m to this *seq*; thus, within a configuration the message sequence number of m is unique. Since the *conf_id* of a regular message is 0 and the *conf_id* of a Configuration Change message is a *ring_id* which is greater than 0, the *conf_id* field distinguishes a regular message from a Configuration Change message.

The identifier of a Configuration Change message that initiates a regular configuration consists of the *ring_id* of that configuration, the message sequence number 0, and the identifier of the old transitional configuration from which this processor is transitioning as the *conf_id* field. The identifier of a Configuration Change message that initiates a transitional configuration consists of the ring_id of the preceding regular configuration, the largest message sequence number on the old ring, and the identifier of the new transitional configuration to which this processor is transitioning as the *conf_id* field. If two Configuration Change messages have the same source ring identifier, then the *conf_id* is either the identifier of a transitional configuration preceding the regular configuration or the identifier of a transitional configuration following the regular configuration. By Theorem 4.1, configuration identifiers are unique. The statement now follows. □

Theorem 4.6 *If processor p delivers message m, then p delivers m only once. Moreover, if processor p delivers two different messages, then p delivers one of those messages strictly before it delivers the other.*

Proof. If processor p delivers message m, then p delivers m either in the regular configuration in which it was originated or in an immediately following transitional configuration. By Theorem 4.14, within those configurations p delivers messages in sequence number order. □

Theorem 4.7 *A processor p delivers its own messages unless it fails.*

Proof. Assume that processor p does not fail. If p is a member of a singleton configuration and thus of a singleton ring, then it delivers m immediately. Suppose now that p is a member of a configuration and thus of a ring with two or more members. There are three possibilities: (1) Either the *aru* in the token will advance above the sequence number of m and will not be lowered again, in which case p will deliver m, or (2) the failure-to-receive mechanism will be invoked, or (3) the membership algorithm will be invoked due to a Token Loss timeout or reception of a foreign message. In cases (2) and (3), by Theorem 4.3, p will install a new ring and thus a new configuration within a bounded time. By Theorem 4.12, p will deliver all messages that were originated by the members of *my_deliver_memb* and thus of the transitional configuration associated with the regular configuration it installs; in particular, p will deliver its own message m. This may involve reduction to a singleton configuration. □

Theorem 4.8 *A processor p delivers all of the messages originated in its regular configuration C unless a configuration change occurs.*

Proof. If no configuration change occurs, then the failure-to-receive mechanism is not invoked. Let m be a message that has been broadcast by a member of C. When processor p receives the token, either p has received m or p determines from the *seq* field of the token that it did not receive m and thus includes the sequence number of m in the *rtr* field of the token. Because the number of messages broadcast in C with sequence numbers at most equal to that of m is finite, the number of processors in C is finite, and the failure-to-receive mechanism is not invoked, each processor in C will eventually receive and deliver all messages with sequence numbers at most equal to that of message m, in particular, m itself. □

Theorem 4.9 *If processor p delivers message m originated in configuration C, then p is a member of C and p has installed C. Moreover, p delivers m in C or in a transitional configuration between C and the next regular configuration it installs.*

Proof. According to the algorithm, processor p delivers message m originated in configuration C only after it has installed C (delivered Initiate Configuration C). Moreover, p delivers m either as a member of the configuration C in which m was originated or as a member of the subsequent transitional configuration that consists of processors in both C and the next regular configuration that p installs. □

Theorem 4.10 *If processors p and q are both members of consecutive configurations C_1 and C_2, then p and q deliver the same set of messages in C_1 before delivering the Configuration Change message that terminates C_1 and initiates C_2.*

Proof. There are two cases to consider: (1) C_1 is a regular configuration and C_2 is a transitional configuration, and (2) C_1 is a transitional configuration and C_2 is a regular configuration.

In case (1) processors p and q have exchanged messages and have the same set of messages for C_1 that were rebroadcast on the new ring following C_2 with sequence numbers up through *my_install_seq*. By the recovery algorithm, p and q both deliver in C_1 all messages up to a gap in the sorted message sequence or a message requiring safe delivery with a sequence number greater than *high_ring_delivered*. (A message with sequence number less than or equal to *high_ring_delivered* was delivered in C_1 by some processor and, thus, can be delivered in C_1 by this processor.) Processors p and q then deliver the Configuration Change message that terminates C_1 and initiates C_2.

In case (2) processors p and q must both have been members of the regular configuration C_0 for which C_1 is the transitional configuration between C_0 and C_2. By case (1) p and q have the same set of messages for C_0 and have delivered the same subset of those messages before delivering the Configuration Change message that terminates C_0 and initiates C_1. By the recovery algorithm, p and

q both deliver in C_1 all remaining messages from C_0 up to the first gap in the sorted message sequence (which were not delivered before the first Configuration Change message) and also all subsequent messages that were originated in C_0 by processors in *my_delivery_memb*. They then deliver the Configuration Change message that terminates C_1 and initiates C_2. $\Box$

Delivery in Causal Order for Configuration C

Theorem 4.11 *If m_1 precedes m_2 in the Lamport causal order and processor p delivers both m_1 and m_2, then p delivers m_1 before p delivers m_2.*

Proof. First we show for Lamport's causal precedence relations that if processor q originates message m_3 before processor q originates message m_4 or, if q receives and delivers m_3 before q originates m_4, then the identifier of m_3 is less than the identifier of m_4 in the lexicographical order of identifiers (src_ring_id,seq,conf_id).

The local variable *my_ring_id.seq* is recorded to stable storage to ensure that any message originated by q after q recovers from a failure is ordered after any message received or originated by q before its failure.

When processor q originates a message, it increments the *seq* field of the token and sets the sequence number of the message to this *seq*, ensuring that the sequence number of the message is higher than that of any message already originated or delivered on that ring. The *ring_id* of the message is the ring identifier of the ring of which q was a member when it originated the message. The *ring_id.seq* of that ring is greater than the *ring_id.seq* of any previous ring of which q was a member.

By the transitivity on the lexicograpical order of identifiers, if m_1 precedes m_2 in the closure of the Lamport causal precedence relations, then the identifier of m_1 is less than the identifier of m_2.

By Theorem 4.17, if the identifier of m_1 is less than the identifier of m_2, then m_1 precedes m_2 in the Global Delivery Order. By Theorem 4.18, if p delivers both m_1 and m_2, and if m_1 precedes m_2 in the Global Delivery Order, then p delivers m_1 before p delivers m_2. $\Box$

Theorem 4.12 *If processor p delivers message m_2, and m_1 precedes m_2 in the causal order for configuration C, then p delivers m_1 before p delivers m_2.*

Proof. The sequence number of the Initiate Configuration message for each regular configuration has the sequence number 0 and the regular messages begin with sequence number 1.

By a simple induction based on message sequence numbers it follows that, for any message m_1 originated in configuration C that precedes m_2 in the causal order, the sequence number of m_1 is at most equal to the sequence number of m_2. Furthermore, if the processor that originated m_1 is different from the processor q that originated m_2, then the sequence number of m_1 is at most equal to processor q's *my_aru* when q originated m_2.

Now either processor p delivers message m_2 in configuration C or p delivers m_2 in the transitional configuration C_2 following C. Suppose then that processor p delivers message m_2 in the transitional configuration C_2 following C and that processor q originated m_2. By the delivery guarantee, p delivers all messages with sequence numbers up to the first gap in the message sequence and also all messages originated by processors in p's *my_deliver_memb* with sequence numbers up through *high_ring_delivered*. There are two cases to consider: (1) q is a member of p's *my_deliver_memb*, and (2) q is not a member of p's *my_deliver_memb*.

(1) Since m_1 precedes m_2 in the causal order and since q originated m_2, either q also originated m_1 and thus, by the delivery guarantee, p has delivered m_1, or q did not originate m_1 and the sequence number of m_1 is less than or equal to q's *my_aru* when q originated m_2. In the latter case, q has the complete sequence of messages up through m_1. Since p delivered m_2 in C_2, the message exchange was completed and p also has the complete sequence of messages up through m_1 and thus, by the delivery guarantee, p delivers m_1 before p delivers m_2.

(2) Since p delivers m_2 and q is not a member of p's *my_deliver_memb*, the sequence number of m_2 is less than that of a message in the first gap of the message sequence. Since the sequence number of m_1 is less than or equal to that of m_2, by the delivery guarantee, p delivers m_1 before p delivers m_2.

By the algorithm, a processor p delivers a Terminate Configuration message with an Initiate Configuration message. After the Initiate Configuration message is delivered, p no longer delivers messages in the old configuration so the Terminate Configuration message is delivered as the last message in the old configuration. □

Delivery in Agreed Order for Configuration C

Theorem 4.13 *The Configuration Delivery Order for C is a total order.*

Proof. By Theorem 4.5, the messages delivered in C have unique sequence numbers which, as a subset of the non-negative integers, form a total order. □

Theorem 4.14 *If processor p delivers message m_2 in configuration C and m_1 is any message that precedes m_2 in the Delivery Order for Configuration C, then p delivers m_1 in C before p delivers m_2.*

Proof. If m_1 is any message that precedes m_2 in the Configuration Delivery Order for C, then the sequence number of m_1 is at most equal to the sequence number of m_2. By the algorithm, every processor in C delivers messages in sequence number order and does not deliver message m_2 until it has delivered all messages in the Configuration Delivery Order for C with smaller sequence numbers. □

Delivery in Safe Order for Configuration C

Theorem 4.15 *If a processor delivers message m in configuration C and the originator of m requested safe delivery, then the processor has determined that each processor in C has received m, and will deliver m or will fail before installing a new regular configuration.*

Proof. Let q be a processor in C that does not fail before installing a new regular configuration. There are two cases to consider.

(1) To deliver m in safe order in regular configuration C, processor p must forward the token on two consecutive rotations with the *aru* at least equal to the sequence number of m. Thus, p determines that, when q forwarded the

token on the first of those rotations, q's *my_aru* must have been at least equal to the sequence number of m. Consequently, p determines that, if the token is not lost, then q will receive the token on the second of those rotations with the *aru* at least equal to the sequence number of m and will then deliver m.

Moreover, p determines that, if the token is lost, then the first gap in the sequence of messages received by q must correspond to a sequence number greater than that of m and, thus, that q will deliver m before it installs a subsequent regular configuration.

(2) To deliver m in safe order in transitional configuration C, processor p must forward the token on three consecutive rotations with the *retrans_flg* set to false and, on the last two of those rotations, with the *aru* at least equal to *my_install_seq* (*my_install_seq* is the largest new ring sequence number of all old ring messages retransmitted on the new ring and, thus, is at least equal to the sequence number of m on the new ring). Processor p then determines that each processor q forwarded the token on two consecutive rotations with its *retrans_flg* set to false and on one rotation with *my_aru* at least equal to *my_install_seq* and *my_deliver_memb* equal to C. Thus, p determines that q has received m and will deliver m in whichever configuration q subsequently installs. □

Extended Virtual Synchrony

Theorem 4.16 *If processor p delivers message m in configuration C, then the requirements for agreed or safe delivery are satisfied.*

Proof. This follows from the preceding theorems. □

Theorem 4.17 *The Global Delivery Order is a total order.*

Proof. By Theorem 4.5, each message has a unique identifier. The identifier of a regular message m consists of the *ring_id* of the regular configuration on which m was originated, the message sequence number for m, and the *conf_id* 0. The identifier of a Configuration Change message that initiates a regular configuration consists of the *ring_id* of that configuration, the message sequence number 0, and the identifier of the old transitional configuration from which this processor is transitioning as the *conf_id* field. The identifier of a Configuration Change message that initiates a transitional configuration consists of the identifier of the preceding regular configuration, the largest message sequence number on the old ring, and the identifier of the transitional configuration to which this processor is transitioning as the *conf_id* field.

The precedes relation of the Global Delivery Order is the lexicographical order on the set of identifiers (*ring_id*,*seq*,*conf_id*). This lexicographical order is a total order and, thus, the Global Delivery Order is a total order. □

Theorem 4.18 *If processor p delivers messages m_1 and m_2, and m_1 precedes m_2 in the Global Delivery Order, then p delivers m_1 before p delivers m_2.*

Proof. Let ($ring\_id_1$,seq_1,$conf\_id_1$) and ($ring\_id_2$,seq_2,$conf\_id_2$) be the identifiers of messages m_1 and m_2, respectively. By Theorem 4.9, a processor only delivers messages originated in configurations of which it is a member and, thus, processor p is a member of the configurations with identifiers $ring\_id_1$ and $ring\_id_2$. Without loss of generality, we assume that $(ring\_id_1, seq_1, conf\_id_1) < (ring\_id_2, seq_2, conf\_id_2)$. The proof is an exhaustive case analysis.

If $ring\_id_1 < ring\_id_2$, then p was a member of the configuration with identifier $ring\_id_1$ before it was a member of the configuration with identifier $ring\_id_2$, because the ring sequence number is increased each time a processor shifts to the Recover state within the membership algorithm. According to the algorithm, processor p delivers all messages that were originated on the ring with identifier $ring\_id_1$ before it delivers any message that was originated on the ring with identifier $ring\_id_2$.

If $ring\_id_1 = ring\_id_2$, $seq_1 < seq_2$, $conf\_id_1 = 0$, and $conf\_id_2 = 0$, then m_1 and m_2 are regular messages originated in the regular configuration with identifier $ring\_id_1$. Since regular messages originated in the same configuration are delivered in sequence number order, processor p delivers m_1 before p delivers m_2.

If $ring\_id_1 = ring\_id_2$, $seq_1 \leq seq_2$, $conf\_id_1 = 0$ and $conf\_id_2 \neq 0$, then m_1 is a regular message and m_2 is a Configuration Change message that initiates a transitional configuration following the regular configuration with identifier

$ring\_id_1$. This Configuration Change message is delivered after the regular message with sequence number seq_2 and, hence, after m_1.

If $ring\_id_1 = ring\_id_2$, $seq_1 < seq_2$, $conf\_id_1 \neq 0$ and $conf\_id_2 = 0$, then m_1 is a Configuration Change message and m_2 is a regular message delivered in the configuration initiated by m_1. This Configuration Change message is delivered before the first regular message delivered in that configuration and, hence, before m_2.

If $\text{ring\_id}_1 = ring\_id_2$, $seq_1 = 0$, $seq_2 > 0$, $conf\_id_1 \neq 0$ and $conf\_id_2 \neq 0$, then m_1 is a Configuration Change message that initiates a regular configuration and m_2 is a Configuration Change message that initiates a transitional configuration and terminates the regular configuration initiated by m_1. By the algorithm, processor p delivers m_1 before p delivers m_2.

If $\text{ring\_id}_1 = ring\_id_2$, $seq_1 > 0$, $seq_2 > 0$, $conf\_id_1 \neq 0$ and $conf\_id_2 \neq 0$, then m_1 and m_2 are both Configuration Change messages that initiate different transitional configurations and terminate the same regular configuration. By the algorithm, no processor delivers both m_1 and m_2.

If $\text{ring\_id}_1 = ring\_id_2$, $seq_1 = 0$, $seq_2 = 0$, $conf\_id_1 \neq 0$, and $conf\_id_2 \neq 0$, then one of three cases arises: (1) Messages m_1 and m_2 are Configuration Change messages that initiate the same regular configuration and terminate different transitional configurations. By the algorithm, no processor delivers both m_1 and m_2. (2) Messages m_1 and m_2 are Configuration Change messages that initiate different transitional configurations and terminate the same regular configuration, and no regular message was delivered in the regular configuration. By the algorithm, no processor is a member of two different transitional configurations that follow the same regular configuration. By the algorithm, no processor delivers both m_1 and m_2. (3) Message m_1 is a Configuration Change message that initiates a regular configuration and m_2 is a Configuration Change message that initiates a transitional configuration following that regular configuration, and no regular message was delivered in the regular configuration. In this case, $conf\_id_1$ is the identifier of a transitional configuration that precedes the regular configuration with identifier $ring\_id_1$, and $conf\_id_2$ is the identifier of a transitional configuration that follows that regular configuration. By the algorithm, processor p delivers m_1 before p delivers m_2.

In any case, processor p delivers m_1 before p delivers m_2. The Global Delivery Order is, thus, the set of messages delivered by all of the processors.$\square$

4.6 Summary

The single-ring protocol provides fast reliable ordered delivery of messages in a broadcast domain where processors may fail and the network may become partitioned. A token circulating around a logical ring imposed on the broadcast domain is used to recover lost messages and to order messages on the ring. The protocol provides delivery of messages in agreed and safe order.

The membership algorithm handles processor failure and recovery, network partitioning and remerging, and loss of all copies of the token. The concept of extended virtual synchrony has been introduced to ensure consistent actions by processors that fail and are repaired with their stable storage intact and in networks that partition and remerge. A recovery algorithm that maintains extended virtual synchrony during recovery after a failure has been provided.

The flow-control algorithm of Totem avoids message loss due to overflow of the input buffers and provides substantially higher throughput than existing total ordering protocols. With the high performance of the single-ring protocol, there is no need to provide a weaker message ordering service, such as partially ordered causal delivery, because totally ordered agreed delivery can be provided at no greater cost. Moreover, partially ordered causal delivery may lead to inconsistencies in different components of a partitioned network.

Although the single-ring protocol was originally designed to be executed by processors in a local-area network, it can be used on top of any communication protocol that provides unreliable multicast or broadcast communication; a single broadcast domain is not necessarily confined to a local-area network. The single-ring protocol can also be executed by a subset of the processors in a local-area network.

Chapter 5

The Multiple-Ring Protocol

The multiple-ring protocol provides agreed and safe delivery of messages across broadcast domains interconnected by gateways, as well as membership and topology maintenance. It uses the single-ring total ordering algorithm to provide reliable ordered delivery within each broadcast domain; the gateways forward messages between broadcast domains. Timestamps in messages are used to create a global ordering of messages that respects causality and is consistent across the entire network. The timestamp is written periodically to stable storage. When a processor comes up, it reads the value of the timestamp from stable storage and increments that value.

Delivery of messages in total order across the network is relatively straightforward if topology changes never occur. Intuitively, a topology is a set of rings (configurations) such that there is a communication path between any two processors that are members of rings in this set. A topology change is inevitable when a failure, restart, partition or remerge event occurs. The gateways executing the multiple-ring membership algorithm use the Configuration Change messages from the single-ring membership algorithm to identify changes to network topology information. Network Topology messages are used to inform the gateways and processors on a ring of the network topology. Topology Change messages are sent by a gateway to notify the other gateways and processors in the network of a change in the topology due to a configuration change. Topology Change messages are delivered to the application.

The gateways perform the same functions as the processors; in particular, they can send messages from, and deliver messages to, the application. Besides forwarding messages, each gateway maintains the current view of the network topology. An early version of the protocol is described in [1].

In this chapter we use the term *ring* to mean either regular configuration or transitional configuration, and *configuration change* to mean a change to either a regular configuration (ring) or to a transitional configuration within the single-ring protocol. The term *configuration* used in the statement of the requirements for the protocol in Chapter 3 should be interpreted as, and is replaced by, *topology* in this chapter. We use the term *processor* to mean either processor or gateway unless explicitly stated otherwise. In the multiple-ring protocol messages are ordered by timestamp, source ring identifier, message type and configuration identifier, but we will often simply say that they are ordered by timestamp.

5.1 The Total Ordering Algorithm

First we describe the operation of the Totem multiple-ring total ordering algorithm without considering topology changes. The difficult task of dealing with topology changes is considered in Section 5.2. Pseudocode for the total ordering algorithm is included in Figures 5.1, 5.3 and 5.4.

The Data Structures

Local Data Structures

Each processor and gateway maintains the following data structures to track the messages that are received and to implement ordering:

For each ring in the network,

- *ring_id*: The unique ring identifier generated by the single-ring protocol.
- *recv_msgs*: A list of received messages that were originated by a processor on the ring and that have not yet been delivered to the

application. This list is sorted in increasing order by timestamp and message type.

- *max_timestamp*: The highest timestamp of a message received that was originated on the ring.
- *min_timestamp*: The lowest timestamp of a message in *recv_msgs* for the ring. If there are no messages in *recv_msgs*, then *min_timestamp* equals *max_timestamp*.

The *ring_table* contains an entry for each ring (regular and transitional configuration) in the network with the above information.

For each directly attached ring

- *my_guar_vector*: The length of *my_guar_vector* is the number of rings in the network. Each vector component contains the value of the highest *timestamp* of a message received for the corresponding ring.

For the entire network

- *cand_msgs*: A sorted list containing the lowest entry in *recv_msgs* for each ring. This list is kept in increasing order sorted by (timestamp, source ring identifier, message type, conf). If *recv_msgs* for a ring is empty, then the entry in *cand_msgs* for the ring is (min_timestamp, ring identifier, regular, 0).
- *guarantee*: An array with rows that are the guarantee vectors received from the gateways on the other rings, one row for each ring in the network. Each column of the array corresponds to messages originated on a particular ring.

Maintained only by the gateways

- *gway_id*: The identifier of this gateway. The *gway_id* is chosen deterministically from the two single-ring processor identifiers for this gateway.

Added to the single-ring protocol

- *my_timestamp*: The highest message timestamp known to this processor.
- *my_stable_timestamp*: The value of the timestamp last written to stable storage.
- *timestamp_interval*: A constant that determines how often the timestamp is written to stable storage.
- *my_future_ring_seq*: The highest ring sequence number known to this processor.

The values of *my_stable_timestamp* and *my_future_ring_seq*, and of the timestamp and ring sequence number in stable storage are all initially 0. The value of the constant *timestamp_interval* is determined as a configuration parameter.

When a processor comes up either initially or after a failure, it is a member of one ring for each interface to a broadcast domain. The number of components of *my_guar_vector* is initialized to the number of interfaces to broadcast domains; each component has an initial timestamp of -1. The rows of the *guarantee* array correspond to the *my_guar_vector*s for the directly attached rings. *My_timestamp* and the *min_timestamp* and *max_timestamp* for each ring are set to zero; *recv_msgs* for each ring is empty. The processor reads the value of *my_stable_timestamp* from stable storage. It then sets *my_stable_timestamp* to *my_stable_timestamp* + *timestamp_interval*. The processor writes the value of *my_stable_timestamp* to stable storage and waits for the completion of that write. It then sets *my_timestamp* to the value of *my_stable_timestamp*.

On receipt of a Configuration Change or Topology Change message introducing a new ring, a processor adds the data structure for the new ring to the *ring_table*. The *ring_id* for the new ring is obtained from the Configuration Change or Topology Change message. The *max_timestamp* and *min_timestamp* are set to the timestamp in the Configuration Change or Topology Change message, *recv_msgs* is set to empty, and a new entry corresponding to this ring is added to *cand_msgs*.

Regular Message

In addition to the single-ring protocol header, each regular message has a multiple-ring protocol header containing the following fields:

- *src_sender_id*: The identifier of the processor that originated the message.
- *timestamp*: The message timestamp.
- *src_ring_id*: The identifier of the ring on which the message was originated.
- *type*: Regular.
- *conf_id*: 0

The last four fields constitute the identifier of the message. The *src_sender_id*, *timestamp*, and *src_ring_id* fields are set by the single-ring protocol on transmission of the message at the site that generated the message. These fields are not changed when a message is forwarded or retransmitted. The *sender_id*, *seq* and *ring_id* in the single-ring protocol header are reset each time the message is forwarded to a new ring.

Guarantee Vector Message

In addition to the fields in a regular message, each Guarantee Vector message contains the following field:

- *guar_vector*: The current *my_guar_vector* for a ring containing the gateway that originated the Guarantee Vector message.

The *src_ring_id* field of the Guarantee Vector message is set to the ring identifier of the ring corresponding to *my_guar_vector*.

Guarantee Vector messages are broadcast periodically by the gateways to the other gateways and processors in the network. The timestamp of a Guarantee Vector message assures a recipient that it will not receive a message with a lower timestamp from the source ring of the Guarantee Vector message; this allows messages to be delivered in agreed order. The contents of a Guarantee Vector message indicate which messages have been received from other rings by the gateways and processors on the source ring of the Guarantee Vector

```
if my_guar_vector[msg source ring] < msg.timestamp then
    my_guar_vector[msg source ring] := msg.timestamp
endif
if msg.timestamp <= msg source ring.max_timestamp then
    discard message
else
    if amgateway then
        forward message
    endif
    source ring.max_timestamp := msg.timestamp
    add message to recv_msgs of source ring
    if recv_msgs of msg source ring contains only one message then
        source ring.min_timestamp := msg.timestamp
        update entry for source ring in cand_msgs
    endif
    call deliver_msgs
endif
```

Figure 5.1: Algorithm executed by a processor on receipt of a regular message.

message; this allows messages to be delivered in safe order. Guarantee Vector messages are forwarded throughout the network, but they are not delivered to the application.

The Algorithm

The multiple-ring protocol relies on the single-ring total ordering algorithm to provide reliable ordered delivery of messages within a broadcast domain. Message sequence numbers remain local to a ring and cannot be used for ordering across the entire network; instead, messages are ordered by timestamp.

We first describe the mechanisms added to the single-ring protocol to handle the timestamping of messages. In addition to the fields described in Chapter 4, the Regular and Commit tokens contain a *timestamp* field. When a new ring is being formed, the timestamp field in the Commit token is used to determine the highest *my_timestamp* of any of the processors on the ring. On the first rotation of the Commit token, a processor compares *my_timestamp* to the *timestamp* field in the Commit token and sets the *timestamp* field to the larger of the two values.

The timestamp field in the regular token ensures that the timestamp order on messages originated on a ring obeys the causal order for the topology. When a processor receives a message or the token, it sets *my_timestamp* to the larger of *my_timestamp* and the *timestamp* field in the message or token. For each new message it broadcasts, a processor increments *my_timestamp* and sets the *timestamp* field in the message to *my_timestamp*. Before forwarding the token, a processor sets the *timestamp* field in the token to *my_timestamp*. The *timestamp* field in the token ensures strictly increasing timestamps for messages generated on the ring. If any of the above actions increments *my_timestamp* to a value greater than or equal to *my_stable_timestamp* plus *timestamp_interval*, then the processor sets *my_stable_timestamp* to *my_timestamp*, and writes *my_stable_timestamp* to stable storage before broadcasting the message or forwarding the token.

Each time a gateway forwards a message onto another ring, it sets *my_timestamp* to the larger of *my_timestamp* and the *timestamp* in the message. A forwarded or retransmitted message retains the timestamp it was given when it was originated. This ensures that the next new message broadcast by a gateway has a higher timestamp than any message previously forwarded by the gateway.

Each time a processor or gateway receives a forwarded message, it sets *my_future_ring_seq* to the larger of *my_future_ring_seq* and the *src_ring_id.seq* of the forwarded message. If *my_future_ring_seq* has changed, then it is written to stable storage before the message is forwarded. When a new ring (regular configuration) is being formed, the *ring_id.seq* of the new ring is four plus the largest of the *my_future_ring_seq* for any of the processors on the new ring. (The ring sequence of the *ring_id* of the transitional configuration is two plus the largest of the *my_future_ring_seq* for any of the processors on the new ring. One or three plus the largest of the *my_future_ring_seq* is the ring sequence number of a transitional configuration consisting of only the processor itself if such a configuration must be used to provide safe delivery of messages.) When the processor shifts to the Recover state and writes the *my_ring_id.seq* to stable storage, it also sets *my_future_ring_seq* to *my_ring_id.seq*. *My_future_ring_seq* en-

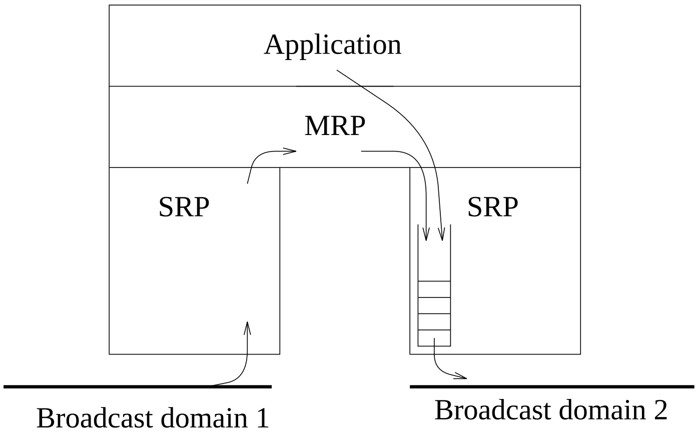


Figure 5.2: The messages broadcast on a directly attached ring by a gateway are messages that were generated by an application process executing at the gateway or messages that were delivered by the Totem Single-Ring Protocol (SRP) to the Totem Multiple-Ring Protocol (MRP) and were forwarded by a gateway.

sures that the causality relations are maintained when two messages originated on different rings have the same timestamp.

A processor or gateway executing the single-ring protocol stores messages received from the application in a FIFO buffer until it can broadcast them on the local ring. A gateway also places messages forwarded from the other single-ring protocol executing at the gateway in the buffer, as shown in Figure 5.2. When a message that was generated by the application at this processor is removed from this buffer, the message's timestamp is set to *my_timestamp* and *my_timestamp* is incremented. The message is then broadcast on the far-side ring.

Delivery of Messages in Agreed Order

The key to agreed ordering in the multiple-ring protocol is the fact that messages originated on a ring are forwarded through the network in order. Messages broadcast on a ring are delivered by the single-ring protocol to the multiple-ring protocol executing at a gateway or processor in sequence number order. Messages are forwarded onto a ring by a gateway in the order in which they are received from the single-ring protocol. A forwarded message is given a new sequence number, but retains its old timestamp, for each new ring onto which it

```
if recv_msgs of low entry in cand_msgs not empty then
    cur_msg := the low message in recv_msgs of the
        low entry in cand_msgs
    if cur_msg.type = agreed then
        deliver cur_msg
    else if cur_msg.type = safe then
        if for all i guarantee[i][source lan] >= cur_msg.timestamp then
            deliver cur_msg
        endif
    endif
    call deliver_msgs
endif
```

Figure 5.3: Deliver_msgs routine executed by processors and gateways to deliver a message.

is forwarded. On a new ring, messages are delivered to the multiple-ring protocol executing at a processor in sequence number order for that ring. Messages are then delivered to the application by a processor executing the multiple-ring protocol in the order of (timestamp, src_ring_id, type, conf_id).

Since the messages generated on any one ring are forwarded in order through the network, each gateway and processor can record a *max_timestamp* in *ring_table* for each ring; all messages from that ring with lower timestamps must already have been received. If a gateway receives a regular message with a timestamp less than the *max_timestamp* of the source ring, it discards the message as a redundant message. This mechanism allows a processor to identify redundant copies of messages forwarded by multiple gateways.

To deliver messages in agreed order, a processor first determines the lowest entry in *cand_msgs*. Messages with the same timestamp are ordered by *src_ring_id* and message type. If the lowest entry in *cand_msgs* corresponds to a message for which agreed delivery was requested, the message is delivered. If the *recv_msgs* list corresponding to the lowest entry in *cand_msgs* is empty, no further messages can be ordered until a message from that ring is received, because the next message from that ring may have a lower timestamp than the messages that have been received from the other rings. A processor can deliver

```
for all i do guarantee[msg.source][i] :=
        MAX( guarantee[msg.source][i], msg.guar_vector[i] )
endfor
```

Figure 5.4: Algorithm executed by processors and gateways on receipt of a Guarantee Vector message.

a message in agreed order only after it has delivered all other messages with lower timestamps.

Delivery of messages in agreed order across a network requires a processor to delay delivery of a message until all preceding messages in the total order have been delivered. There may be significant delays in forwarding messages through the network, but such delays are unavoidable.

Delivery of Messages in Safe Order

Delivery of a message in safe order requires information about whether the message has been received by all of the other processors in the network. A message to be delivered in safe order is originated with a request for safe delivery in its header. When a processor executing the single-ring protocol delivers a message in safe order, all of the other processors on that local ring must have received the message.

A processor executing the multiple-ring protocol uses *my_guar_vector* to record, for each directly attached ring, the messages that have been received from the single-ring protocol. A component of *my_guar_vector* corresponding to a particular ring is greater than or equal to the timestamp of a safe message only if that message is safe on the ring (has been received by every processor on the ring).

Gateways periodically create and broadcast Guarantee Vector messages. When a processor executing the multiple-ring protocol receives a Guarantee Vector message, it compares the *guar_vector* in the message with the appropriate row of its local *guarantee* array and changes a component of the row to the corresponding *guar_vector* component if the vector component contains a higher timestamp. The pseudocode executed by a processor or gateway on receipt of a Guarantee Vector message is given in Figure 5.4.

To deliver a message in safe order, a processor executing the multiple-ring protocol must wait until all entries in the column of the *guarantee* array, corresponding to the ring on which the message was generated, contain timestamps greater than or equal to the timestamp of the message. This guarantees that the message has been received by each processor in the network, and will be delivered by that processor unless it fails. The array is checked each time a message for which safe delivery was requested is the lowest entry in *cand_msgs*. Gathering the additional knowledge required for delivery of a message in safe order may delay delivery of messages with higher timestamps.

Example

The message ordering mechanisms of the multiple-ring protocol provide consistent agreed and safe message ordering across an entire network. A processor delivers a message in agreed order only after it has delivered all messages that precede it in the total order. As an example, consider the network shown in Figure 5.5, where the rings are represented by circles and the processors and gateways by squares. A processor p on ring A is ordering messages from rings A, B and C. If processor p has the data structures shown, p can deliver the message with timestamp 7 from ring A, the message with timestamp 8 from ring C, the message with timestamp 9 from ring B, and the messages with timestamp 10 from rings B and C in agreed order.

After these messages have been delivered, the *min_timestamp* and *max_timestamp* at processor p for ring C will be set to 11 until new messages have been received from C. The lowest entry in *cand_msgs* is the entry for ring C with timestamp 11. The undelivered message with lowest timestamp is now the message from ring B with timestamp 13, but no further messages can be delivered until the next message from ring C is received. Otherwise, there may be a message from C with a timestamp 12 that has not yet been received.

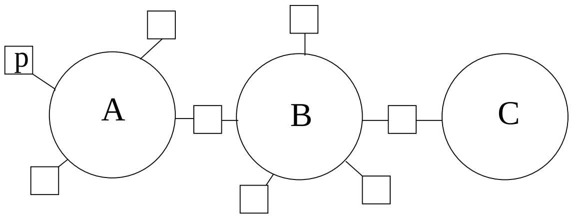
If the message received from ring A with timestamp 7 contains a request for safe delivery, then processor p can deliver that message as safe since the guarantee array column for A has all entries at least equal to 7 which indicates

Data structures at processor p

A: Recv_msgs = 7,14,15,26,29
Max_timestamp = 29
Min_timestamp = 7

B: Recv_msgs = 9,10,13,15
Max_timestamp = 17
Min_timestamp = 9

C: Recv_msgs = 8,10
Max_timestamp = 11
Min_timestamp = 8



	A	B	C
A	29	17	11
B	14	15	10
C	7	9	10

guarantee array at p

Figure 5.5: An example with rings A, B and C indicated by circles. The processors and gateways are drawn as squares. The data structures at processor p are also shown. A row of the guarantee array corresponds to the guarantee vector received from a gateway on the ring.

that the message is safe on all of the rings in the current topology. The same is true for the message from ring C with timestamp 8 and the message from ring B with timestamp 9. The message from ring B with timestamp 10 cannot, however, yet be delivered as safe since the guarantee vector from C reports receipt of messages from ring B only up to timestamp 9.

5.2 The Topology Maintenance Algorithm

The message ordering algorithm described above depends on knowledge of the network topology. If messages are originated on a ring of which a processor p is unaware, processor p must be informed of the new ring and must wait for such messages. Otherwise, p will prematurely deliver messages with higher timestamps. Similarly, if a ring becomes inaccessible and processor p is not informed, p must wait for a message from that ring and message ordering must stop until p deletes the ring.

It is very important that a topology change has a consistent effect throughout the set of processors that were previously able to, and can still, communicate with each other. Even though the various processors learn of the topology change at different times, they must agree on the same logical time for the change, and must agree on the sets of messages to be delivered before and after the topology change. Only with care is it possible to maintain consistent network-wide message delivery.

In addition, topology information must be exchanged between merged components of the network to ensure that the processors and gateways proceed with a consistent view of the network after the components have merged. This exchange of topology information is handled by the gateways through an additional round of message passing.

The Data Structures

The data structures and message types given below facilitate the maintenance of topology information.

Local Data Structures

Each processor and gateway maintains the local data structure:

- *neighbors*: The neighboring gateways on each of the rings to which this processor is directly attached. Each neighbor has associated with it a *begin_timestamp*, which is the timestamp of the Configuration Change message that first identified the gateway as a neighbor.

Each gateway also maintains the local data structure:

- *topology*: This gateway's view of the current topology of the network. *Topology* is maintained as a graph with each ring represented as a node and each gateway as an edge. Unreachable rings and gateways are not included in *topology*. Each edge has a timestamp associated with it indicating when it will be deleted from the graph based on the Configuration Change messages the gateway has received. This timestamp is initialized to -1 (infinity) when an edge is first added to *topology*.

The topology identifier is the lexicographically ordered list of ring identifiers of the rings that comprise the topology. Since ring identifiers are unique, topology identifiers are also unique.

Message Types

Configuration Change Message

A Configuration Change message is generated by the single-ring protocol to signal a change in the membership of a ring. The Configuration Change message informs processors and gateways of the existence of a new ring, and is forwarded by the gateway that generated it after the gateway has forwarded all of the messages from the old ring. The Configuration Change message is delivered by both the single-ring and multiple-ring protocols as an agreed message, and contains the following fields:

- *timestamp*: The timestamp in the Commit token on its second rotation in the single-ring membership algorithm, if the Configuration Change message initiates a regular configuration, and the highest timestamp of a message delivered in the preceding regular configuration if the Configuration Change message initiates a transitional configuration
- *src_ring_id*: The identifier of the new ring (regular or transitional).
- *type*: Configuration Change.
- *conf_id*: The identifier of the old ring (regular or transitional).
- *memb_list*: A list of the processor identifiers of the membership of the new configuration.
- *gateways*: A vector of booleans containing a position for each processor in *memb_list*. A vector component contains a one if the associated processor is a gateway, and a zero otherwise. This information is gathered in the Commit token of the single-ring membership algorithm.
- *gateway_ids*: A list containing the *gway_id* of each gateway on the new ring. This information is gathered in the Commit token of the single-ring membership algorithm.
- *guar_vector*: The guarantee vector at the processor or gateway that generated the Configuration Change message for the ring that experienced the configuration change.

The first four fields constitute the identifier of the Configuration Change message.

Network Topology Message

A Network Topology message is sent by a gateway on a directly attached ring that experienced a configuration change when a Configuration Change message is delivered at the gateway. The Network Topology message is not delivered to the application or forwarded by the multiple-ring protocol; it informs the other gateways and processors on the ring of the part of the network connected to the ring by this gateway. The Network Topology message contains the following fields:

- *timestamp*: The timestamp of the associated Configuration Change message.
- *src_sender_id*: The processor identifier on this ring of the gateway that originated the message.
- *gateway_id*: The identifier of the gateway that originated the message.
- *topology*: The gateway's current view of the network topology outside the ring that experienced the configuration change.

Note that a gateway deterministically chooses one of two ring identifiers as its *gateway_id*, and that its identifier *src_sender_id* on the ring that experienced the configuration change is not necessarily the same as *gateway_id*.

Topology Change Message

A Topology Change message is sent by a gateway to notify the other gateways and processors in the network of a change in the topology due to a configuration change. The Topology Change message is forwarded and delivered in order along with the regular messages. A Topology Change message is also created and delivered locally by a processor (not a gateway) when it receives a Configuration Change message for a directly attached ring. The local view of the topology is updated when the Topology Change message is delivered to the application. A Topology Change message is sent with a request for agreed delivery and contains the following fields:

- *timestamp*: The timestamp of the corresponding Configuration Change message if the Topology Change message adds one or more rings. Otherwise, it is the max_timestamp of the ring to be deleted (which, in the case that no message has been received from the ring, will be the timestamp of the Topology Change message that introduced the ring).

- *src_ring_id*: The *src_ring_id* of the corresponding Configuration Change message, or the *ring_id* of the ring to be deleted if the Topology Change message contains a ring deletion only and a message has been received from that ring, or the *src_ring_id* of the preceding Topology Change message if the topology change consists of a ring deletion only and no message has been received from the ring to be deleted.

- *type*: Topology Change, or Topology Change None if the topology change consists of a ring deletion only and no message has been received from the ring to be deleted.

- *conf_id*: the *conf_id* of the corresponding Configuration Change message, or the *ring_id* of the ring to be deleted if the topology change consists of a ring deletion only and no message was received from the ring to be deleted.

- *new_rings*: The identifiers of added rings, if any.

- *del_rings*: The identifiers of deleted rings, if any.

- *new_gateways*: A list of the gateways added to the topology, if any.

The identifier of the Topology Change message consists of the first four fields above.

Messages are delivered by a processor executing the multiple-ring protocol in the global total order defined by the lexicographical order on the set of ordered 4-tuples (*timestamp*, *src_ring_id*, *type*, *conf_id*). The timestamps are arranged in increasing order, the ring identifiers (*ring_seq*, *rep_id*) are lexicographically ordered, and the message types are ordered by the relation: Regular < Configuration Change < Topology Change < Topology Change None.

The Events of the Topology Maintenance Algorithm

There are five topology events, namely:

- Receipt of a Configuration Change message. The Configuration Change message is created by the single-ring membership algorithm when a new ring is formed. This message is not broadcast on the local ring but is generated locally and is propagated by the multiple-ring protocol through the rest of the network. On receipt of a Configuration Change message, a data structure for the new ring is added to the *ring_table*. If a gateway receives a Configuration Change message that indicates that a ring has become disconnected but no Configuration Change message is pending for the ring, the gateway sends a Topology Change message deleting the ring.

- Delivery of a Configuration Change message. A gateway directly attached to a ring that experienced a configuration change exchanges topology information with the other gateways and processors on that ring by sending a Network Topology message on that ring. The Network Topology message describes the component of the network connected to the ring by the gateway, based on its current topology and on Configuration Change messages. Before sending a Network Topology message, a gateway waits until the Configuration Change message is the next message to be delivered, to ensure that the local topology information is up-to-date.

- Receipt of a Network Topology message. When a gateway has received Network Topology messages from all of the gateways on the directly attached ring that experienced the configuration change, it sends a Topology Change message. The Topology Change message serves to inform the other processors and gateways in the network of the change in the network topology based on the gateway's local topology information.

- Receipt of a Topology Change message. When a processor or gateway receives a Topology Change message, it accepts the message and adds it to *recv_msgs* for the *src_ring_id*, unless it has already placed a Topology Change message with the same identifier in *recv_msgs*, in which case it discards the Topology Change message. If a gateway accepts the Topology Change message, it forwards it to the rest of the network.

- Delivery of a Topology Change message. The local view of the network topology is updated when the Topology Change message is delivered. The Topology Change messages are delivered in order along with the regular messages and, thus, are delivered by the processors and gateways in a consistent total order. Consequently, the gateways update their topology information in the same total order at all sites.

Handling a Single Configuration Change

First, we describe the steps taken by a processor to handle a single Configuration Change message without considering further topology changes during execution of the multiple-ring membership algorithm. Receipt of a Configuration Change message generated by the single-ring protocol signals a topology change. Since the gateways maintain the current view of the network topology, they are responsible for determination of the changes to the topology caused by the configuration change and dissemination of this information to the other gateways and processors in the network. A gateway determines the new topology by exchanging topology information with the other gateways on the ring and combining the information to determine the new topology.

On receipt of a Configuration Change message for a directly attached ring, a gateway takes the following steps:

1. Copy the current old ring *my_guar_vector* into the Configuration Change message *guar_vector*. Forward the Configuration Change message.

2. Add a data structure for the new ring to the *ring_table*. This data structure provides a location to store messages originated on the new ring, including this Configuration Change message.

3. Add a row for the new ring to the *guarantee* array.

4. Buffer messages received from new gateways on the ring until a Topology Change message adding the new ring has been ordered.

5. Until the Configuration Change message is the lowest entry in *cand_msgs*, deliver messages and, if *recv_msgs* of the ring with the lowest entry in *cand_msgs* is empty, then execute step 6. If the lowest entry in *cand_msgs* corresponds to a safe message that has not been guaranteed by one of the rings, then proceed to step 7. If the Configuration Change message is the lowest entry in *cand_msgs*, go to step 8.

6. Determine if the ring with the lowest entry in *cand_msgs* has become disconnected. If so, create and broadcast a Topology Change message indicating the deletion of the ring. Otherwise, wait for a message from the ring. Return to step 5.

7. Determine if the rings, that have not guaranteed the message, have become disconnected. If so, create and broadcast a Transitional Topology Change message (defined later) indicating the topology changes required to deliver the message. Otherwise, wait for a guarantee message from the ring. Return to step 5.

8. Construct a Network Topology message and broadcast it on the new ring.

9. Gather Network Topology messages from the other gateways on the new ring.

10. Combine Network Topology messages to determine changes to the current topology and broadcast a Topology Change message indicating changes to the local topology.

11. Update *topology* and *ring_table*.

12. Deliver the Configuration Change message and the associated Topology Change message.

On receipt of a Configuration Change message for a directly attached ring, a processor that is not a gateway takes the following steps:

1. If there is no gateway that transitioned from the old ring to the new ring, create Topology Change messages deleting the data structures for each ring from the *ring_table*, as of the highest entry in *recv_msgs* for the ring.

2. Steps 2-4 above.

3. Deliver messages until the Configuration Change message is the lowest entry in *cand_msgs*.

4. Combine Network Topology messages to determine the rings in the current topology and create a Topology Change message indicating ring additions and deletions.

5. Update *ring_table*.

6. Deliver the Configuration Change message and the associated Topology Change message.

On receipt of a Configuration Change message for a ring that is not directly attached, a processor takes the following steps:

1. Add a data structure for the new ring to the *ring_table*.

2. Add a row for the new ring to the *guarantee* array.

3. Continue delivering messages until the Configuration Change message is the lowest entry in *cand_msgs* and the accompanying Topology Change message has been received.

4. Deliver the Configuration Change message and the Topology Change message.

The processors and gateways directly attached to a ring are responsible for determining topology change information associated with a configuration change. The gateways are also responsible for disseminating the topology change information associated with the configuration change. Pseudocode for handling the messages associated with a configuration change is given in Figures 5.6, 5.7, 5.8, 5.9 and 5.10. The actions taken by the processors and gateways when a multiple-ring protocol topology event occurs are described below in greater detail.

```
if new ring data structure already exists in ring_table then
    discard message
    return
endif
if amgateway then
    forward Configuration Change message
endif
add new ring data structure
    recv_msgs := empty list
    max_timestamp := msg.timestamp
    min_timestamp := msg.timestamp
add message to new ring recv_msgs list
add row for new ring to guarantee array
if msg.src_ring_id = directly connected ring ring_id then
    update neighbor gateways list for new ring
    mark all new gateways as starting at Configuration Change msg.timestamp
    copy my_guar_vector for old ring into msg.guar_vector
    if amgateway then
        for each gway_id on new ring do
            if gway_id is in current topology then
                mark gateway for deletion at msg.timestamp
            endif
        endfor
    endif
endif
Update guarantee array row for old ring with msg.guar_vector
```

Figure 5.6: Algorithm executed by a processor or a gateway on receipt of a Configuration Change message.

Receipt of a Configuration Change Message

A Configuration Change message serves to inform the processors and gateways in the network of the new ring identifier and to flush messages from the old ring. The Configuration Change message precedes all messages forwarded from the new ring. In essence, the Configuration Change message places a marker in the message order for the topology change.

If the Configuration Change message is for a directly attached ring, then the processor (or gateway) copies its *my_guar_vector* for the *old_ring_id* into the *guar_vector* field in the Configuration Change message. On receipt of a Configuration Change message from any ring, a processor adds a data structure for the new ring to the *ring_table* with a *min_timestamp* and *max_timestamp* equal to the timestamp of the Configuration Change message. The processor then places the Configuration Change message in *recv_msgs* for the new ring.

The processor also advances the *max_timestamp* for the *old_ring_id* to the timestamp of the Configuration Change message. The processor adds a row to the *guarantee* array for the new ring and sets all entries in the row equal to the timestamp of the Configuration Change message. It also updates the row of the *guarantee* array associated with the old ring using *guar_vector* and updates *neighbors* according to the *gway_ids* in the Configuration Change message.

A gateway also updates the edges in its local *topology*. For each gateway identifier in *gway_ids*, the gateway marks the current *topology* edge for that gateway with the timestamp of the Configuration Change message. The edge is known to connect two different rings as of the timestamp of the Configuration Change message. The timestamps on edges are used to determine the rings to delete from the topology to allow delivery of the Configuration Change message.

The Configuration Change message is not used to add the new ring to, or to delete the old ring from, the gateway *topology*. To maintain extended virtual synchrony, the topology change needs to occur at the same logical time at all gateways and processors that experience the change. To accomplish this, a gateway delays updating the topology until the Configuration Change message is the lowest entry in *cand_msgs*.

Since messages may be delayed by the forwarding operation, the *max_timestamp* for a ring may be lower than the timestamp of the Configuration Change message. This will prevent the Configuration Change message from being delivered until the ring is deleted.

A gateway can send a Topology Change message to delete a ring when it has received a Configuration Change message that deletes the final connection to the ring and all of the messages in *recv_msgs* for that ring have been delivered. The ring can be deleted because messages are forwarded in order, and all messages that will be forwarded from the ring were forwarded ahead of the Configuration Change message that indicated the disconnection of the ring.

A processor (not a gateway), that receives a Configuration Change message for a directly attached ring indicating that there are no gateways that were on both the old ring and new ring, creates and delivers locally Topology Change messages to delete all rings in the *ring_table*, except for the old ring, before adding the new ring to the *ring_table*.

A Topology Change message to delete a ring has the same *src_ring_id* as that of the ring to be deleted, a *timestamp* equal to the *max_timestamp* for that ring, and contents indicating that the ring is to be deleted. The pseudocode executed by a processor on receipt of a Configuration Change message is given in Figure 5.6.

Delivery of a Configuration Change Message

When a Configuration Change message is the next message to be delivered (the lowest entry in *cand_msgs*), each gateway on the ring that experienced the configuration change sends a Network Topology message on that ring. A gateway waits to send the Network Topology message until the Configuration Change message is the next message to be delivered to ensure that its local *topology* has been updated to the timestamp of the Configuration Change message. The Network Topology messages are particularly necessary when there are gateways or processors that are added to the ring. They serve to inform the gateways and processors on the ring of the current topology connected to the ring by each gateway.

Before building the Network Topology message, the gateway deletes the gateway identifiers listed in *gway_ids* (including its own identifier) from *topology*. The gateways are deleted from *topology* because they are no longer connected

```
if msg.src_ring_id ≠ directly connected ring ring_id then
    wait for Topology Change message
else
    if amgateway then
        for each gway_id on new ring do
            if gway_id is in topology then
                delete gateway from topology
            endif
        endfor
        send Network Topology message
    endif
    wait for Network Topology messages from all gateways on new ring
    combine Network Topology messages to determine new topology
    create Topology Change message
    if amgateway then
        send Topology Change message
    endif
endif
deliver Configuration Change and Topology Change messages
```

Figure 5.7: Algorithm executed by a processor or a gateway when a Configuration Change message is the lowest entry in *cand_msgs*.

to their old ring and, if they were left in *topology*, they would cause problems in determining the currently connected topology to report in the Network Topology message. The gateway determines the connected topology and sends this information in a Network Topology message on the ring that experienced the configuration change.

A processor or gateway delays delivery of a Configuration Change message until it has received an associated Topology Change message with the same timestamp and source ring, or Network Topology messages from all of the gateways on the ring with the same timestamp as the Configuration Change message, in which case it generates a Topology Change message. (Note that we are assuming that no further configuration changes occur.) The pseudocode executed by a processor to deliver a Configuration Change message is given in Figure 5.7.

```
store as neighbor topology
if have Network Topology msgs from all gateways on new ring then
    combine neighbor topologies to determine rings in network
    add structures for new rings
    add new rings to Topology Change message
    list deleted rings in Topology Change message del_rings
    discard neighbor topologies
    if amgateway then
        list added gateways in Topology Change message
        send Topology Change message
    endif
    add Topology Change message to recv_msgs of new ring
endif
```

Figure 5.8: Algorithm executed on receipt of a Network Topology message by a processor or a gateway.

Receipt of a Network Topology Message

The gateways and processors on a ring that experienced a configuration change are responsible for determining the topology changes associated with that configuration change. To accomplish this, each gateway and processor gathers the Network Topology messages from the gateways listed in *gway_ids* of the Configuration Change message.

When a gateway has received Network Topology messages from all of the gateways on the new ring, it merges the *topology* in the messages into its own *topology*. For each ring or gateway added to *topology*, it records that ring or gateway in the fields *new_rings* or *new_gateways* of a Topology Change message. The gateway also records any disconnected rings in the field *del_rings* of the Topology Change message. When all ring and gateway additions and deletions are complete, the gateway sends a Topology Change message indicating the changes with *timestamp* equal to the timestamp of the Configuration Change message (and, therefore, also of the Network Topology messages) and *src_ring_id* equal to the new ring. The Topology Change message is sent on all directly attached rings except the ring that experienced the configuration change.

When a processor has received Network Topology messages from all of the gateways on the new ring, the processor merges the *topology* in the messages

```
if msg.timestamp < min_timestamp for msg source ring or
        msg is already in recv_msgs for ring source then
    discard msg
    return
endif
if amgateway then
    forward msg
    for each gateway identifier in new_gateways do
        if gateway is in topology then timestamp := msg.timestamp
    endfor
endif
add to msg source ring recv_msgs list
add new_rings data structures to ring_table with
        max_timestamp := msg.timestamp and
        min_timestamp := msg.timestamp
add a row to guarantee for each ring in new_rings
source ring max_timestamp := msg.timestamp
```

Figure 5.9: Algorithm executed by a processor or a gateway on receipt of a Topology Change message.

to determine the rings in the network, including the new ring initiated by the Configuration Change message. The processor then generates a Topology Change message containing any added rings in *new_rings* and any deleted rings in *del_rings*. The Topology Change message once generated is added to *recv_msgs* for the new ring and is forwarded by the gateways. The pseudocode executed by a processor on receipt of a Network Topology message is given in Figure 5.8.

Receipt of a Topology Change Message

When a processor receives a Topology Change message, it accepts the message and adds it to *recv_msgs* for the *src_ring_id*, unless it has already placed a Topology Change message with the same identifier in *recv_msgs*, in which case it discards the Topology Change message. If a gateway accepts the Topology Change message, it forwards it to the rest of the network.

```
for each ring in del_rings do
    delete data structure for ring from ring_table
    if amgateway then
        delete ring and connected gateways from topology
    endif
    delete column and row for ring from guarantee array
    delete entry for ring from my_guar_vector
endfor
for each ring in new_rings do
    add row for ring to guarantee array
endfor
if amgateway then
    for each ring in new_rings do
        add ring to topology
    endfor
    for each gateway in new_gateways list do
        delete gateway from topology
        add gateway as edge in topology
    endfor
endif
deliver Topology Change message
```

Figure 5.10: Algorithm executed by a processor or a gateway on delivery of a Topology Change message.

If a processor accepts the Topology Change message, it adds data structures for the previously unknown rings in the *new_rings* list of the Topology Change message to *ring_table*. Each ring is added to *ring_table* with an empty *recv_msgs* list and a *min_timestamp* and *max_timestamp* equal to the *timestamp* of the Topology Change message.

A gateway also marks the edges corresponding to *new_gateways* in *topology* with the timestamp of the Topology Change message indicating that those edges will be deleted from the topology at that timestamp. These new gateways will connect a different pair of rings after the Topology Change message is ordered. The pseudocode executed by a processor or gateway on receipt of a Topology Change message is given in Figure 5.9.

Delivery of a Topology Change Message

When a Topology Change message is the lowest entry in *cand_msgs* at a gateway, the gateway deletes the *new_gateways* from *topology* since they will now connect a different pair of rings. The gateway then adds the rings in *new_rings* and the gateways in *new_gateways* to *topology*. The gateway also deletes the rings in *del_rings* from *topology*. All gateways connected to a deleted ring are deleted with that ring.

The processor or gateway also deletes the components corresponding to the rings in *del_rings* from the *guarantee* array, *my_guar_vector*s and *ring_table*. A row for each ring in *new_rings* is added to the *guarantee* array; the timestamp for each entry in the row is initialized to the timestamp of the Topology Change message. When a processor has completed processing a Topology Change message, it delivers the message. The pseudocode executed by a processor to deliver a Topology Change message is given in Figure 5.10.

Message Ordering During a Topology Change

When a Configuration Change message for a directly connected ring is pending (received but not delivered), a processor discards a message (does not place it into *recv_msgs*, if the message has a timestamp less than the timestamp of the Configuration Change message and if it was forwarded onto the ring by a new gateway (a gateway that was not a member of the old ring). All messages received from a new gateway with timestamps greater than the timestamp of the Configuration Change message are buffered by the processors and gateways on the ring until after the Configuration Change message is delivered. These buffered messages may be from previously known rings and may follow a temporary interruption in the forwarding of messages. Once the Configuration Change message and its associated Topology Change message have been delivered, the buffered messages are added to *recv_msgs* and are forwarded by the gateways in order.

To deliver a message in safe order, a processor waits until it knows that all processors and gateways in the current topology have received the message. If a topology change has partitioned the network and a message that requested safe delivery is not known to have been received by all of the processors on a disconnected ring (because a Guarantee Vector message that guaranteed the message as safe had not been received), that ring must be deleted from the topology before the message can be delivered in safe order.

Since there is a delay in forwarding messages through the network, some messages forwarded onto a ring after a configuration change will have a lower timestamp than the timestamp of the Configuration Change message that disconnected the forwarding path. When the forwarded message is the low entry in *cand_msgs*, the topology still contains the ring disconnected by the configuration change. If a message requesting safe delivery was not guaranteed as safe on the old ring, each processor and gateway on the old ring proceeds to a new ring containing only itself to deliver the message. To accomplish this, the multiple-ring protocol generates additional configuration changes using two additional message types.

The Data Structures

The data structures and message types described below facilitate the removal of rings from the current topology to allow ordering of safe messages.

Local Data Structures

- *pending_SCC_buffer*: List of Transitional Configuration Change messages that have been received but not processed.

Message Types

Transitional Configuration Change Message

A Transitional Configuration Change message is created by a gateway when a message requesting safe delivery has not been guaranteed as safe by a directly attached ring that experienced a configuration change. The Transitional Configuration Change message informs the other processors and gateways in the network that the processor or gateway is proceeding to a ring containing only itself. The Transitional Configuration Change message is not delivered to the application and contains the following fields:

- *timestamp*: The timestamp of the message that requested safe delivery.
- *src_ring_id*: The identifier of the ring that did not guarantee the message as safe.
- *conf_id*: The identifier of the old ring in the pending Configuration Change message for the new ring.
- *temp_ring_id*: The identifier of the new ring. The new ring identifier consists of the identifier of this processor or gateway (as a representative) and a ring sequence number one greater than the sequence number of the *conf_id*.
- *next_ring_id*: The *src_ring_id* from the pending Configuration Change message for the ring.
- *gway_id*: The identifier of the gateway that originated the message.

The Transitional Configuration Change message does not contain a *memb_list* field since the membership of the new ring is the processor or gateway itself.

Transitional Topology Change Message

A Transitional Topology Change message is generated by each processor or gateway on the ring that did not guarantee the message as safe. The Transitional Topology Change message is forwarded by the gateways to notify the other processors and gateways in the network of the change in the topology and to allow delivery of a message requesting safe delivery. The Transitional Topology Change message is forwarded and delivered in order along with the other messages in the network. The local view of the topology is updated when the Transitional Topology Change message is delivered to the application. The Transitional Topology Change message is sent with a request for agreed delivery and contains the following fields:

- *timestamp*: The *timestamp* of the message that requested safe delivery.
- *src_ring_id*: The *src_ring_id* of the message that requested safe delivery.

- *type*: Transitional Topology Change.
- *conf_id*: The lowest ring identifier in *new_rings* determined by lexicographical order.
- *new_rings*: The identifiers of the added rings, *i.e.* the singleton rings containing the gateways.
- *del_rings*: The identifiers of the deleted rings.
- *new_gateways*: A list of the gateways added to the topology.

The identifier of the Transitional Topology Change message consists of the first four fields above.

The Transitional Topology Change messages are delivered by a processor or gateway executing the multiple-ring protocol in the global total order. The Transitional Topology Change message type is ordered in the message delivery order as follows: Transitional Topology Change < Regular.

The Algorithm

When a gateway determines that the low entry message in *cand_msgs* cannot be delivered as safe on a directly attached ring in the current topology, it generates a Transitional Configuration Change message. This message cannot be guaranteed as safe on the ring because it has been received from the single-ring protocol after a Configuration Change message for the ring, and the Configuration Change message had a timestamp greater than the timestamp of the message that requested safe delivery. This situation can only occur when messages are forwarded onto the ring, and is caused by a delay in the forwarding of messages. The Transitional Configuration Change message generated by a gateway is forwarded throughout the connected component of the network. A Transitional Configuration Change message generated by a processor (not a gateway) is only delivered locally. The pseudocode executed by a processor or gateway to generate a Transitional Configuration Change message is given in Figure 5.11.

```
if low message in cand_msgs requested safe delivery and
        a directly attached ring did not guarantee message and
        directly attached ring has pending Configuration Change message then
    generate Transitional Configuration Change message to reduce
            directly attached ring to singleton
    send Transitional Configuration Change message on all other
            directly attached rings
    add Transitional Configuration Change message to pending_SCC_buffer
endif
```

Figure 5.11: Algorithm executed by a processor or a gateway when the low message in *cand_msgs* requested safe delivery and cannot be guaranteed as safe by a locally attached ring.

On receipt of a Transitional Configuration Change message, a processor or gateway adds an entry to its *ring_table* for *temp_ring_id* with *min_timestamp* and *max_timestamp* equal to the timestamp of the Configuration Change message pending for the *next_ring_id*.

On receipt of a Transitional Configuration Change message, a gateway marks the edge corresponding to *gway_id* with the timestamp of the Transitional Configuration Change message. If the message corresponding to the low entry in *cand_msgs* contains a request for safe delivery and the message has been guaranteed as safe in the topology that remains accessible, then the gateway generates a Transitional Topology Change message listing in *new_rings* the *temp_ring_id* from each of the Transitional Configuration Change messages received. The identifiers of all of the rings that have become disconnected due to the configuration changes listed in the Transitional Configuration Change messages are listed in *del_rings*. The identifiers of the gateways that sent the Transitional Configuration Change messages are listed in *new_gateways*. These gateways are each directly connected to a new singleton ring in *new_rings*. The pseudocode executed by a processor or gateway on receipt of a Transitional Configuration Change message is given in Figure 5.12.

A processor (not a gateway) on the ring that incurred the configuration change can generate the Transitional Topology Change message directly since it proceeds to a singleton ring containing only itself. It places the identifier of its new singleton ring in *new_rings* and the identifiers of all rings in its *ring_table*

```
add Transitional Configuration Change message to pending_SCC_buffer
add temp_ring_id to ring_table
update conf_id of Configuration Change pending for next_ring_id
if amgateway then
    mark gateway in topology with message.timestamp
    if low message in cand_msgs is safe in
            remaining connected network then
        construct Transitional Topology Change message
        for each message in pending_SCC_buffer do
            add temp_ring_id to new_rings
            add conf_id to del_rings
            add gway_id to new_gateways
            discard Transitional Configuration Change message
        endfor
        send Transitional Topology Change message
        add Transitional Topology Change message to recv_msgs of src_ring_id
    endif
endif
```

Figure 5.12: Algorithm executed by a processor or a gateway on receipt of a Transitional Configuration Change message.

(except the *new_ring_id* from the pending Configuration Change message) in *del_rings*. It then delivers the Transitional Topology Change message locally and does not broadcast it; this suffices because there are no other processors or gateways on the ring with identifier *temp_ring_id*.

When the Transitional Topology Change message is the low entry in *cand_msgs*, a gateway deletes the rings in *del_rings* and adds the rings in *new_rings* to the topology. It also adds the gateways in *new_gateways*. A processor or gateway delivers a Transitional Topology Change message to inform the application of the changes to the topology. If there are messages in *recv_packets* from the deleted rings, these messages can still be delivered if the requirements for agreed or safe delivery in the reduced topology are met. A processor on a detached ring can also deliver the message as agreed or safe on that ring. The pseudocode executed by a processor or gateway when it delivers a Transitional Topology Change message is given in Figure 5.13.

```
for each ring in del_rings do
    if recv_msgs for ring is empty then
        delete ring from ring_table
    else
        mark ring for deletion
    endif
    if amgateway then
        delete ring from topology
    endif
endfor
add each ring in new_rings to ring_table
if amgateway then
    add each ring in new_rings to topology
    add each gateway in new_gateways to topology
endif
```

Figure 5.13: Algorithm executed by a processor or a gateway when the low message in *cand_msgs* is a Transitional Topology Change message.

Example

Returning to the example in Figure 5.5, let's examine what happens if ring B partitions into B' and B'' as shown in Figure 5.14. The Configuration Change message delivered by the processors on ring B' has timestamp 25. The data structures at gateway q, after the Configuration Change message is received, are also shown in Figure 5.14.

When gateway q receives the Configuration Change message, it adds the message to *recv_msgs* for ring B', increases the *max_timestamp* for B to 25, copies *my_guar_vector* into the Configuration Change message and forwards it. Gateway q also sets its own timestamp in *topology* to 25. The Configuration Change message cannot be delivered yet, because messages beyond timestamp 11 have not been received from C. Since the lowest entry in *cand_msgs* corresponds to ring C and *recv_msgs* for C is empty, gateway q investigates the connected component of *topology*. In determining the connected component, q considers all gateways with timestamps not equal to -1 to be deleted. Since there are no other connections to ring B, q determines that C has become disconnected and sends a Topology Change message deleting C at timestamp 11. When this Topology Change message is ordered, C is deleted by the processors on

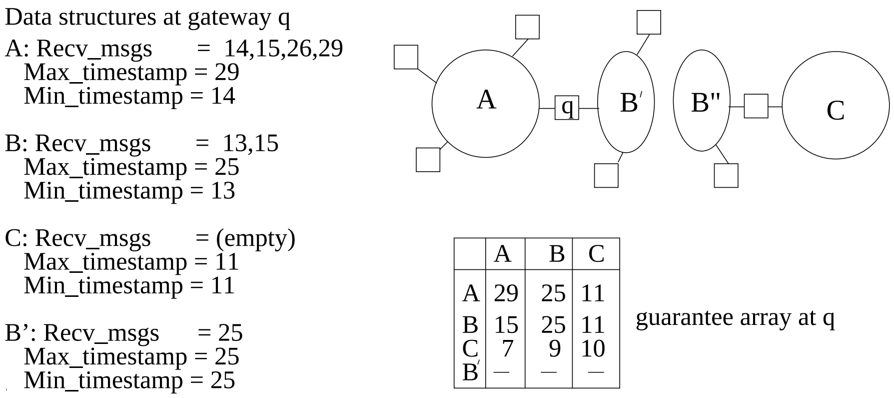


Figure 5.14: An example with ring B partitioned into B' and B''. The rings are indicated by circles and the processors are drawn as squares. The data structures at gateway q are also shown.

rings A and B'. This allows those processors and q to order messages beyond timestamp 11, in particular the message from ring B with timestamp 13, the message from ring A with timestamp 14, and the messages from rings A and B with timestamp 15.

When the Configuration Change message is the lowest entry in *cand_msgs*, gateway q sends a Network Topology message on ring B' with a timestamp 25 and a *topology* consisting of ring A. Since q is the only gateway on the new ring, it does not wait for additional Network Topology messages and instead proceeds immediately to construct the Topology Change message. Gateway q sends a Topology Change message on A indicating the addition of ring B' and the deletion of ring B. When they receive the Configuration Change message, the processors on A add a data structure for ring B' to their *ring_table*. They delete the data structure for ring B from their *ring_table* when they deliver the Topology Change message.

When a processor on ring B' receives the Network Topology message from q, it has all of the Network Topology messages and determines that the current topology consists of A and the new ring B'. Each processor on the new ring B' creates a Topology Change message indicating the addition of ring B' and the deletion of ring B with *timestamp* 25 and *src_ring_id* that of B'. The processor then adds the Topology Change message to *recv_msgs* for ring B'. The processor deletes ring B when it delivers the Topology Change message.

Handling Multiple Concurrent Configuration Changes

The multiple-ring membership algorithm does not have any control over the order or timing of configuration changes, thus Topology Change messages and further Configuration Change messages may be received before the current Configuration Change message has been delivered. If a processor is waiting for a Network Topology message from a gateway and instead receives a Configuration Change message indicating that the gateway is no longer connected, the protocol must be able to proceed without that Network Topology message. We now describe the handling of Configuration Change messages that arrive before a pending Configuration Change message has been delivered. The receipt and delivery of Topology Change messages and the delivery of Configuration Change messages are unaffected by multiple concurrent topology changes.

Receipt of a Configuration Change Message

On receipt of a Configuration Change message, a processor takes the actions described in handling a single Configuration Change message. If the Configuration Change message is for a directly attached ring that already has a Configuration Change message pending, a processor also takes the actions as if a Network Topology message had just been received. The processor may now have received all the needed Network Topology messages for a pending Configuration Change message since some gateways may not be on the new ring of the most recent Configuration Change message.

Receipt of a Network Topology Message

On receipt of a Network Topology message, a processor checks whether it has all of the Network Topology messages for the pending Configuration Change message with the same timestamp. It must have received Network Topology messages from all neighboring gateways with *begin_timestamp* less than or equal to the timestamp of the associated Configuration Change message; subsequent Configuration Change messages may have reduced this set of neighbors. If it has received these messages, the processor takes the actions described for receipt of a Network Topology message when handling a single topology change.

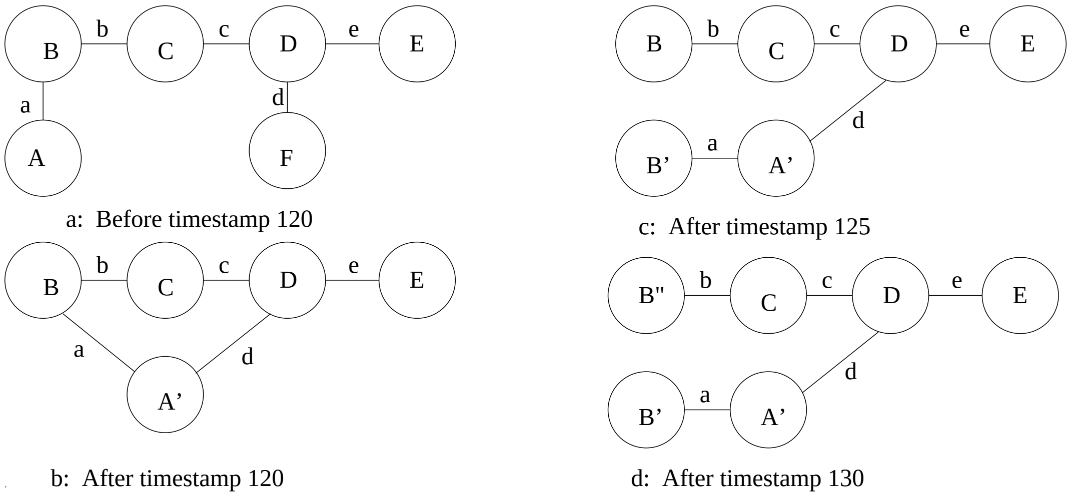


Figure 5.15: An example with rings A, B, C, D, E and F indicated by circles. The gateways are drawn as lines. At timestamp 120, rings A and F merge and become A'. At timestamp 125, ring B' is formed by a partition of B, and at timestamp 130 the rest of the processors from B form B''.

Message Ordering

If there are multiple Configuration Change messages pending for a single ring, a processor buffers messages forwarded by all new gateways. The messages are removed from the buffer when the associated Configuration Change message is delivered.

A processor also uses all of the pending Configuration Change messages to determine if the ring with the lowest entry in *cand_msgs* has become disconnected. If the lowest entry in *cand_msgs* requested safe delivery, the processor uses all pending Configuration Change messages to determine if a ring that did not guarantee the message has become disconnected. Otherwise, the ordering of messages proceeds as in the case of a single topology change.

Example

Consider the network in Figure 5.15a. The topology of the network is progressing through the topology changes shown. Initially, there are six rings. These rings progress through two topology changes.

The first topology change is the merging of rings A and F into the new ring A' (Figure 5.15b). The Configuration Change message received by the processors and gateways on A' reports the change and has timestamp 120. The second topology change is the partitioning of ring B into B' and B'' (Figures 5.15c and 5.15d). Two Configuration Change messages are delivered: one is received by the processors and gateways on B' and has timestamp 125, and the other is received by the processors and gateways on B'' and has timestamp 130. The gateways that receive the Configuration Change messages forward these messages to inform the other processors in the network of the configuration change.

The guarantee vectors at gateway a, after the Configuration Change messages for A' and B' have been received, are shown in Figure 5.16. The guarantee vectors for A' and B' are maintained by a starting immediately after the Configuration Change messages are received. Since neither of the Configuration Change messages has been delivered, they have not yet affected the topology information. The messages forwarded onto A' by gateway d with timestamps greater than 120 are buffered at a and are not forwarded or added to *recv_msgs* until the ordering timestamp reaches 120 and the Configuration Change message has been delivered. Messages forwarded by d onto A' with timestamps less than 120 are discarded by a.

At gateway a, *recv_msgs* for E is empty and *max_timestamp* for E is 85. Prior to the topology changes, gateway b forwarded E's messages to gateway a. After the first topology change, gateway d forwards messages from E to gateway a, but gateway a buffers these messages until after it has delivered the Configuration Change and Topology Change messages for A'. When gateway a receives the Configuration Change message for B', it knows that gateway b is no longer forwarding messages from E to gateway a.

Gateway a will never receive the messages from E between timestamps 85 and 120. Gateway a must delete E from its topology at timestamp 85 to avoid inconsistencies. To inform the rest of the network of the deletion, a transmits a Topology Change message with timestamp 85, *src_ring_id* equal to E, and contents stating that E should be deleted. This Topology Change message will be ordered by the processors on A' and B', but will be discarded by d because

	A	B	C	D	E	F
A	120	110	100	95	85	90
B	120	125	124	122	85	90
A’	120	125	124	122	120	120
B’	125	125	125	125	125	125

Figure 5.16: Some of the guarantee vectors at gateway a for rings A and B after receiving the Topology Change messages informing of the formation of rings A' and B' (Figure 10c).

it has a timestamp less than 120. Gateway d maintains a connection with E and does not need to delete it. When the ordering timestamp at a reaches 90, gateway a sends a Topology Change message deleting F.

Rings B, C and D are not deleted by gateway a at this time because its *recv_msgs* list for each of these rings has messages up through timestamp 120. The Configuration Change message at timestamp 120 adds a connection via gateway d through A', and messages from B, C and D are received at gateway a. The deletions of E and F at a are essential to allow the ordering of messages to progress at gateway a.

Gateway a waits until its ordering timestamp has reached 120 and then sends a Network Topology message on A' indicating that its current topology outside of A' is B, C, and D. When a receives the Network Topology message with timestamp 120 from d, a is informed of the new connection to B, C, D and E and adds E back into the topology of the network. Gateway a also sends a Topology Change message indicating that rings A' and E were added at timestamp 120. The message also indicates the deletion of A at timestamp 120. Upon receipt of the Topology Change message, the processors have all of the information they need to order messages beyond timestamp 120.

5.3 Performance

Flow Control

The single-ring protocol provides effective flow control within a single broadcast domain. It ensures that all processors and gateways on the ring have an opportunity to broadcast messages, and limits the maximum rate of transmission by any individual processor on the ring. However, a mechanism to create backpressure on the applications generating messages is required to avoid unlimited output buffering requirements within the single-ring protocol for messages waiting to be broadcast.

Messages are also queued in an input buffer for delivery to the application. This buffer can grow to unlimited size if the application is not keeping up with the message generation rate. Consequently, a backpressure mechanism is also required at the application interface.

Although the Totem protocol is defined as separate layers, several of the layers are normally compiled into a single process, but there may be multiple processes implementing Totem within a processor. For a processor to buffer messages between layers of the protocol within a process is pointless. Instead, we have implemented a process oriented flow-control mechanism. Each process maintains a variable to indicate if any of the interface queues has reached a predefined overflow point. The setting of this variable activates a backpressure mechanism which reduces the incoming traffic until the queue has reached a more reasonable size. This creates a network-wide flow-control mechanism, which is described below.

The Data Structures

Regular Token

The following fields are added to the regular token:

- *block*: A boolean indicating whether the network is congested, as explained below.
- *block_seq*: A sequence number indicating the current block sequence number for this ring.

Local Variables

Each process maintains the following variables:

- *site_block*: An integer indicating the number of message queues in this process that are full, as defined by *max_threshold* below.

- *site_block_seq*: The highest block sequence number known to this processor.

For each queue of messages waiting to be sent the process maintains the following variables:

- *max_threshold*: Number of messages allowed in the queue before this queue is in danger of overflowing.

- *min_threshold*: Number of messages in the queue for the queue so that it no longer in danger of overflowing.

- *set_site_block*: An enumerated type with one of three values that indicate whether this queue has reached the *max_threshold*, or has been asked by the receiving site to block, or neither.

For each single-ring protocol connection, the process maintains the following variable:

- *blocking_token*: Boolean indicating whether this site set the block field of the token.

The variables *site_block* and *site_seq* are initialized to zero, the *max_threshold* and *min_threshold* values for each queue are established using heuristics based on several factors including buffer and latency requirements. The *set_site_block* for each queue and *blocking_token* for each process executing the single-ring protocol are initialized to false.

The Algorithm

A process maintains a queue at each output interface, and adds a message to the queue when the message destination is not ready to receive it. For each queue, the process counts the number of messages in the queue. When the count reaches *max_threshold*, the process increments *site_block* and *site_seq*. The process also sets *set_site_block*. The pseudocode executed by a processor on overflow of a queue is given in Figure 5.17. When a queue that had previously reached *max_threshold* reaches *min_threshold*, the block can be removed; *site_block* is

```
if queue size >= max_threshold and set_site_block = FALSE then
    site_block++
    site_seq++
endif
set_site_block := SND_BLOCK
```

Figure 5.17: Algorithm executed by a processor or a gateway when the number of messages in a send queue reaches the maximum threshold.

```
If queue size <= min_threshold and set_site_block = SND_BLOCK then
    site_block--
    set_site_block := FALSE
endif
```

Figure 5.18: Algorithm executed by a processor or a gateway when the number of messages in a send queue reaches the minimum threshold after having reached the maximum threshold.

then decremented and *set_site_block* is unset. The pseudocode executed by a processor to unset *site_block* is given in Figure 5.18.

A process does not accept messages from users unless *site_block* equals zero. At the single-ring protocol interface, there is no benefit to the process refusing to accept messages. Instead, the process uses the token to propagate the site block and to reduce the traffic generated by other processors in the network. A single-bit *block* field is added to the token to propagate a processor's site block to the other processors on the ring. The *block_seq* field is added to the token to allow processors to recognize an out-of-date block.

When the token arrives and the site is blocked but the token is not blocked, if the token *block_seq* is less than *site_seq* then the processor sets the *block* field of the token and sets *blocking_token* to true to remember that this site blocked the token. If the token is blocked but the processor is not blocked and this processor set the token *block* field, then the processor unsets the token *block* field. Otherwise, if the token *block_seq* is greater than *site_seq* then some other processor wants the processors on the ring to block so this processor increments *site_block* and sets *set_site_block* to remember that it set the *site_block*.

```
if token.block = TRUE then
    if site_block = 0 then
        if blocking_token = TRUE then
            token.block := FALSE
            blocking_token := FALSE
        else if token.block_seq > site_seq then
            site_block++
            set_site_block := RCV_BLOCK
        endif
    endif
else
    if set_site_block = RCV_BLOCK then
        site_block--
        set_site_block := FALSE
    endif
    if site_seq > token.block_seq and site_block > 0
        blocking_token := TRUE
        token.block := TRUE
    endif
    if site_seq = token.block_seq and site_block > 0 and
            set_site_block = SND_BLOCK then
        site_seq++
        token.block := TRUE
        blocking_token := TRUE
    endif
endif
site_seq := MAX( site_seq, token.block_seq )
token.block_seq := site_seq
```

Figure 5.19: Algorithm executed by a process on receipt of the token.

Before forwarding the token, the processor sets *site_seq* to the maximum of *site_seq* and the token *block_seq* field. The processor then sets the token *block_seq* field to the value in *site_seq*.

A process is blocked if its *site_block* variable is greater than zero. A blocked process continues to send messages as allowed by the single-ring protocol. If the process is part of a gateway, it also continues to forward messages.

A process uses multiple values for *set_site_block* to remember what caused *site_block* to be set. The process sets *set_site_block* to RCV_BLOCK when it increments *site_block* because it received a token with the *block* field set. If the queue reaches *max_threshold*, the process sets *set_site_block* to SND_BLOCK, because even if the *site_block* was originally set by receipt of a blocked token, this site's send queue is now also blocking which overrides the token block. If the token is received with the *block* field unset and the site's *set_site_block* set to RCV_BLOCK, this site had previously received a token with the *block* field set which is now unset so the process decrements *site_block*. The pseudocode executed by a processor to handle flow control on receipt of the token is given in Figure 5.19.

The token and site blocking mechanisms provide a means of propagating information about a congested node throughout the network. The time to propagate the block operation to a particular processor on the ring is on average one token rotation, one half rotation waiting for the token and one half rotation before the token reaches the other processor. The use of the *site_block* variable provides immediate forwarding of the block operation through a gateway. Thus, a block or unblock operation will propagate through the network quickly. Although more sophisticated mechanisms of gradually decreasing or increasing network traffic to avoid congestion could be used, they would require additional space in the token and additional processing at each site.

Simulation

The single-ring protocol simulator has been extended to allow study and debugging of the Totem multiple-ring protocol [19]. The single-ring protocol simulator is designed to simulate a ring with an arbitrary number of processors using one physical host. To simulate multiple rings, this simulator is distributed across multiple physical hosts with each host simulating a ring. The connections between rings are provided by the gateways. Each gateway is split into two parts connected by a TCP socket. The single-ring protocol executes on the physical host that simulates the ring. Most of the gateway code executes on one of the two physical hosts and the other side executes a simple filtering process; the two sides communicate over a TCP socket. The distributed simulator was developed for the multiple-ring protocol to allow study of larger systems than are physically available.

Implementation

The Totem multiple-ring protocol has been implemented using the C programming language on a network of Sun 4/IPC workstations connected by an Ethernet. To allow study of the multiple-ring protocol, the single-ring protocol was modified to allow specification of a network number that determines the socket to be used for broadcasts. This allows multiple rings to be executed on a single Ethernet. However, the design of the single-ring protocol precludes running more than a few rings on a single Ethernet since the protocol is built on an assumption that the message loss rate is relatively low.

Measurements of the multiple-ring protocol were made for two rings, operating over separate Ethernets, connected by a gateway with two processors other than the gateway per ring. A Sun Sparcstation 20 was used as the gateway and four Sun 4/IPCs were used as the processors.

The single-ring protocol with three processors (two IPCs and one Sparcstation 20) on a ring sending 1024 byte messages achieves a throughput of 768 messages ordered per second. If the multiple-ring protocol is run as well, the throughput drops to 636 messages ordered per second. This translates to an additional 0.27 milliseconds per message for the multiple-ring protocol. Two rings running with the gateway forwarding messages between them acheived a throughput of 631 messages ordered per second.

The above tests had the single-ring flow control parameters set to allow the gateway to send twice as many messages per token rotation as each of the individual processors. This ensured that the multiple-ring protocol flow-control was seldom invoked. To study the effectiveness of the flow control of the multiple-ring protocol, we varied the proportion of messages sent by the processors and the gateway on each ring. Setting the proportion of messages sent by the gateway to 1.5 times that for each individual processor (flow-control was invoked regularly) resulted in a throughput degradation of less than 2 messages per second.

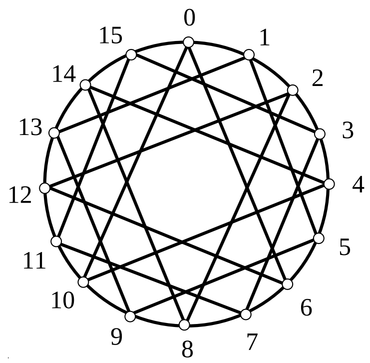


Figure 5.20: A robust sixteen node graph. Even nodes are connected at hops of one and six, and odd nodes are connected at hops of one and four.

5.4 Network Configuration

Reliable ordered delivery of messages can only be achieved if the processors are able to communicate with each other. Many wide-area networks are deliberately designed to provide alternative routes between nodes so that failure of a single link does not preclude further operation. Typical networks provide a small number of alternative routes, but in many applications the additional delay involved in using an alternative route may be unacceptable. Thus, the network design must be carefully considered if reliable ordered delivery protocols are to be effective.

If we represent the network as a graph where each local-area network is a node and each gateway is an edge, then network connectivity can be analyzed as a graph partitioning problem. Alternative path lengths can be determined from the minimum depth spanning trees of a graph after failures occur. The height of the minumum depth spanning tree rooted at node n is the length of the minumum route to the most distant node from n in the network.

We have built a simulator to evaluate network designs and have investigated typical network graphs that contain 50 nodes and are at least bi-connected, each node having maximum degree 4 (i.e. 50 local-area networks with at most 4 gateways on each local-area network). Two types of network designs were investigated: random and robust. The random graphs were constructed by adding random edges to the graph. An edge was not added if it made the degree of a node greater than four or if it created a self-loop or parallel edge.

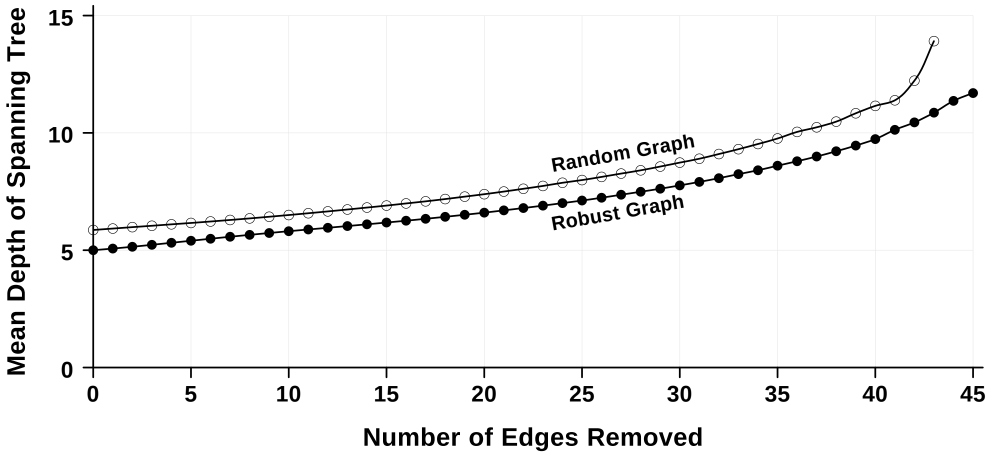


Figure 5.21: The effect on the spanning tree height of deletion of edges in a graph. The graphs contain 50 nodes with a maximum vertex degree of 4 and, thus, contain a maximum of 100 edges. A random biconnected graph and a graph designed for robustness are depicted. Note than the heights of the spanning tree increase quite slowly as edges are deleted.

The robust graphs were constructed to be resilient to edge deletion. A robust graph was specified by numbering the nodes sequentially and then specifying the number of hops between connected nodes in the pattern. A sample sixteen node robust graph is shown in Figure 5.20. The even numbered nodes have edges with one hop and six hops. The odd numbered nodes have edges of one hop and four hops. These graphs are highly symmetric and are similar to the circulant graphs studied in [14].

The effects of failure in the random and robust graphs were studied by randomly choosing edges to delete from the graph. The spanning tree heights and connectivity were measured as a function of edge deletion.

In Figure 5.21 we see that the heights of the spanning trees increase very slowly as more links fail, up to quite large numbers of failed links. Consequently, networks based on flooding should continue to deliver messages promptly even in the presence of failed links. The increase in the average depths of the spanning trees as failures occur in a graph indicates the added delay that messages will experience in reaching their destinations. Both random and robust graphs were investigated as edges (gateways) were deleted from the graphs. Simulations show that for both random and robust graphs, if the graph remains connected, the average depth of the spanning trees increases very slowly with edge deletion.

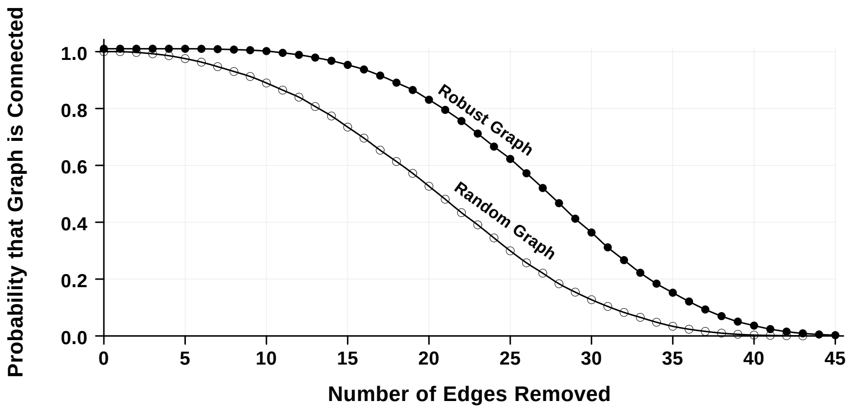


Figure 5.22: The effect on the proportion of graphs that remain connected as edges are deleted, one edge at a time. Note that edge deletion is much more significant for disconnection than for spanning tree height.

Although the Totem protocol is designed to continue despite network partitions, reliable delivery of messages can only be provided between processors that are in the same component of the partition. Thus, it is desirable to design the network to increase the probability that the network will remain connected despite failures.

Figure 5.21 includes data only for networks that remain connected. Figure 5.22 shows the proportion of networks that remain connected as edges are removed one at a time. Note that the networks start to loose connectivity with relatively few failed links. Clearly, failure to deliver a message at all is a more serious problem than late delivery due to a circuitous route. Note also that random networks are more vulnerable to disconnection due to link failure than are specially designed networks.

The data indicate that, as gateways fail, partitioning of the network is of more concern than the increased length of the routes in the network. We have found that the network layout does affect the mean number of gateways that can fail before the network partitions but that most "reasonable" network designs are resistant to partitioning if the graph of the network is at least bi-connected and the nodes have degree 4 or more.

In Figure 5.23 we consider the size of the two components into which the network is partitioned when it is disconnected. Note that, particularly for the specially designed graphs, the large majority of disconnections involve the dis-

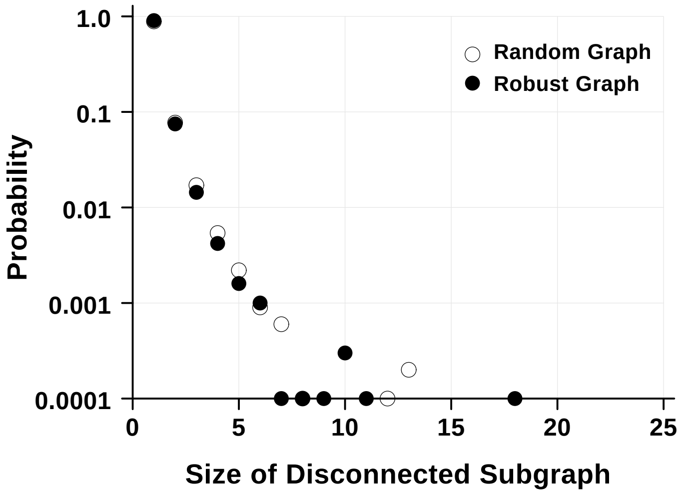


Figure 5.23: The size of the smaller disconnected component when the graph first partitions. Note that the scale on the vertical axis is logarithmic.

connection of only a few nodes. This is important as it may permit continuous operation of the larger component despite the loss of contact with a subset of the nodes. The robust graphs, however, remained connected on average five to seven edge deletions longer than the random graphs with the smaller disconnected component being only one or two nodes in over 99% of the cases. This work is also reported in [41].

5.5 Proof of Correctness

Membership

Uniqueness of Topologies

Theorem 5.1 *Each topology identifier is unique; moreover, at any time a processor or gateway is a member of at most one topology.*

Proof. A topology identifier is the lexicographically ordered list of ring identifiers of the rings that comprise the topology. By Theorem 4.1, the ring identifiers are unique and, therefore, a topology identifier is unique.

On startup, a processor is a member of the topology whose identifier is the list consisting of the identifier of the ring consisting of the processor itself, and a gateway is a member of the topology whose identifier is the pair of identifiers of the rings between which it is a gateway. Each change in the topology is signalled by delivery of a Topology Change message, which terminates the current topology and initiates the next topology. Thus, at any time a processor or gateway is a member of at most one topology. □

Consensus

Theorem 5.2 *If p and q are members of the same topology T_1 and neither receives a Configuration Change message that disconnects it from the other and if p installs a subsequent topology T_2 with a given set of rings, then q determines that the same set of rings constitutes its subsequent topology.*

Proof. Since p and q are members of the same topology T_1 and neither receives a Configuration Change message that disconnects it from the other, the Topology Change message generated by p to initiate T_2 will be forwarded to q. If q had already installed a subsequent topology T_3, then the Topology Change message initiating T_3 would also have been forwarded to p. Now either the Topology Change message to initiate T_2 or the Topology Change message to initiate T_3 has a smaller identifier (*timestamp*, *src_ring_id*, *type*, *conf_id*). The one with the smaller identifier will introduce the topology, and hence the set of rings, installed by both p and q. □

Termination

Theorem 5.3 *If a topology ceases to exist for any reason such as processor failure or network partitioning, then every processor of that topology will install a new topology, or will fail before doing so.*

Proof. Each topology change in the network is signalled by the receipt of a Configuration Change message from the single-ring protocol on each of the rings directly affected by the topology change. (At a gateway, a Configuration Change message triggers a Network Topology message which signals the change.) According to the multiple-ring protocol, a gateway that receives a Configuration Change message will forward it onto the other attached ring. By Theorem 4.8, a processor (or gateway) executing the single-ring protocol on these rings will deliver the Configuration Change message to the multiple-ring protocol executing, or will fail before doing so. If a configuration change occurs before it delivers the Configuration Change message then, by Theorem 4.3, it will deliver a Configuration Change message for a different configuration, or will fail before doing so. By Theorem 5.7, a processor or gateway executing the multiple-ring protocol will deliver the Configuration Change message or will fail before doing so.

When a gateway delivers the Configuration Change message, it sends a Network Topology message. Network Topology messages are retransmitted on the local ring if they are not received. When a gateway has received Network Topology messages from all of the gateways on the directly attached ring that experienced the configuration change, it sends a Topology Change message. A gateway will receive a Network Topology message from each of the other gateways unless a further configuration change occurs that results in its eliminating the sender of the message from the topology. Since there are only a finite number of gateways on a ring, only a finite number can be eliminated and eventually a gateway will have all of the Network Topology messages it needs to send a Topology Change message.

Topology Change messages are forwarded by the gateways and are retransmitted on the local ring if they are not received. If a forwarding path exists to a processor or gateway, then it will receive and deliver the Topology Change message in timestamp order. If no forwarding path exists, then the processor or gateway eliminates the ring containing the sender of the message prior to delivering the Topology Change message. □

Topology Change Consistency

Theorem 5.4 *Processors that are members of rings in the same topology T_2 deliver the same Initiate Topology T_2 message to begin the topology. Furthermore, if two processors install a topology T_2 directly after T_1, then the processors deliver the same Topology Change message to terminate T_1 and initiate T_2.*

Proof. The Initiate Topology T_2 message contains a list of identifiers of the rings in T_2 and is delivered when a processor installs topology T_2. Thus, processors that are members of the same topology T_2 deliver the same Initiate Topology T_2 message to begin T_2.

The Topology Change messages delivered by two processors that install T_2 directly after T_1 have the same identifier and contents. If the topology change consists of a ring deletion only and a message has been received from that ring, then the timestamp of the Topology Change message is the highest timestamp of a message in *recv_msgs* for the ring to be deleted and the *src_ring_id* of the Topology Change message is the identifier of the ring to be deleted. If the topology change consists of a ring deletion only and no message has been received from that ring, then the timestamp of the Topology Change message is the timestamp of the preceding Topology Change message and the *src_ring_id* of the Topology Change message is the *src_ring_id* of the preceding Topology Change message. Otherwise, the *timestamp* and *src_ring_id* of the Topology Change message are those of the corresponding Configuration Change message. The contents of a Topology Change message specify the rings to be added to the topology and the rings to be deleted from the topology.

If the Topology Change messages delivered by two processors that install T_2 directly after T_1 are Transitional Topology Change messages, the *timestamp* and *src_ring_id* of the Transitional Topology Change message are the same as those of the message requesting safe delivery. The *conf_id* is the lowest ring identifer in the set of singleton rings containing gateways introduced by the Transitional Topology Change message. The contents of a Transitional Topology Change message specify the rings to be added to the topology and the rings to be deleted from the topology.

Topology Change messages and Transitional Topology Change messages are delivered in the order of their identifiers, along with the other messages. The theorem now follows. $\Box$

Ordering

Reliable Delivery

Theorem 5.5 *Each ordered message m has a unique identifier.*

Proof. Each message m is identified by its *timestamp*, *src_ring_id*, *type* and *conf_id* fields. In the modified single-ring protocol, the processor that originates a regular message m places a *timestamp* in m that is greater than the value in the *timestamp* field of the token. The processor then places the *timestamp* of m into the *timestamp* field of the token; thus, timestamps on regular messages originated on a particular ring are unique. By Theorem 4.1, the *src_ring_id* is unique. The *type* of a regular message is regular and the *conf_id* field is 0.

A Configuration Change message delivered by a processor to terminate a configuration C_1 and initiate another configuration C_2 has a *src_ring_id* equal to C_2, which is unique by Theorem 4.1. If C_2 is a regular configuration, then the *timestamp* is the timestamp in the Commit token on its second rotation. If C_2 is a transitional configuration, then the *timestamp* is the highest timestamp of any message delivered by the single-ring protocol before it delivered the Configuration Change message. The *type* is Configuration Change and the *conf_id* is the identifier of C_2. For each transition from a configuration with identifier *conf_id* to a configuration with identifier *src_ring_id*, all copies of the Configuration Change message are identical and are regarded as the same message.

A Topology Change message that corresponds to a Configuration Change message (not a ring deletion only) has a *timestamp*, *src_ring_id*, and *conf_id* equal to the *timestamp*, *src_ring_id*, and *conf_id* of the corresponding Configuration Change message. The *type* is Topology Change. If the topology change consists of a ring deletion only and a message has been received from the ring to be deleted, the *timestamp* of the Topology Change message is the *max_timestamp* for the ring to be deleted, the *src_ring_id* is the identifier of the ring to be deleted, the *type* is Topology Change, and the *conf_id* is the identifier of the ring to be deleted. If the topology change consists of a ring deletion only and no message has been received from the ring to be deleted, the *timestamp* of the Topology Change message is the *timestamp* of the preceding Topology Change message, the *src_ring_id* is the identifier of that Topology Change message, the *type* is Topology Change None, and the *conf_id* is the identifier of the ring to be deleted.

All of the members of the topology initiated by the preceding Topology Change message delete rings in the same order determined by the timestamp at which they are unable to order messages (because of the lack of messages from the ring to be deleted) and by the identifiers of the rings to be deleted. Thus, the Topology Change messages to delete a ring are generated with the same *timestamp*, *src_ring_id* and *conf_id* fields.

A Transitional Topology Change message corresponds to a message that requested safe delivery but that cannot be guaranteed as safe in the current topology. The Transitional Topology Change message has a *timestamp* and *src_ring_id* of the message requesting safe delivery. The *conf_id* is the lowest ring identifier in the set of singleton rings containing gateways introduced by the Transitional Topology Change message. If the system partitions, there may be two Transitional Topology Change messages associated with the same safe message and, thus, with the same *timestamp* and *src_ring_id*. These two messages will, however, have different *conf_id* fields.

Thus, the *timestamp*, *src_ring_id*, *type* and *conf_id* fields uniquely identify the message. □

Theorem 5.6 *If processor p delivers message m, then p delivers m only once. Moreover, if processor p delivers two different messages, then p delivers one of those messages strictly before it delivers the other.*

Proof. By Theorem 5.5, each message has a unique identifier. When processor p receives a message, it places the message into its *recv_msgs* list based on the *src_ring_id* of the message, unless it has already placed a message with the same identifier in that list in which case it discards the message.

Processor p also maintains *cand_msgs* which contains, for each ring in the ring_table, the message with the lowest timestamp in *recv_msgs* for that ring. The next message to be delivered is the message with the lowest timestamp in *cand_msgs*. When processor p delivers this message, p removes it from *cand_msgs* and replaces it with the message with the next higher timestamp in *recv_msgs* for that ring and removes that message from *recv_msgs*. The theorem now follows. □

Theorem 5.7 *If processor p executing the multiple-ring protocol receives a message m from the single-ring protocol, then p will deliver m or will fail before doing so, unless m was forwarded by a gateway that is not in the topology of p as of the timestamp of the message.*

Proof. Processor p maintains a data structure in its *ring_table* for each ring in the current topology, which includes the *recv_msgs* list for that ring. Processor p deletes a ring from its *ring_table* only when it has delivered a Topology Change or Transitional Topology Change message deleting that ring and it has delivered all messages from that ring in *recv_msgs*.

Processor p places each message m it receives from the single-ring protocol into the *recv_msgs* list of the ring whose identifier is the *src_ring_id* of m, unless p has already placed a copy of m in *recv_msgs* in which case p discards m. Processor p also discards m if m has a timestamp lower than the *begin_timestamp* of the gateway that forwarded the message; this gateway will not be added to the topology until *begin_timestamp*.

Processor p maintains the *cand_msgs* list which contains, for each ring in the ring_table, the message with the lowest entry in the *recv_msgs* list for that ring. Processor p delivers messages from *cand_msgs* in the order of their timestamps. When processor p delivers a message, p removes it from *cand_msgs* and replaces it with the next lowest entry in *recv_msgs* for that ring and removes that message from *recv_msgs*.

In order to remove a message from *cand_msgs* and deliver it to the application, processor p must have a message in *cand_msgs* from each of the rings in its *ring_table* with a timestamp at least equal to that of the message to be delivered. If processor p delivers message m in safe order, then all entries in the column of the *guarantee* array, corresponding to the ring on which m was generated, contain timestamps greater than or equal to the timestamp of m. This indicates that p has received a Guarantee Vector message from each of the rings indicating that the message is safe at the single-ring protocol level on each of those rings. The Guarantee Vector messages, generated periodically by all rings and the eventual delivery properties of the single-ring protocol (Theorem 4.9) ensure that this requirement will eventually be satisfied, provided that all rings in the current topology remain connected.

If a ring becomes disconnected, a Configuration Change message will be generated by the single-ring protocol. Receipt of this message will result in a processor's generating a Topology Change message or a Transitional Topology Change message to remove the disconnected ring from its *ring_table*. Because there are only a finite number of identifiers (*timestamp*, *src_ring_id*, *type*, *conf_id*) less than a given identifier and only a finite number of rings that can be removed from the *ring_table*, by induction it follows that these messages will eventually be delivered to the application. □

Theorem 5.8 *Processor p executing the multiple-ring protocol delivers its own messages or will fail.*

Proof. By Theorem 4.7, processor p executing the single-ring protocol delivers each message originated by p to the multiple-ring protocol at p. By Theorem 5.7, processor p executing the multiple-ring protocol delivers all messages received from the single-ring protocol executed at p or will fail. □

Theorem 5.9 *If processor p is a member of topology T and no topology change ever occurs, then processor p executing the multiple-ring protocol will deliver in T all messages originated in T.*

Proof. If no topology change ever occurs, then no configuration change ever occurs. Consequently, by Theorem 4.8, a processor or gateway receives the messages originated on its local ring from the single-ring protocol. The gateways forward these messages throughout the network. Each forwarded message is broadcast on the local rings onto which it is forwarded. Again, by Theorem 4.8, a processor or gateway on these rings also receives the messages from the single-ring protocol. By Theorem 5.7, processor p executing the multiple-ring protocol will deliver these messages to the application. □

Theorem 5.10 *If processors p and q are both members of rings in consecutive topologies T_1 and T_2, then p and q executing the multiple-ring protocol deliver the same set of messages in T_1 before delivering the Topology Change message that terminates T_1 and initiates T_2.*

Proof. By Theorem 5.4, if processors p and q are both members of rings in consecutive topologies T_1 and T_2, then they both delivered the same Topology Change message to terminate T_1 and initiate T_2. Thus, they both have the same rings in their *ring_table*. By the forwarding of messages of the multiple-ring protocol and by Theorem 4.10 processors p and q receive in T_1 the same set of messages from the single-ring protocol. By Theorem 5.7, processors p and q, executing the multiple-ring protocol, will deliver these messages. They will then deliver the Topology Change message that terminates T_1 and initiates T_2. □

Delivery in Causal Order for Topology T

Theorem 5.11 *If m_1 precedes m_2 in the Lamport causal order and processor p delivers both m_1 and m_2, then p delivers m_1 before p delivers m_2.*

Proof. First we show for Lamport's causal precedence relations that if processor q originates message m_3 before processor q originates regular message m_4 or if q receives and delivers m_3 before q originates m_4, then the identifier of m_3 is less than the identifier of m_4 in the lexicographical order of identifiers (*timestamp*,*src_ring_id*,*type*,*conf_id*).

When processor q receives a message it updates its *my_timestamp* and also its *my_future_ring_seq*. The local variables *my_timestamp* and *my_future_ring_seq* are recorded to stable storage to ensure that any regular message originated by q after q recovers from a failure is ordered after any message received or originated by q before its failure.

When processor q originates a regular message, it increments its *my_timestamp* and uses that as the timestamp of the new message. The *src_ring_id* of the message is the *ring_id* of the ring of which q is a member when it originated the message. The *ring_id.seq* of that ring is greater than the *ring_id.seq* of any previous ring of which q was a member.

For Configuration Change messages and Topology Change messages, the *type* field ensures that these messages are delivered after any regular message with the same *timestamp* and *src_ring_id*. If two or more Configuration Change messages or Topology Change messages corresponding to a ring deletion have the same *timestamp*, *src_ring_id* and *type*, they are generated and delivered by disjoint sets of processors in the order of the ring identifiers in the *conf_id* field.

The *type* field in the Transitional Topology Change message ensures that it is delivered before any regular message with the same *timestamp* and *src_ring_id*. If two or more Transitional Topology Change messages have the same *timestamp* and *src_ring_id*, they are generated and delivered in the order of the ring identifier in the *conf_id* field.

By the transitivity on the lexicograpical order of identifiers, if m_1 precedes m_2 in the closure of the Lamport causal precedence relations, then the identifier of m_1 is less than the identifier of m_2.

By Theorem 5.18, if the identifier of m_1 is less than the identifier of m_2, then m_1 precedes m_2 in the Global Delivery Order. By Theorem 5.19, if p delivers both m_1 and m_2, and if m_1 precedes m_2 in the Global Delivery Order, then p delivers m_1 before p delivers m_2. $\Box$

Theorem 5.12 *If processor p originates message m with timestamp t, then processor q delivers m if and only if p is a member of q's topology at timestamp t.*

Proof. By the algorithm, at timestamp t, q delivers messages from only those rings and processors represented in its *ring_table*, *i.e.* from the members of its current topology. If p is a member of q's topology at timestamp t and q has received message m, then q delivers m. If p is a member of q's topology but q does not receive message m, then q cannot deliver messages with timestamps greater than or equal to t unless q generates a Topology Change message to remove p from its topology at the timestamp of the last message from p delivered by the single-ring protocol. Consequently, p is not a member of q's topology at t. $\Box$

Theorem 5.13 *If processor q originates message m_1, processor r originates message m_2, processor r delivers m_1 before r originates m_2, processor p delivers m_2, Topology Change or Transitional Topology Change message m_0 delivered* *by p precedes m_1 in the Lamport causal order, and for every Topology Change or Transitional Topology Change message delivered by p after m_0 and before m_2 (including m_0) r is a member of that topology, then p delivers m_1 before p delivers m_2.*

Proof. If message m_0 precedes message m_1 in the Lamport causal order, then the timestamp t_0 of m_0 is less than or equal to the timestamp t_1 of m_1. If processor p delivers message m_1 before it originates message m_2, then the timestamp t_1 of m_1 is less than the timestamp t_2 of m_2. If processor r delivers m_1 then, by Theorem 5.12, q is a member of r's topology at timestamp t_1. But r is a member of p's topology from $t_0 \leq t_1$ until $t_2 > t_1$. Thus, q is a member of p's topology at t_1. By Theorem 5.12, p delivers m_1. □

Delivery in Agreed Order for Topology T

Theorem 5.14 *The Topology Delivery Order for topology T is a total order.*

Proof. By Theorem 5.5, each message delivered in topology T has a unique identifier (*timestamp*, *src_ring_id*, *type*, *conf_id*). The lexicographical order on these identifiers defines a total order on the messages. □

Theorem 5.15 *If processor p delivers message m_2 in topology T and m_1 is any message that precedes m_2 in the Topology Delivery Order for topology T, then p delivers m_1 in T before p delivers m_2.*

Proof. If message m_1 precedes message m_2 in the Delivery Order for Topology T, then the identifier (*timestamp*, *src_ring_id*, *type*, *conf_id*) of m_1 is less than or equal to the identifier of m_2.

Processors executing the single-ring protocol on each ring originate regular messages with monotonically increasing sequence numbers and timestamps. Messages are forwarded between rings by gateways in sequence number order, and are delivered by the single-ring protocol to the multiple-ring protocol in sequence number order. Thus, if processor p executing the multiple-ring protocol receives a message from the single-ring protocol originated on a ring then, by

Theorem 4.14, p will never subsequently receive any regular message originated on that ring with a lower timestamp.

Now, the rings on which m_1 and m_2 were originated are in topology T and, thus, are present in processor p's *ring_table* when p delivered m_2. If there had been a disconnection that prevented m_1 from being communicated to p, a Configuration Change message would have been generated leading to a Topology Change message, disconnecting the source ring of m_1 immediately after the last message received from that ring. That Topology Change message would have terminated T. Since m_2 is delivered in topology T, the *ring_table* contains a message from the source ring of m_1 with an identifier greater than or equal to the identifier of m_2. Consequently, p received m_1. Since m_2 has the lowest identifier in p's *ring_table*, p has already delivered m_1. □

Delivery in Safe Order for Topology T

Theorem 5.16 *If processor p executing the multiple-ring protocol delivers message m in topology T and the originator of m requested safe delivery, then p has determined that each processor in T has received m, and will deliver m or will fail before doing so.*

Proof. If processor p delivers message m in safe order, then all entries in the column of its *guarantee* array, corresponding to the ring on which m was generated, contain timestamps greater than or equal to the timestamp of m. This condition indicates that p has received a Guarantee Vector message from a gateway on each of the rings in T containing a *guar_vector* with a timestamp for the source ring of m that is greater than or equal to the timestamp of m.

A gateway will generate and forward such a Guarantee Vector message only if the single-ring protocol executing at that gateway has delivered m in safe order. By Theorem 4.15, the single-ring protocol delivers m in safe order only if the gateway can determine that each of the processors and gateways on the ring have received m. Therefore, processor p has determined indirectly that each processor in topology T has received m. By Theorem 5.7, a processor p executing the multiple-ring protocol that receives m will deliver m or will fail before doing so. □

Extended Virtual Synchrony

Theorem 5.17 *If processor p delivers message m in topology T, then the requirements for agreed or safe delivery are satisfied.*

Proof. This follows from the preceding theorems. □

Theorem 5.18 *The Global Delivery Order is a total order.*

Proof. By Theorem 5.5, each message has a unique identifier consisting of a *timestamp*, *src_ring_id*, *type* and *conf_id*. The identifier of a regular message m consists of the *timestamp* of the message, the identifier of the regular configuration in which m was originated, the *type* regular, and the *conf_id* field 0.

The identifier of a Configuration Change message that initiates a regular configuration contains a *timestamp* that is obtained from the Commit token, whereas the identifier of a Configuration Change message that initiates a transitional configuration contains a *timestamp* that is the highest timestamp of any message delivered by the single-ring protocol before the Configuration Change message. The *src_ring_id* is the identifier of the configuration initiated by the Configuration Change message, the *type* is Configuration Change, and the *conf_id* is the identifier of the previous configuration.

The identifier of a Topology Change message for a configuration change (not a ring deletion only) consists of the *timestamp*, *src_ring_id*, and *conf_id* of the corresponding Configuration Change message, and the *type* is Topology Change. If the configuration change is a ring deletion and a message has been received from the ring to be deleted, then the *timestamp* is the *max_timestamp* for that ring, the *src_ring_id* is the identifier of the ring to be deleted, the *type* is Topology Change, and the *conf_id* is the identifier of the ring to be deleted. If the configuration change is a ring deletion only and no message has been received from the ring to be deleted, then the *timestamp* is the timestamp of the preceding Topology Change message, the *src_ring_id* is the source ring identifier of that Topology Change message, the *type* is Topology Change None, and the *conf_id* is the identifier of the ring to be deleted.

The identifier of a Transitional Topology Change message consists of the *timestamp* and *src_ring_id* of the corresponding regular message (requesting safe delivery). The *type* is Transitional Topology Change and the *conf_id* is the smallest ring identifier in the *new_rings* field of the message.

The lexicographical order on the identifiers (*timestamp*, *src_ring_id*, *type*, *conf_id*) is a total order and, thus, the Global Delivery Order is a total order. □

Theorem 5.19 *If processor p delivers messages m_1 and m_2 and m_1 precedes m_2 in the Global Delivery Order, then p delivers m_1 before p delivers m_2.*

Proof. Let $(timestamp_1,\ src\_ring\_id_1,\ type_1,\ conf\_id_1)$ and $(timestamp_2,\ src\_ring\_id_2,\ type_2,\ conf\_id_2)$ be the identifiers of messages m_1 and m_2, respectively. Without loss of generality, we assume that $(timestamp_1,\ src\_ring\_id_1,\ type_1,\ conf\_id_1) < (timestamp_2,\ src\_ring\_id_2,\ type_2,\ conf\_id_2)$. The proof is an exhaustive case analysis.

If $timestamp_1 < timestamp_2$ then, according to the algorithm, processor p delivers m_1 before p delivers m_2.

If $timestamp_1 = timestamp_2$ and $src\_ring\_id_1 < src\_ring\_id_2$, then, according to the algorithm, processor p delivers m_1 before p delivers m_2.

If $timestamp_1 = timestamp_2$, $src\_ring\_id_1 = src\_ring\_id_2$, $type_1 =$ regular and $type_2 =$ regular, then $conf\_id_1 = conf\_id_2 = 0$ and this case cannot occur.

If $timestamp_1 = timestamp_2$, $src\_ring\_id_1 = src\_ring\_id_2$, $type_1 =$ regular and $type_2 =$ Configuration Change, then this case cannot occur. For if the Configuration Change message m_2 introduces a regular configuration, then all regular messages from that configuration have higher timestamps than the timestamp of the Configuration Change message m_2. If the Configuration Change message m_2 introduces a transitional configuration, then the source ring identifier of any regular message is that of the regular configuration in which that message was originated and, thus, cannot equal the identifier of the transitional configuration.

If $timestamp_1 = timestamp_2$, $src\_ring\_id_1 = src\_ring\_id_2$, $type_1 =$ regular, $type_2 =$ topology change and the Topology Change corresponds to a Configuration Change message (not a ring deletion only), then this case cannot occur because the corresponding Configuation Change message cannot exist.

If $timestamp_1 = timestamp_2$, $src\_ring\_id_1 = src\_ring\_id_2$, $type_1 =$ regular, $type_2 =$ Topology Change, the topology change corresponds to a ring deletion and some message has been received from the ring to be deleted, then the Topology Change message is delivered immediately after the last message from that ring delivered by the multiple-ring protocol, *i.e.* the message with identifier ($timestamp_1$,$src\_ring\_id_1$,regular,0).

If $timestamp_1 = timestamp_2$, $src\_ring\_id_1 = src\_ring\_id_2$, $type_1 =$ regular, and $type_2 =$ Topology Change None, then this case cannot occur. The Topology Change None message m_2 must be delivered immediately following the Topology Change message that introduced the ring to be deleted. Thus, the Topology Change None message m_2 has the same timestamp and source ring identifier as the Topology Change message and, thus, the same timestamp and source ring identifier as the Configuration Change message corresponding to the Topology Change message. Since the regular message m_1 cannot have the same timestamp and source ring identifier as a Configuration Change message, it cannot have the same timestamp and source ring identifier as the Topology Change None message.

If $timestamp_1 = timestamp_2$, $src\_ring\_id_1 = src\_ring\_id_2$, $type_1 =$ Configuration Change, and $type_2 =$ Configuration Change, then it cannot be the case that one of the messages initiates a regular configuration and the other initiates a transitional configuration because then $src\_ring\_id_1 = src\_ring\_id_2$. If both of the messages initiate a regular configuration, then these messages correspond to different prior transitional configurations and a processor delivers only one of these messages. If both messages initiate a transitional configuration, then $conf\_id_1 = conf\_id_2 = 0$ and this case cannot occur.

If $timestamp_1 = timestamp_2$, $src\_ring\_id_1 = src\_ring\_id_2$, $type_1 =$ Configuration Change, $type_2 =$ Topology Change and the topology change corresponds to a Configuration Change message, then the Topology Change message is delivered immediately after the Configuration Change message.

If $timestamp_1 = timestamp_2$, $src\_ring\_id_1 = src\_ring\_id_2$, $type_1 =$ Configuration Change, $type_2 =$ Topology Change, the topology change corresponds to a ring deletion and some message has been received from the ring to be deleted, then this case cannot occur. The timestamp of the Topology Change message is the timestamp of the last regular message delivered from the ring to be deleted and the source ring identifier is the identifier of that ring. But, no regular message can have the same timestamp and source ring identifier as a Configuration Change message.

If $timestamp_1 = timestamp_2$, $src\_ring\_id_1 = src\_ring\_id_2$, $type_1 =$ Configuration Change, and $type_2 =$ Topology Change None, then the Topology Change None message is delivered immediately after the Topology Change message associated with the Configuration Change message.

If $timestamp_1 = timestamp_2$, $src\_ring\_id_1 = src\_ring\_id_2$, $type_1 =$ Topology Change corresponding to a Configuration Change message, and $type_2 =$ Topology Change corresponding to a Configuration Change message, then messages m_1 and m_2 are associated with Configuration Change messages that initiate the same configuration. A processor delivers only one such message.

If $timestamp_1 = timestamp_2$, $src\_ring\_id_1 = src\_ring\_id_2$, $type_1 =$ Topology Change corresponding to a Configuration Change message, and $type_2 =$ Topology Change corresponding to a ring deletion and some message has been received from the ring to be deleted, then this case cannot occur. Message m_1 has the same timestamp and source ring identifier as some Configuration Change message. Message m_2 has the same timestamp and source ring identifier as some regular message. But no Configuration Change message and regular message can have the same timestamp and source ring identifier.

If $timestamp_1 = timestamp_2$, $src\_ring\_id_1 = src\_ring\_id_2$, $type_1 =$ Topology Change corresponding to a Configuration Change message, and $type_2 =$ Topology Change None, then message m_2 is delivered immediately after message m_1.

If $timestamp_1 = timestamp_2$, $src\_ring\_id_1 = src\_ring\_id_2$, $type_1 =$ Topology Change corresponding to a ring deletion and some message has been received from the ring to be deleted, and $type_2 =$ Topology Change corresponding to a ring deletion and some message has been received from the ring to be deleted, then $conf\_id_1 = conf\_id_2$ and so this case cannot occur.

If $timestamp_1 = timestamp_2$, $src\_ring\_id_1 = src\_ring\_id_2$, $type_1 =$ Topology Change corresponding to a ring deletion and some message has been received from the ring to be deleted, and $type_2 =$ Topology Change None, then this case cannot occur. Message m_1 has the same timestamp and source ring identifier as a regular message. Message m_2 has the same timestamp and source ring identifier as a Configuration Change message. But a regular message and a Configuration Change message cannot have the same timestamp and source ring identifier.

If $timestamp_1 = timestamp_2$, $src\_ring\_id_1 = src\_ring\_id_2$, $type_1 =$ Topology Change None, and $type_2 =$ Topology Change None then, by the algorithm, messages m_1 and m_2 are delivered in the order of the identifiers of the rings to be deleted.

If $timestamp_1 = timestamp_2$, $src\_ring\_id_1 = src\_ring\_id_2$, $type_1 = type_2 =$ Transitional Topology Change, then this case cannot occur. Processor p delivers either m_1 or m_2 because m_1 and m_2 are generated in disjoint topologies.

If $timestamp_1 = timestamp_2$, $src\_ring\_id_1 = src\_ring\_id_2$, $type_1 =$ Transitional Topology Change, and $type_2 =$ regular then, by the algorithm, the Transitional Topology Change message m_1 is delivered immediately before the corresponding safe message m_2.

If $timestamp_1 = timestamp_2$, $src\_ring\_id_1 = src\_ring\_id_2$, $type_1 =$ Transitional Topology Change, and $type_2 =$ Configuration Change, then there must have been a regular message m_3 that requested safe delivery such that $timestamp_1{=}timestamp_3$ and $src\_ring\_id_1 = src\_ring\_id_3$. As demonstrated above m_1 is delivered m_3 and m_3 is delivered m_2 and, therefore, m_1 is delivery before m_2. The arguments for $type_2 =$ Topology Change and $type_2 =$ Topology Change None are similar.

In any case, processor p delivers m_1 before p delivers m_2. The Global Delivery Order is, thus, the set of messages delivered by all of the processors. □

5.6 Summary

The Totem multiple-ring protocol uses a hierarchical approach to provide reliable ordered delivery of messages across interconnected rings. It allows the design of fault-tolerant distributed systems to be simplified through the use of reliable ordered message delivery by exploiting efficient local-area message ordering and processor membership protocols.

Although several reliable ordered message delivery protocols have been developed in the past, none has adequately addressed the difficult issues of maintaining consistency in a network that partitions and remerges. The multiple-ring protocol leverages off the previous work on reliable ordered delivery in broadcast domains to provide message delivery across a larger network, while maintaining consistency through partitioning and remerging of the network.

The multiple-ring protocol provides message delivery in a total order that is consistent across the entire network. Total ordering of messages across a larger network may delay the ordering of a message longer than the message would have been delayed with partial ordering of messages. This longer delay is due to the need to satisfy the additional constraints of the total order. Partial ordering may, however, introduce inconsistencies in message delivery if partitioning and remerging occurs.

Chapter 6

Conclusions and Recommendations

This dissertation has presented a reliable ordered delivery protocol, called Totem, for interconnected local-area networks. Each local-area network is assumed to be a broadcast domain; imposed on the broadcast domain is a logical token-passing ring. The single-ring protocol delivers messages in a total order to the multiple-ring protocol which, in turn, delivers messages in a total order to the next higher layer in the protocol stack. Messages are ordered by timestamp, and messages with the same timestamp are ordered by source ring identifier. If the timestamp and source ring identifier are the same, messages are ordered by type. This ordering on messages provides a single consistent total order across the entire network, even though some messages may not be delivered to all processors. Timestamps can be used to order messages only if a processor knows that it has received all of the preceding messsages in the order. Receipt of such messages is guaranteed by the consecutive sequence numbers on the individual rings and by the Guarantee Vector messages that are forwarded through the network.

The dissertation also presents a membership algorithm for Totem that provides recovery from processor failure and network partitioning, as well as loss of all copies of the token. The membership algorithm is integrated with the message ordering algorithm to provide consistent membership information to the application. The Configuration Change messages ordered by the single-ring protocol are used by the multiple-ring protocol to maintain consistent topology information. Each gateway in the network maintains the current topology, and topology changes are indicated by Topology Change messages. These messages are delivered to the application in a consistent total order along with the other messages. Consistent information regarding membership and topology changes is important for the application programs since they are making decisions regarding which action to take based on which processors are currently in the network when a message is delivered.

As a part of the work on Totem we have redefined the consistency requirements to allow separate components of a network to continue ordering messages after partitioning and remerging. In particular, we have introduced the concept of extended virtual synchrony, which extends the properties of virtual synchrony defined by Birman and others [13]. Extended virtual synchrony ensures that the processors are provided membership information in an order that is consistent with the order of the regular messages.

An implementation of the Totem single-ring protocol on Sun IPC Sparcstations over a 10Mbit Ethernet has been completed. Five processors executing the protocol achieved a throughput of approximately 810 KBytes/second while passing messages containing 1 KByte of data each. This performance compares favorably to Isis and Transis which achieved 151 KBytes/second and 300 KBytes/second respectively with similar equipment [5, 13].

The performance of the single-ring membership algorithm has also proven excellent. The same five processors executing the membership algorithm required on average 40 milliseconds to reach consensus on the membership of a token ring, to form the token ring, and to begin normal operation after determining that the token was lost. With the token retransmission mechanism activated, the time to resume normal operation was less than 20 milliseconds on average.

An implementation of the Totem multiple-ring protocol has also been completed on Sun IPC Sparcstations over two 10Mbit Ethernets with a Sparcstation 20 acting as a gateway between the two Ethernets. Three processors on each ring achieved a throughput of approximately 625 KBytes/second while passing messages containing 1 KByte of data each. This compares to a throughput of approximately 769 KBytes/second for the single-ring with three processors. The multiple-ring protocol adds approximately 0.28 milliseconds per message in ordering overhead.

Although the foundation of a system has been laid by the work in this dissertation, there are still several issues that need to be addressed to complete the work on the Totem system. These issues include process groups, adaptive flow control, routing, real-time guarantees, application tools and demonstration applications.

In a distributed system, processes executing application tasks normally co-operate in groups with each group spanning only a subset of the processors. The difficulty lies in maintaining consistency of the message order when there are overlapping process groups. The totally ordered message delivery provided by the Totem protocol ensures that consistency is maintained across all process groups and all messages. Messages are ordered network-wide, but messages destined for a particular process group are only delivered to the members of that group. Design and implementation of the Totem process group interface and membership protocol is complete.

The Totem multiple-ring protocol is the first protocol known to provide consistent reliable ordered delivery of messages across interconnected broadcast domains, but its real potential will be realized when process group dependent forwarding of messages is implemented. This will require a gateway to filter messages so that messages are forwarded only if there is a member of the destination process group in the direction of the forwarding. The multiple-ring membership/topology protocol and the process group membership protocol provide the foundation for this filtering mechanism. The effective throughput seen by each application should improve significantly once filtering is implemented. This is particularly true in cases where the process groups exhibit a high degree of locality and their messages need not be forwarded across the entire network.

The current version of Totem assumes that all messages are flooded through the network. Other routing strategies, such as spanning trees, need to be investigated, as well as their effect on maintaining a consistent total order. This may involve modifications to the current protocol, but will certainly benefit from the current version. Among the interesting questions to be investigated related to routing are the effects of failures on maintaining routes, changing the routes without disrupting the total order on messages, providing different routes for messages intended for different process groups, and selection of routing strategies to enhance performance.

The rudimentary flow-control mechanisms implemented in the single-ring protocol and in the multiple-ring protocol do not adapt to changing conditions in the network. The multiple-ring protocol flow control is designed to allow the gateways and processors in the network to signal congestion conditions and to avoid message loss, and has been effective at accomplishing this goal. But, its current response of refusing all new messages from the application processes until the congestion is relieved may be too harsh. A solution which adaptively modifies the flow-control parameters to maintain high throughput and low latency without congestion might provide better characteristics at the application interface. In particular, the single-ring protocol needs to adapt to the load offered by a gateway and to allow it to forward messages.

Although reliable ordered delivery protocols have been around for a number of years, the limited performance and correctness criteria have hampered their usefulness to application developers. We believe that the Totem protocol provides significant improvements in both of these areas and will ease the application developer's task. Application developers need, however, to rethink their designs to utilize reliable ordered delivery protocols effectively. Relatively little work in this area has been done, and more effort to develop interface tools and demonstration applications is warranted.

Bibliography

[1] D. A. Agarwal, P. M. Melliar-Smith, and L. E. Moser. Totem: A protocol for message ordering in a wide-area network. In *Proceedings of the First International Conference on Computer Communications and Networks*, pages 1–5, San Diego, CA, June 1992.

[2] D. Agrawal and A. El Abbadi. Integrating security with fault-tolerant distributed databases. *The Computer Journal*, 33(1):71–78, 1990.

[3] D. Agrawal and A. El Abbadi. Storage efficient replicated databases. *IEEE Transactions on Knowledge and Data Engineering*, 2(3):342–352, September 1990.

[4] D. Agrawal and A. Malpani. Efficient dissemination of information in computer networks. *The Computer Journal*, 34(6):534–541, 1991.

[5] Y. Amir. Performance measurements of Transis. Private communication, August 1992.

[6] Y. Amir, D. Dolev, S. Kramer, and D. Malki. Membership algorithms for multicast communication groups. In *Proceedings of the 6th International Workshop on Distributed Algorithms, Lecture Notes in Computer Science 647*, pages 292–312, Haifa, Israel, November 1992.

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

[8] Y. Amir, L. E. Moser, P. M. Melliar-Smith, D. A. Agarwal, and P. Ciarfella. Fast message ordering and membership using a logical token-passing ring. In *Proceedings of the 13th International IEEE Conference on Distributed Computing Systems*, pages 551–560, Pittsburgh, PA, May 1993.

[9] H. Attiya, A. Bar-Noy, D. Dolev, D. Koller, D. Peleg, and R. Reischuk. Achievable cases in an asynchronous environment. In *28th Annual Symposium on Foundations of Computer Science*, pages 337–346, Los Angeles, CA, October 1987.

[10] D. Bertsekas and R. Gallager. *Data Networks, 2nd ed.* Prentice Hall, 1992.

[11] K. P. Birman. A response to Cheriton and Skeen's criticism of causal and totally ordered communication. *Operating Systems Review*, 28(1):11–21, January 1994.

[12] K. P. Birman and T. A. Joseph. Reliable communication in the presence of failures. *ACM Transactions on Computer Systems*, 5(1):47–76, February 1987.

[13] K. P. Birman, A. Schiper, and P. Stephenson. Lightweight causal and atomic group multicast. *ACM Transactions on Computer Systems*, 9(3):272–314, August 1991.

[14] F. T. Boesch and R. Tindell. Circulants and their connectivities. *Journal of Graph Theory*, 8(4):487–499, 1984.

[15] S. Casner and S. Deering. First IETF Internet audiocast. *Computer Communications Review*, 22(3):92–97, July 1992.

[16] T. Chandra and S. Toueg. Unreliable failure detectors for asynchronous systems. In *Proceedings of the Tenth ACM Symposium on Principles of Distributed Computing*, pages 325–340, Montreal, Québec, August 1991.

[17] J. M. Chang and N. F. Maxemchuk. Reliable broadcast protocols. *ACM Transactions on Computer Systems*, 2(3):251–273, August 1984.

[18] D. R. Cheriton and D. Skeen. Understanding the limitations of causally and totally ordered communication. *Operating Systems Review*, 27(5):44–57, December 1993.

[19] W. J. Chun. Virtual gateways: Performing distributed simulations in the Totem protocol development environment. Master's thesis, University of California, Santa Barbara, 1994.

[20] P. W. Ciarfella. The Totem protocol testbed. Master's thesis, University of California, Santa Barbara, 1993.

[21] P. W. Ciarfella, L. E. Moser, P. M. Melliar-Smith, and D. A. Agarwal. The Totem protocol development environment. In *Proceedings of the 1994 International Conference on Network Protocols*, pages 168–177, Boston MA, October 1994.

[22] F. Cristian. Synchronous atomic broadcast for redundant broadcast channels. *Journal of Real-Time Systems*, 2:195–212, 1990.

[23] F. Cristian. Understanding fault-tolerant distributed systems. *Communications of the ACM*, 34(2):56–78, February 1991.

[24] S. B. Davidson, H. Garcia-Molina, and D. Skeen. Consistency in partitioned networks. *Computing Surveys*, 17(3):341–370, September 1985.

[25] D. Dolev, S. Kramer, and D. Malki. Early delivery totally ordered multicast in asynchronous environments. In *23nd Annual International Symposium on Fault-Tolerant Computing*, pages 544–553, Toulouse, France, June 1993.

[26] A. El Abbadi and S. Toueg. Maintaining availability in partitioned replicated databases. *ACM Transactions on Database Systems*, 14(2):264–290, June 1989.

[27] M. J. Fischer, N. A. Lynch, and M. Merritt. Easy impossibility proofs for distributed consensus problems. *Distributed Computing*, 1:26–39, 1986.

[28] M. J. Fischer, N. A. Lynch, and M. S. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the Association of Computing Machinery*, 32(2):374–382, April 1985.

[29] N. C. Hutchinson and L. L. Peterson. Design of the x-kernel. In *Proceedings ACM SIGCOMM '88 Symposium on Communications Architectures and Protocols*, pages 65–75, 1988.

[30] N. C. Hutchinson and L. L. Peterson. The x-kernel: An architecture for implementing network protocols. *IEEE Transactions on Software Engineering*, 17(1):64–76, January 1991.

[31] M. F. Kaashoek. Group communication in distributed computer systems. Technical report, Vrije Universiteit, Amsterdam, 1992.

[32] M. F. Kaashoek and A. S. Tanenbaum. Group communication in the Amoeba distributed operating system. In *International Conference on Distributed Computing Systems*, pages 222–230, Arlington, TX, May 1991.

[33] M. King. A network monitor for the Totem simulation testbed. Master's thesis, University of California, Santa Barbara, 1993.

[34] H. Kopetz, A. Damm, C. Koza, M. Mulazzani, W. Schwabl, C. Senft, and R. Zainlinger. Distributed fault-tolerant real-time systems: The Mars approach. *IEEE Micro*, pages 25–40, February 1989.

[35] H. Kopetz and G. Grünsteidl. TTP– A protocol for fault-tolerant real-time systems. *Computer*, 27(1):14–23, January 1994.

[36] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, July 1978.

[37] C. A. Lingley-Papadopoulos. The Totem process group membership and interface. Master's thesis, University of California, Santa Barbara, 1994.

[38] S. W. Luan and V. D. Gligor. A fault-tolerant protocol for atomic broadcast. In *Proceedings of the 7th Symposium on Reliable Distributed Systems*, pages 112–126, Columbus, Ohio, October 1988.

[39] N. A. Lynch. A hundred impossibility proofs for distributed computing. In *Proceedings of the Eighth Annual ACM Symposium on Principles of Distributed Computing*, pages 1–28, Edmonton, Alta., Canada, August 1989.

[40] P. M. Melliar-Smith and L. E. Moser. Trans: A reliable broadcast protocol. *IEE Transactions on Communications*, 140(6):481–493, December 1993.

[41] P. M. Melliar-Smith, L. E. Moser, and D. A. Agarwal. Ring-based ordering protocols. In *Proceedings of the International Conference on Information Engineering*, pages 882–891, Singapore, December 2-5 1991.

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

[43] S. Mishra, L. L. Peterson, and R. D. Schlichting. A membership protocol based on partial order. In *Proceedings of the International Working Conference on Dependable Computing for Critical Applications*, volume 6, pages 309–331, Tucson, AZ, February 1991.

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

[45] L. E. Moser, P. M. Melliar-Smith, and V. Agrawala. Asynchronous fault-tolerant total ordering algorithms. *SIAM Journal of Computing*, 22(4):727–750, August 1993.

[46] L. E. Moser, P. M. Melliar-Smith, and V. Agrawala. Necessary and sufficient conditions for broadcast consensus protocols. *Distributed Computing*, 7(2):75–85, December 1993.

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

[48] B. M. Oki and B. H. Liskov. Viewstamped replication: A new primary copy method to support highly-available distributed systems. In *Proceedings of the ACM Symposium on Principles of Distributed Computing*, pages 8–17, Toronto, Canada, 1988.

[49] L. L. Peterson, N. C. Buchholz, and R. D. Schlichting. Preserving and using context information in interprocess communication. *ACM Transactions on Computer Systems*, 7(3):217–246, August 1989.

[50] B. Rajagopalan and P. K. McKinley. A token-based protocol for reliable, ordered multicast communication. In *Proceedings of the 8th IEEE Symposium on Reliable Distributed Systems*, pages 84–93, Seattle, WA, October 1989.

[51] A. M. Ricciardi and K. P. Birman. Using process groups to implement failure detection in asynchronous environments. In *Proceedings of the Tenth ACM Symposium on Principles of Distributed Computing*, pages 341–353, Montreal, Quebec, Canada, August 1991.

[52] R. D. Schlichting and F. B. Schneider. Fail-stop processors: an approach to designing fault-tolerant computing systems. *ACM Transactions on Computer Systems*, 1(3):222–238, August 1983.

[53] A. Siegel. *Performance in flexible distributed file systems.* PhD thesis, Cornell University, 1992.

[54] J. S. Turner. New directions in communications (or which way to the information age?). *IEEE communications Magazine*, 24(10):8–15, October 1986.

[55] R. van Renesse. Why bother with CATOCS. *Operating Systems Review*, 28(1):22–27, January 1994.

Appendix A

A User's Guide for Totem

This appendix describes how to compile and run the Totem protocol. Totem provides reliable ordered delivery of messages across interconnected broadcast domains. Reliable ordered delivery within a broadcast domain is provided by the single-ring protocol. The multiple-ring protocol uses the single-ring protocol to provide reliable ordered delivery across interconnected broadcast domains. The Totem protocol code has been designed to run either as an implementation or as a simulation.

The Implementation

The Totem protocol has been implemented in C and has been tested on Sun IPC Sparcstations running Sun OS 4.1.1 on a 10Mbit/s Ethernet. The code has been compiled using gcc version 2.3.3. The code was also recently compiled on Sun OS 4.1.3 using gcc without problems.

The current implementation uses UNIX UDP sockets to broadcast messages and to transfer the token. The token socket numbers are deterministic based on processor identifier and network identifier. The system clock is utilized to track all timing information.

To allow greater flexibility, multiple rings can be run on a single Ethernet. This allows the use of a processor with only one network interface as a gateway by defining two virtual network interfaces on the one actual interface; each gateway runs as a single process on a processor. We have found in practice that up to four rings can be run on a single Ethernet before the performance degrades too far to be useful. This number was determined running Sun 4/IPC processors and will probably decrease with faster processors that are better able to saturate a single Ethernet.

The Totem protocol can be exercised either by a packet driver which generates periodic traffic or by application processes which pass messages. For a description of the application interface routines, refer to [37].

The Simulator

The need to study the protocols in a controlled environment led to the development of a simulation testbed. The simulator allows testing and debugging of the protocol.

The simulator is executed by linking in simulated versions of the system clock and system socket calls. The simulated clock halts time when an individual processor is halted and continues when the processor continues. This allows an individual processor to be halted in the simulator and in the system beingq studied while the other processors wait for the halted processor.

The simulator tries to keep all the processors executing in parallel as much as possible. This task is more difficult than would appear since the processors do not operate in lock-step. The simulator must determine whether the current next event for a processor is indeed the next event to be executed by the processor; the current next event may in fact be preceded by the receipt of a message or receipt of the token, since these events are generated by outside sources.

The simulator uses a simplified state machine model of the protocol to represent events. This state machine is used to decide whether to execute the next event for a processor. The simulator is separated into two parts: one that models the communication medium and the other that provides the processor interface to the communication medium. The communication medium model maintains a global event list which registers the events for each of the processors. It also maintains the system clock and a state machine for each of the processors.

The communication medium model maintains the global clock in shared memory. This memory can be accessed by the processors. Shared memory is also used to implement the socket select calls. A more complete description of the simulator can be found in [20].

The simulator also allows execution of the multiple-ring protocol. To accomplish this, the gateway is split across two physical processors where each processor is running a separate ring. The two components of the gateway are connected by a TCP socket which provides reliable message delivery between the two portions of the gateway. The multiple-ring protocol extensions to the simulator are described in [19].

A network monitor has been developed in Motif for use in monitoring the activity of the Totem single-ring protocol within the simulator. The monitor provides a graphical display of the progress of message delivery and membership in the protocol. A more complete description of the network monitor can be found in [33].

Compilation of Totem

The Totem code is divided into four subdirectories:

SRC - source code files

HDR - header files

OBJ - compiled object files

EXEC - protocol executables.

The make file for the Totem protocol has four targets:

totem - Totem implementation version of a processor or gateway

pm - Totem simulator version of a processor or gateway

cmm - Totem simulator for the communication medium model

xtmm - Totem simulator monitor (requires Motif to compile).

The *totem* and *pm* executables have similar arguments. These arguments can be specified on the command line or a configuration file. The command line must at least contain a -c or -f argument to be valid. Defaults for most of the rest of the arguments are supplied automatically:

-f <config file> - allows specification of the arguments in a file

-c <computer number> - computer number of this processor (assumed to be unique)

-n <network num> - network to which to connect; this defaults to the local network

-d <discard limit> - number of messages to deliver before halting execution

-e <mess each round> - maximum number of messages sent on a token visit

-w <window size> - maximum number of messages that can be sent in a token rotation

-o - run only the single-ring protocol

-p - use the packet driver to generate traffic

-s <message size> - size of each message if using the packet driver

-t <monitor hostname> - name of the machine running the xtmm protocol monitor (used only with the simulator)

-u - deliver packets to the user (requires the multiple-ring protocol to be running)

The *cmm* executable also has arguments which can be specified on the command line.

-f <events file> - file containing scheduled network partition events

-m <probability> - message reception probability

-t <probability> - token reception probability.

Examples

Implementation

To run the Totem implementation using the packet driver to generate traffic, type

totem -c 101 -p

To run the Totem implementation with a user application above it (the packet driver should not be run if a user is specified), type

totem -c 200 -u

To run the Totem implementation using a configuration file, type

totem -f proc1.cfg

where the file proc1.cfg contains the line

srp: -c 101 -n 63

The Totem protocol is normally run on several processors simultaneously, and each processor is given a unique computer number. To run several rings on a single Ethernet, use the network number to specify a network number for each ring. When running a gateway, the computer and network numbers must be specified in a configuration file. To run a gateway which also contains a packet driver, type

totem -f gway1.cfg -g -p

where the file gway1.cfg contains the lines

srp: -c 94 -n 193
srp: -c 35 -n 154

Simulator

To run the simulator, first start the *cmm*. Once the *cmm* has been started, the individual *pm*s can be started to simulate the actual processors on the network. The individual processors and the *cmm* should be run on the same physical processor.

cmm

To run the simulator with 5% message loss, type

cmm -m 0.95

To run partitioning scenarios using an events file, type

cmm -f part.cfg

The file part.cfg contains

<event type> <simulation time> <network> <listing of processors in each partition>

partition: 40000000 193 (100 104 105) (101 103)
partition: 65000000 193 (100 104 101 103 105)
partition: 74000000 193 (100 101) (105) (104 103)

To run an individual processor in the simulation with the packet driver, use the following command on the same physical processor as the one on which the *cmm* is running:

pm -c 101 -p

The same arguments can be used with *pm* as were used in the examples. The -t argument is, however, used only with the *pm* to connect to the protocol monitor. For example, to connect a *pm* to a monitor running on a machine named omega, type

pm -c 201 -t omega

For a description of how to run the simulator with multiple rings, refer to [19]. For a description of how to run the network monitor, refer to [33]. Note that the *xtmm* runs without arguments and should be started before any of the *pm*s. The *pm*s need to be run with the -t option to have their status displayed by the *xtmm*. Each instance of the *xtmm* can display the status of a single *cmm*.

Parameters in Totem

There are several parameters which can be set to customize the Totem protocol. These parameters are all in the file HDR/ring_public.h and include everything from frequency of status printouts to flow control parameters.

Defaults

#define COMP_NUM_DEFAULT -1
#define MEMB_SIZE_DEFAULT 9999
#define DISCARD_LIMIT_DEFAULT 100000
#define EACH_TIME_DEFAULT 10

```
#define WINDOW_SIZE_DEFAULT 50
#define MESSAGE_SIZE_DEFAULT 1024
#define NETWORK_DEFAULT 63 /* Hard-coded per site */
#define MAX_DATA_PACKET 1460 /* max will be 1512 */
#define MAX_REQUESTS 90
#define MAX_PROC_RING 60
#define MAX_GWAY_RING 25
```

Single-Ring Parameters

Seconds between status printouts

```
#define SRP_DEBUG_TIMEOUT 500
```

Parameters for delay of token when there is no traffic on the ring.

Delay before retransmitting token

```
#define MAX_TKN_RETRANS_SEC 0
#define MAX_TKN_RETRANS_USEC 800000
```

Max delay any one processor can add to token

```
#define MAX_DELAY_PROC 50000
```

Number of rounds of no traffic before delay

```
#define MAX_FAST_ROUNDS 30
```

Limit for number of times the aru has been seen unchanged for failure to receive.

```
#define FAIL_RCV_LIMIT 20
```

Single-Ring Membership Parameters

```
#define JOIN_TMO_SEC 1
#define JOIN_TMO_USEC 300000
#define CONSENSUS_TMO_SEC 3
#define CONSENSUS_TMO_USEC 750000
#define TOKEN_TMO_SEC 8
#define TOKEN_TMO_USEC 500000
```

Multiple-Ring Parameters

```
#define MAX_PACKET_GAP (2*EACH_TIME_DEFAULT)
```

Timeout for periodic status message

```
#define WAP_DEBUG_TIMEOUT 8
```

Maximum Numbers for Topology

Max rings in topology graph or message

#define MAX_NODES 20

Max gateways listed in topology message

#define MAX_EDGES 40

Packet Driver Parameters

Defines the maximum number of packets in the send queue when the packet driver is done as a multiple of window_size.

#define MAX_QUEUE_MULTIPLE 1

Defines how often the periodic null packet driver is called

#define NULL_PERIODIC_TMO_SECS 2

#define NULL_PERIODIC_TMO_USECS 100000

Defines how often the packet driver is called

#define PKT_DRVR_TMO_SECS 5

#define PKT_DRVR_TMO_USECS 500000

Flow Control Parameters

Constants for determining upper and lower thresholds for send queue flow control.

When send queue reaches this level block site

#define MAX_SRP_SND_Q 300

When send queue gets back down to this level unblock site

#define MIN_SRP_SND_Q 250

User Applications

An application built on top of Totem must use the calls for the process group interface defined in [37]. To build a user application on top of Totem, see the source file ha.c (high availability) which contains the routines that must be used.