The Totem Multiple-Ring Ordering and Topology Maintenance Protocol

D. A. AGARWAL, L. E. MOSER, P. M. MELLIAR-SMITH, and R. K. BUDHIA
University of California

The Totem multiple-ring protocol provides reliable totally ordered delivery of messages across multiple local-area networks interconnected by gateways. This consistent message order is maintained in the presence of network partitioning and remerging, and of processor failure and recovery. The protocol provides accurate topology change information as part of the global total order of messages. It addresses the issue of scalability and achieves a latency that increases logarithmically with system size by exploiting process group locality and selective forwarding of messages through the gateways. Pseudocode for the protocol and an evaluation of its performance are given.

Categories and Subject Descriptors: C.2.1 [**Computer-Communication Networks**]: Network Architecture and Design—*network communications*; C.2.2 [**Computer-Communication Networks**]: Network Protocols—*protocol architecture*; C.2.4 [**Computer-Communication Networks**]: Distributed Systems—*network operating systems*; C.2.5 [**Computer-Communication Networks**]: Local Networks—*rings*; D.4.4 [**Operating Systems**]: Communications Management—*network communication*; D.4.5 [**Operating Systems**]: Reliability—*fault-tolerance*; D.4.7 [**Operating Systems**]: Organization and Design—*distributed systems*

General Terms: Algorithms, Performance, Reliability

Additional Key Words and Phrases: Lamport timestamp, network partitioning, reliable delivery, topology maintenance, total ordering, virtual synchrony

---

An earlier abbreviated version of the Totem multiple-ring ordering and topology maintenance protocol appeared in the Proceedings of the International Conference on Network Protocols, Tokyo, Japan (November 1995).
This research was supported by NSF Grant No. NCR-9016361 and DARPA Contract No. N00174-93-K-0097 and N00174-95-K-0083.
Authors' addresses: D. A. Agarwal, Ernest Orlando Lawrence Berkeley National Laboratory, 1 Cyclotron Road, MS 50B-2239, Berkeley, CA 94720; email: daagarwal@lbl.gov; L. E. Moser and P. M. Melliar-Smith, Department of Electrical and Computer Engineering, University of California, Santa Barbara, CA 93106; email: {moser; pmms}@ece.ucsb.edu; R. K. Budhia, Tandem Computers, Inc., 19333 Vallco Parkway, Cupertino, CA 95014; email: budhia_ravi@ntos.tandem.com.
Permission to make digital/hard copy of part or all of this work for personal or classroom use is granted without fee provided that the copies are not made or distributed for profit or commercial advantage, the copyright notice, the title of the publication, and its date appear, and notice is given that copying is by permission of the ACM, Inc. To copy otherwise, to republish, to post on servers, or to redistribute to lists, requires prior specific permission and/or a fee.
© 1998 ACM 0734-2071/98/0500–0093 $5.00

1. INTRODUCTION

In a fault-tolerant distributed system, both the processes executing application tasks and the data must be replicated to protect against faults, but inconsistencies in the replicated data can arise if processes receive and process the same messages in different orders. A reliable totally ordered delivery protocol that delivers messages in a global total order across the distributed system makes programming the application considerably simpler than if messages are unordered or only causally ordered. If the processes holding copies of the replicated data receive the same messages in the same order, they will update the replicated data in the same order, thereby preserving replica consistency. Simpler programming increases system reliability and reduces system development time and cost.

The Totem multiple-ring protocol provides a consistent global total order on messages transmitted over multiple local-area networks interconnected by gateways within a local area. A logical token-passing ring is imposed on each local-area network (LAN) with reliable totally ordered delivery of messages being provided on each LAN. The protocol also provides detection of, and recovery from, processor and network faults, as well as topology maintenance services. Consistency of message ordering is guaranteed, even if the network partitions and remerges, or if processors fail and restart. These services are provided for arbitrary topologies and for arbitrarily intersecting process groups.

The Totem multiple-ring protocol mitigates the effects of large system size on the latency to message delivery. If a single LAN with a single ring were used, the latency to message delivery would be linear in the number of processors on the ring. With multiple rings interconnected by gateways, the latency to message delivery can be reduced from linear to logarithmic. This is achieved by structuring the application so that processes communicating frequently are located on the same LAN and by forwarding messages through the gateways only if there are processes in the destination group in the direction of the forwarding. This principle of process group locality with selective forwarding of messages enables Totem to regain some of the concurrency that is lost by requiring that messages be delivered in a global total order, while still providing that high quality of service. While the Totem multiple-ring protocol provides efficient operation over multiple LANs within a local area, it is not intended for operation over a wide-area system, interconnected for example by ATM or the Internet.

As Figure 1 shows, the Totem multiple-ring protocol operates on top of a protocol, such as the Totem single-ring protocol, that provides reliable totally ordered message delivery and membership services within a single LAN. The process group layer operating above the multiple-ring protocol delivers messages to application processes within process groups and maintains the process group membership. The service provided to the application is the same regardless of whether the process group layer is operating over only the Totem single-ring protocol or over both the Totem multiple-ring and single-ring protocols.

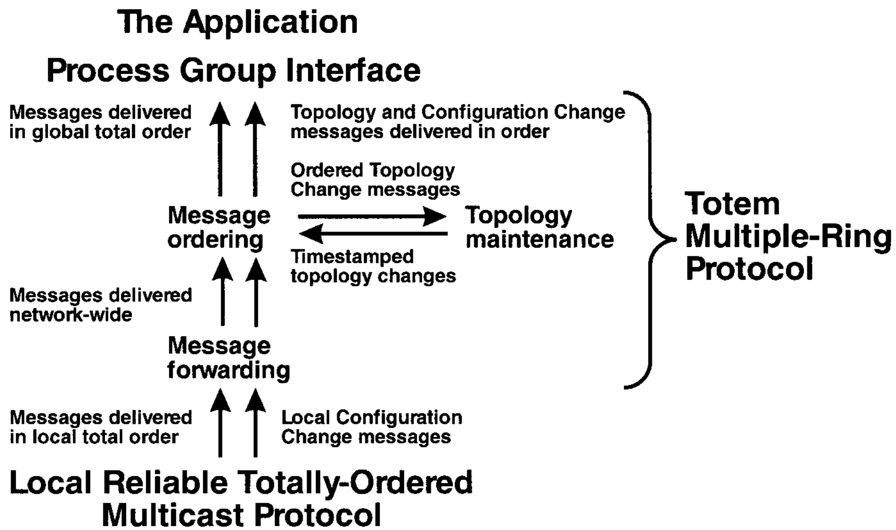


Fig. 1. The Totem multiple-ring protocol hierarchy.

In Moser et al. [1996] we presented an overview of the Totem system, and in Amir et al. [1995] a description of the Totem single-ring ordering and membership protocol. This article focuses on the Totem multiple-ring ordering and topology maintenance protocol. An earlier version of the multiple-ring protocol appeared in Agarwal et al. [1995]; more details, including proofs of correctness, can be found in Agarwal [1994].

2. RELATED WORK

Reliable totally ordered group communication protocols can be classified as symmetric or asymmetric, depending on whether all nodes play the same role or some nodes are distinguished from others.

The most symmetric protocols are those that derive the total order from a causal order. The causal order protocol of the Isis system [Birman and van Renesse 1994], the first widely used group communication system and the first system to provide virtual synchrony, derives the causal order from a vector clock, which is effective for small groups. The Consul protocol [Mishra et al. 1993] and the Trans protocol [Melliar-Smith and Moser 1993; Melliar-Smith et al. 1990] and its derivative, the Lansis protocol of the Transis system [Amir et al. 1992], form the causal order from acknowledgments piggybacked on messages. The Total protocol [Melliar-Smith et al. 1990; Moser et al. 1993] implemented on top of Trans has the interesting characteristic that it continues to order messages even in the presence of faulty processors. The Toto protocol [Dolev et al. 1993] implemented on top of Lansis has a similar characteristic but, unlike the Total protocol, uses additional messages for voting. All of the other totally ordered multicast protocols discussed here block in the presence of processor faults until a

membership algorithm has detected the fault and removed the faulty processor from the configuration.

Clearly asymmetric are the static sequencer protocols, such as the protocol of the Amoeba system [Kaashoek and Tanenbaum 1991], in which messages are transmitted point-to-point to a central sequencer which multicasts them in order. Disadvantages of the central sequencer approach are that all messages are transmitted twice and that the sequencer may be a bottleneck and a single point of failure.

Part way between the symmetric and asymmetric protocols are the rotating sequencer protocols. Here there are two major subclasses, the sequencing acknowledgment protocols and the token protocols.

In the sequencing acknowledgment strategy, first described in Chang and Maxemchuk [1984], processors broadcast messages at will, and unordered. One processor acts as a sequencer. The sequencer determines an order on the messages it has received and broadcasts a message that communicates that order to the other processors. Periodically, the role of the sequencer moves from one processor to the next in a predetermined order. Sequencer-based protocols exhibit low latency at low loads but suffer from flow control problems at high loads. Several reliable totally ordered multicast protocols have been developed using the sequencer approach, including the two similar but distinct RMP protocols of Jia et al. [1996] and of Whetten et al. [1995]. An interesting variation is the Pinwheel protocol of Cristian and Mishra [1995].

The use of a circulating token to sequence ordered multicasts was described in Rajagopalan and McKinley [1989]. This technique has been extensively developed in the Totem single-ring protocol. In these protocols, only the token holder is allowed to broadcast messages and to determine the order of the messages that it broadcasts. Reliable totally ordered multicast protocols based on a token ring can provide high throughput, good flow control, and rapid detection of faults. However, the latency to delivery of a message increases linearly with the size of the ring. Interesting variations of the token-based protocols are the Total protocol of the Horus system described in van Renesse et al. [1996] (quite distinct from the Total protocol of Melliar-Smith et al. [1990] and Moser et al. [1993]), and the On-demand protocol of Alvarez et al. [1998]. In these protocols, the token moves in response to requests for the token, avoiding visits to sites with nothing to transmit so as to reduce the latency. Buffer management is, however, more difficult than if the token is passed in a predetermined fashion.

Also interesting and clearly asymmetric are the Newtop protocol [Ezhilchelvan et al. 1995] and the Hybrid protocol [Rodrigues et al. 1996]. These protocols are designed for systems in which some communication links are much slower than others. Typically, a rotating sequencer circulates through sites connected by high-speed links. Sites connected by low-speed links transmit their messages point-to-point to one of the sequencer sites, where they are ordered and multicast. Both the Newtop protocol and the Hybrid protocol depend on a single set of sites to act as

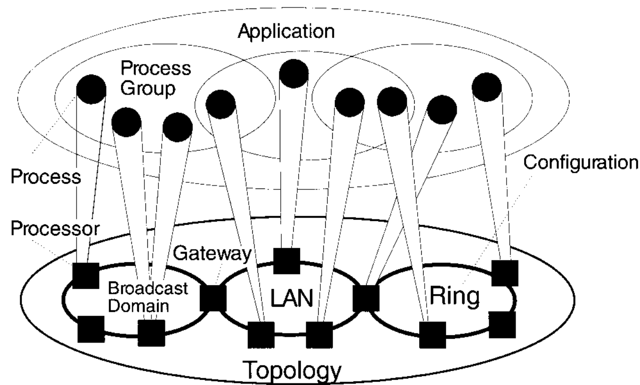


Fig. 2. The elements of the environment in which Totem operates.

sequencers; thus, they are more centralized than the Totem multiple-ring protocol with its arbitrary topology of interconnected LANs.

Total ordering of messages by timestamp is an efficient and elegant approach that depends critically on knowing that all messages with lower timestamps have been received. The Totem multiple-ring strategy obtains this knowledge from a local reliable ordered delivery protocol (potentially any of the protocols cited above).

3. DEFINITIONS AND ASSUMPTIONS

We consider a finite number of LANs that are interconnected by gateways, as shown in Figure 2. Each LAN is a *broadcast domain* consisting of a finite number of processors that communicate by broadcasting messages. Each processor has a unique identifier, and each has stable storage.

Each *gateway* consists of two processors, one on each of the LANs that it connects; communication through a gateway is bidirectional. A gateway forwards messages between rings and maintains the current view of the topology. Other than these functions, a gateway behaves exactly like a processor; in particular, it can send messages from, and deliver messages to, the application. We use the term *processor* to mean either processor or gateway unless explicitly stated otherwise.

Processors may fail and restart, but do not exhibit malicious behavior. When a processor recovers from a failure, it retains the same identifier it had before the failure. The processor may have written all or part of its data into stable storage before it failed. A processor that is excessively slow, or that fails to receive a message an excessive number of times, can be regarded as having failed.

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 between separated components can subsequently be reestablished.

The communication medium does not corrupt messages maliciously. Each broadcast message is received immediately (sufficiently promptly that the broadcast domain does not reorder messages) or not at all by each processor in the broadcast domain, i.e., it may be received by a subset of the processors. It is assumed that a processor receives all of its own broadcast messages and that messages are not dropped within a processor.

Although the broadcast domain does not reorder messages, loss and retransmission of messages may cause a processor to receive messages out of order. Consequently, processors within a broadcast domain execute a protocol that provides reliable totally ordered message delivery and membership services within the broadcast domain. This protocol can be the Totem single-ring protocol [Agarwal 1994; Amir et al. 1995] or any other protocol that provides similar services.

We refer to the set of processors within a broadcast domain that can communicate with each other as a ring or the membership of a ring, and use the term *configuration* to define a particular membership view provided to the application. More specifically, a *ring* is a set of processor identifiers. Each ring has a unique identifier, referred to as the *ring identifier*. Ring identifiers are totally ordered so that each new ring identifier is greater than any ring identifier previously known to the processors in the membership. Each processor is a member of at most one ring at a time; a ring may consist of a single processor. Processor failure and network partitioning result in invocation of the single-ring membership protocol and formation of a new ring.

For the multiple-ring protocol, we use the term topology to mean a set of rings interconnected by gateways such that there is a communication path between any two processors that are members of rings in this set. More specifically, a *topology* is a set of ring identifiers and a set of gateway identifiers that represent a connected component of the network. The *topology identifier* is the lexicographically ordered list of ring identifiers of the rings that comprise the topology. Because ring identifiers are unique, topology identifiers are also unique.

We use the term *originate* to refer to the generation of a message by the application when it is broadcast the first time. We distinguish between receipt and delivery of a message. A message is *received* from the next lower layer in the protocol hierarchy and is *delivered* to the next higher layer. The originator of a message may specify one of two levels of delivery, called *agreed* and *safe*, defined below. Delivery of a message may be delayed to achieve the level of service requested by the originator of the message.

Three types of messages are delivered to the application. *Regular* messages are originated by the application for delivery to the application. A *regular* message is addressed to one or more process groups; the identifiers of those groups are enumerated in the header attached to the message by the process group layer. A *Configuration Change* message, delivered by the single-ring protocol, signals a membership change within a broadcast domain. A *Topology Change* message, delivered by the multiple-ring

protocol, signals a topology change within the larger network. The Configuration Change and Topology Change messages terminate one configuration or topology and initiate another.

3.1 Membership Services

The following membership services are provided by the multiple-ring topology maintenance algorithm. These services are similar to those provided by the single-ring membership algorithm [Agarwal 1994; Amir et al. 1995], except that they define topologies system-wide rather than configurations for a local broadcast domain. The multiple-ring protocol provides the following services:

(1) *Delivery of Topology Change Messages*: Each topology change is signaled by delivery of a Topology Change message by the membership algorithm. The Topology Change message contains a topology identifier and the membership of the new topology.

(2) *Uniqueness of Topologies*: Each topology identifier is unique; moreover, at any time a processor is a member of a ring in at most one topology.

(3) *Termination*: If a topology ceases to exist for any reason, such as processor failure or network partitioning, then every processor that is a member of a ring of that topology will install a new topology, or will fail before doing so.

(4) *Topology Change Consistency*: Processors that are members of rings in the same topology T_2 deliver the same Topology Change 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.

3.2 Reliable Totally Ordered Delivery Services

The multiple-ring ordering algorithm provides the services of reliable totally ordered message delivery across multiple LANs. These services are similar to those provided for the single-ring protocol, except that they now apply to multiple LANs interconnected by gateways.

We define a causal order that is similar to Lamport's definition [Lamport 1978], but that is defined in terms of messages, rather than events, and with respect to a particular topology, rather than across all configurations.

(1) *Causal Order for a Topology*: For a given topology T and for all processors p that are members of T, the causal order for T is the reflexive transitive closure of the "precedes" relation defined as follows:
 —The Topology Change message initiating topology T delivered by p precedes every message originated by p in T.
 —For each message m_1 delivered by p in T and each message m_2 originated by p in T, if m_1 is delivered by p before m_2 is originated, then m_1 precedes m_2.

—For each message m_1 originated by p in T and each message m_2 originated by p in T, if m_1 is originated before m_2, then m_1 precedes m_2.
—Each message delivered by p in T precedes the Topology Change message delivered by p to terminate T.

This definition of causal order allows processors to deliver messages after a network partition by limiting the causal relationships to the topology in which the message was originated. It also allows processors in one component of a partitioned network to deliver messages without having to deliver the messages that are delivered in the other components.

Partitioning of the network can result in different sets of messages being delivered in different components of the network and therefore in different topologies. Moreover, the processors within a topology do not necessarily deliver the same last few messages in that topology. However, each message is delivered by timestamp so that the relative order of any two messages can be established deterministically by the processors that deliver both messages. We define the message delivery order within a topology and across the entire network in the following manner:

(2) *Delivery Order for Topology T*: The reflexive transitive closure of the "precedes" relation for topology T defined on the union over all processors p in T of the sets of regular messages delivered in T by p, as follows:
—Message m_1 precedes message m_2 in T if processor p delivers m_1 in T before p delivers m_2 in T.

Proofs that the Delivery Order for Topology T is a total order can be found in Agarwal [1994]. Again note that, if the network partitions, some processors in topology T may not deliver all messages of the Delivery Order for Topology T.

(3) *Global Delivery Order*: The reflexive transitive closure of the union of the delivery orders for all topologies and of the "precedes" relation defined on the set of Topology Change messages and regular messages, as follows:
—Message m_1 precedes message m_2 if some processor p delivers m_1 before p delivers m_2.

Proofs that the Global Delivery Order is a total order can be found in Agarwal [1994]. The message-ordering services defined below are for all topologies T and for all processors p in T.

(4) *Reliable Delivery for Topology T*:
—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 topology T unless a topology change occurs.
—If processors p and q are both members of consecutive topologies T_1 and T_2, then p and q deliver the same set of messages in T_1 before delivering the Topology Change message that terminates T_1 and initiates T_2.

(5) *Delivery in Causal Order for Topology T*:
—Reliable delivery for topology T.
—If processor p delivers messages m_1 and m_2, and m_1 precedes m_2 in the causal order for topology T, then p delivers m_1 before p delivers m_2.

(6) *Delivery in Agreed Order for Topology T*:
—Delivery in causal order for topology T.
—If processor p delivers message m_2 in topology T and m_1 is any message that precedes m_2 in the Delivery Order for Topology T, then p delivers m_1 in T before p delivers m_2.

(7) *Delivery in Safe Order for Topology T*:
—Delivery in agreed order for topology T.
—If processor p delivers message m in topology T and the originator of m requested safe delivery, then p has determined that every processor in T has received m.

(8) *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 a consistent total order for the topology. When a processor delivers a message in agreed order for a topology, the processor has delivered all preceding messages in the total order for that topology. Moreover, all processors that deliver the message in the topology deliver it at the same point in the total order for the topology.

When a processor delivers a message in safe order for a topology, in addition to meeting the requirements for agreed delivery, the processor must know that all of the other processors in the topology have received the message. The determination of whether a message can be delivered in safe order for a topology is based on the receipt of acknowledgments from the other processors in the topology. Once a processor has acknowledged a message and its predecessors, the processor will deliver the message unless

that processor fails. We cannot require that two processors that deliver a message as safe necessarily deliver it in the same topology because the network may partition. Nevertheless, a processor knows the topology in which it delivered the message as safe, and it knows that each of the other processors in that topology has received the message and will deliver it unless that other processor fails.

Extended virtual synchrony [Moser et al. 1994] ensures that messages are delivered in a consistent total order systemwide, even if the network partitions and remerges or if processors fail and restart with stable storage intact. In contrast, virtual synchrony [Birman and van Renesse 1994] constrains the delivery of messages to processors in a single connected component of the network, the *primary component*, even if processors in the other components have received the messages. Systems based on virtual synchrony must ensure that only the processors in the primary component continue to operate after the network partitions. Processors in the other components are stopped and may deliver messages inconsistently before they are stopped.

4. THE TOTEM MULTIPLE-RING PROTOCOL

The Totem multiple-ring protocol provides agreed and safe delivery of messages across multiple interconnected LANs, as well as topology maintenance services. Delivery of messages in a consistent global total order across a distributed system is relatively straightforward if membership and topology changes do not occur. However, addition and removal of processors, and partitioning and remerging of the network, make the delivery of messages in a consistent global total order more difficult.

4.1 The Total-Ordering Algorithm

We describe here the operation of the multiple-ring total-ordering algorithm without considering topology changes. In Section 4 we consider the difficult task of handling topology changes.

The multiple-ring total-ordering algorithm uses timestamps on messages to create a global total order of messages that respects causality and that is consistent across the entire network. A processor or gateway maintains a Lamport (logical) clock, with which it timestamps messages that it originates, and delivers messages that it has received in timestamp order. A *message-ordering timestamp* consists of the timestamp of the message, a source ring identifier, a message type, and a configuration identifier. Where no ambiguity can arise, we refer to a message-ordering timestamp simply as a timestamp. This simple and efficient algorithm is effective provided that message delivery is reliable and that, when a processor delivers a message, it can determine that it will not subsequently receive a message with a lower timestamp.

Within each individual ring, the single-ring protocol provides reliable totally ordered delivery of messages to processors and gateways. The single-ring protocol uses sequence numbers to deliver messages reliably to

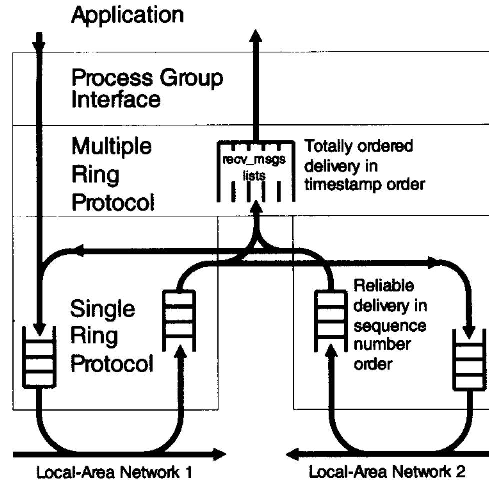


Fig. 3. Gateway message path. The messages broadcast to a directly attached ring by a gateway are messages that were generated by an application process executing at the gateway or messages that were delivered by the single-ring protocol to the multiple-ring protocol and were forwarded by a gateway.

the multiple-ring protocol in sequence number order. If a message needs to be retransmitted, both the sequence number and the timestamp are unchanged. The gateways forward messages from one ring to the next, if necessary, in the order in which the single-ring protocol delivers messages to the gateway, as shown in Figure 3. When a gateway forwards a message from one ring to the next, it gives the message a new sequence number for the new ring, but the timestamp of the message remains unchanged.

A processor or gateway can deliver a message in timestamp order only if it knows that it will not subsequently receive a message with a lower timestamp. Because messages are delivered by the single-ring protocol in order and are forwarded by the gateways in order, if a processor receives a message, it cannot subsequently receive another message that was originated on the source ring of the message with a lower timestamp. Thus, a processor can deliver a message in timestamp order if it has received a message with a higher timestamp from every other ring.

The multiple-ring total-ordering algorithm employs the following local data structures and message types, which it uses to track the messages received from each of the rings and to deliver messages in agreed and safe order.

4.1.1 *Local Data Structures*. Each processor (and gateway) maintains a *ring_table* to record the messages received and to implement total ordering of messages. The *ring_table* contains an entry for each ring in the processor's current topology and an entry for each new ring that is about to be added to the current topology, with the following information:

—*ring_id*: The unique ring identifier generated by the single-ring protocol.

—*recv_msgs*: A list of messages that were originated on the ring and that this processor has received and will deliver but has not yet delivered. This list is sorted in increasing order by timestamp.

—*max_timestamp*: The highest timestamp of a message such that this processor has received all messages originated on the ring with lower timestamps.

—*min_timestamp*: The lowest timestamp of a message in this processor's *recv_msgs* list for the ring. If *recv_msgs* is empty, then *min_timestamp* equals *max_timestamp*.

For each directly attached ring (i.e., ring on which it can itself transmit messages), each processor (and gateway) maintains the following:

—*my_guarantee_vector*: A vector, with an entry for each ring in this processor's current topology, that contains the highest timestamp of a message that this processor received for that ring.

—*my_timestamp*: The highest timestamp of any message known to this processor.

—*my_stable_timestamp*: The value of the timestamp that this processor last wrote to stable storage.

—*timestamp_interval*: A constant that determines how often the timestamp is written to stable storage.

Each processor (and gateway) maintains:

—*cand_msgs*: A list containing the lowest entry in this processor's *recv_msgs* list for each of the rings in its *ring_table*. This list is sorted in increasing order by message-ordering timestamp, i.e., (*timestamp, source ring identifier, message type, conf_id*). If the *recv_msgs* list is empty, then the entry in *cand_msgs* for the ring is (*min_timestamp, ring identifier, regular, 0*).

—*guarantee_array*: An array with a row for each entry in *ring_table* and a column for each ring in *topology*. The entries in a column are timestamps of messages originated on the corresponding rings. The row for the local ring contains entries that are the *max_timestamp* values for the rings. A row for any other ring in the current topology contains the guarantee vector that this processor received from a gateway on that ring. A row for a ring not yet in the current topology contains entries that are the timestamp at which that ring will be added to the topology.

Each gateway maintains:

—*seen_table*: A table, with an entry for each ring known to the gateway that contains the highest timestamp of a message received from that ring together with the type of that message. This table contains information derived from all messages received by the gateway to ensure that no message crosses the gateway more than once.

—*gateway_id*: The identifier of the gateway. The *gateway_id* is chosen deterministically from the two processor identifiers for the gateway.

When a processor or gateway starts, its *my_stable_timestamp* and timestamp in stable storage are initially 0. The value of *timestamp_interval* is determined as a configuration parameter.

When a processor or gateway starts or restarts, it is a member of one ring for each broadcast domain to which it is interfaced. The number of entries of *my_guarantee_vector* is initialized to the number of broadcast domains; each entry of *my_guarantee_vector* has an initial timestamp of –1. The *recv_msgs* lists are empty, and *min_timestamp* and *max_timestamp* are 0.

On receipt of a Configuration Change or Topology Change message introducing a new ring, a processor or gateway adds the data structure for the new ring to the *ring_table*. It obtains the *ring_id* for the new ring from the Configuration Change or Topology Change message, as well as the *max_timestamp* and *min_timestamp*. It sets the *recv_msgs* list for the new ring to empty, and adds an entry for the new ring to the *cand_msgs* list and to the *seen_table*.

4.1.2 *Message Types*.

4.1.2.1 *Regular Message*. 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 timestamp given the message at origination.

—*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 fields *src_sender_id*, *timestamp*, and *src_ring_id* are set by the single-ring protocol on transmission of the message by the processor that generated the message. These fields are not changed when a message is forwarded or retransmitted.

4.1.2.2 *Guarantee Vector Message*. In addition to the fields of a regular message, each Guarantee Vector message contains the following field:

—*guarantee_vector*: The current *my_guarantee_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 on which the Guarantee Vector message is originated and sent. The contents of a Guarantee Vector message indicate which messages have been received from the other rings by the processors and

```
if my_guarantee_vector[msg.src_ring_id] < msg.timestamp then
   my_guarantee_vector[msg.src_ring_id] := msg.timestamp
endif
if msg.timestamp <= ring_table[msg.src_ring_id].max_timestamp then
  discard message
else
  add message to ring_table[msg.src_ring_id].recv_msgs
  ring_table[msg.src_ring_id].max_timestamp := msg.timestamp
  if ring_table[msg.src_ring_id].recv_msgs contains only one message then
    ring_table[msg.src_ring_id].min_timestamp := msg.timestamp
    update entry for msg.src_ring_id in cand_msgs
  endif
  call deliver_msgs
endif
```

Fig. 4. Algorithm executed by a processor or gateway on receipt of a message.

gateways on the ring generating the guarantee vector in the Guarantee Vector message. This allows messages to be delivered in safe order. Guarantee Vector messages are forwarded throughout the network, but are not delivered to the application.

Null messages are transmitted periodically by the gateways to the other gateways and processors in the network and are not delivered to the application. The timestamp of a null message assures a recipient that it will not receive a message with a lower timestamp from the source ring of the null message. This allows messages to be delivered in agreed order, even if few or no messages are generated on the ring.

4.1.3 *The Algorithm.* When a processor receives a message, it sets *my_timestamp* to the larger of *my_timestamp* and the *timestamp* in the message. This ensures that the next new message broadcast by the processor has a higher timestamp than any message that it had previously received. Pseudocode is shown in Figure 4.

Likewise, when a processor receives the token, it sets *my_timestamp* to the larger of *my_timestamp* and the timestamp in the token. When it passes the token on to the next processor around the ring, it sets the timestamp in the token to *my_timestamp*.

When a processor removes a message from the FIFO buffer containing the messages from the application, it increments *my_timestamp* and sets the timestamp of the message to *my_timestamp*. It then broadcasts the message. This ensures that the next new message broadcast by the processor has a higher timestamp than any message previously broadcast by the processor.

To ensure that a processor never transmits regular messages with the same timestamp, even if it fails and subsequently recovers, each processor maintains the value of *my_stable_timestamp* on stable storage. If a processor increments *my_timestamp* to a value greater than or equal to *my_stable_timestamp* + *timestamp_interval*, it immediately sets *my_stable-*

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

Fig. 5. Algorithm executed by a processor or gateway to deliver a message.

_timestamp to *my_timestamp* and writes the value of *my_stable_timestamp* to stable storage before taking further action. Thus, a processor never broadcasts a message with a timestamp greater than *my_stable_timestamp* + *timestamp_interval*. Although processing is momentarily delayed while the value of *my_stable_timestamp* in stable storage is updated, the value of *timestamp_interval* can be quite large so that the overall performance degradation is small.

When a processor starts or restarts after a failure, it first reads the value of *my_stable_timestamp* from stable storage. It then sets *my_stable_timestamp* to *my_stable_timestamp* + *timestamp_interval*. The processor then writes the value of *my_stable_timestamp* to stable storage and waits for the completion of that write. Finally, it sets *my_timestamp* to *my_stable_timestamp*. Thus, if a processor fails and recovers, the timestamp of the first message it broadcasts after recovery is greater than the value of *my_stable_timestamp* (obtained from stable storage) + *timestamp_interval*. This ensures that a processor never broadcasts two regular messages with the same timestamp even if it fails and recovers.

The delivery of messages in agreed and safe order is discussed below, and pseudocode is shown in Figure 5.

4.1.3.1 *Delivery of Messages in Agreed Order.* The key to agreed delivery is that messages originated on a ring are forwarded by the gateway through the network in the order in which they are received from the single-ring protocol. When a message is forwarded, it is given a new sequence number for the ring onto which it is forwarded, but retains its old timestamp. On each ring, the single-ring protocol delivers messages to the multiple-ring protocol in sequence number order. The multiple-ring protocol places messages in the *ring_table* and delivers them to the application in message timestamp order, i.e., (*timestamp*, *src_ring_id*, *message type*, *conf_id*).

Because the messages generated on any one ring are forwarded in the order of their sequence numbers, each processor can record a *max_timestamp* for each ring in its *ring_table*, which indicates that it has received all messages from that ring with lower timestamps. If a gateway receives a

message with a timestamp less than the *max_timestamp* of the source ring of that message, it discards the message as a redundant message. This mechanism allows a processor to identify redundant copies of messages forwarded by different gateways.

A processor can deliver a message in agreed order only if it has delivered all messages with lower timestamps. To ensure that messages are delivered in agreed order, even if some messages are delayed, a processor first determines the lowest entry in *cand_msgs*. If the *recv_msgs* list for the ring with the lowest entry in *cand_msgs* is empty, the processor can deliver no further messages until it has received a message from that ring, because the next message from that ring might have been delayed and may have a lower timestamp than the messages that it has received from the other rings. If the lowest entry in *cand_msgs* corresponds to a message for which agreed delivery was requested, the processor can deliver the message.

4.1.3.2 *Delivery of Messages in Safe Order*. In addition to the requirements for agreed delivery, safe delivery requires information about whether the message has been received by all of the processors that require them. When a processor executing the single-ring protocol delivers a message in safe order, all of the processors on its local ring have received the message.

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

A gateway periodically transmits Guarantee Vector messages containing its *my_guarantee_vector*. When a processor executing the multiple-ring protocol receives a Guarantee Vector message, it compares the *guarantee_vector* in the message with the appropriate row of its *guarantee_array* and changes an entry of the row to the corresponding *guarantee_vector* entry if the vector entry contains a higher timestamp.

To deliver a message in safe order, a processor must wait until all of the entries in the column of its *guarantee_array*, corresponding to the ring on which the message was originated, contain timestamps greater than or equal to the timestamp of the message. This guarantees (1) that the message has been received by each processor that should have received it and (2) that the message will be delivered by that processor unless that processor fails. Gathering the additional knowledge required for delivery of a message in safe order may delay delivery of messages with higher timestamps.

4.1.3.3 *Forwarding of Messages*. A gateway forwards messages, as it receives them in agreed order, from one LAN to the next. Whenever it forwards a message, the gateway updates the *last_timestamp* and *message_type* fields for the row corresponding to the *src_ring_id* of the message, in the *seen_table*. It forwards only those messages that have a greater

```
if msg.timestamp > seen_table[msg.src_ring_id].timestamp then
  seen_table[msg.src_ring_id].timestamp = msg.timestamp
  forward message
endif
```

Fig. 6. Algorithm executed by a gateway for the forwarding of messages.

message-ordering timestamp than the last forwarded message from the same *src_ring_id*. This ensures that a message that has crossed the gateway in one direction does not come back in the opposite direction. The pseudocode for the forwarding of messages is shown in Figure 6.

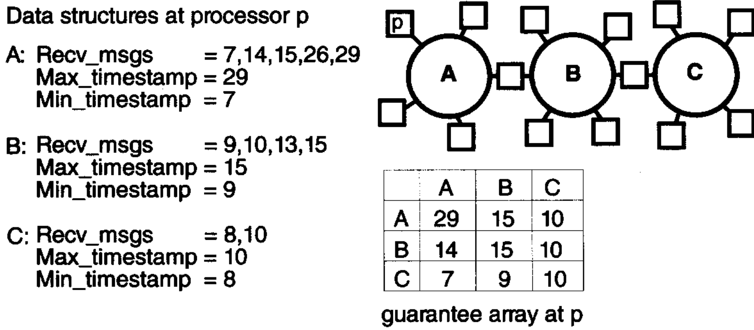
Example. Consider the network shown in Figure 7, where the rings are represented by circles and the processors and gateways by squares. A processor p on ring A is ordering messages from rings A, B, and C. The first row of the *guarantee_array* at processor p is the vector of *max_timestamp* values at p. The second and third rows are the *guarantee_vectors* in the Guarantee Vector messages for rings B and C.

Processor p can deliver in agreed order the message with timestamp 7 from ring A, the messages with timestamps 9 and 10 from ring B, and the messages with timestamps 8 and 10 from ring C. After those messages have been delivered, the *min_timestamp* and *max_timestamp* at processor p for ring C will be set to 10 until it receives further messages from ring C. The undelivered message with the lowest timestamp is the message from ring B with timestamp 13, but p cannot deliver that message until it receives the next message from ring C. Otherwise, p may subsequently receive a message from ring C with timestamp 12.

If processor p receives the message from ring A with timestamp 7 that contains a request for safe delivery, then p can deliver that message as safe because the column of the *guarantee_array* corresponding to ring A has all entries at least equal to 7, which indicates that the message is safe on all rings in the current topology. The same is true for the message from ring B with timestamp 9 and the message from ring C with timestamp 8. Processor p cannot, however, deliver the message from ring B with timestamp 10 as safe because the guarantee vector from ring C reports receipt of messages from B only up through timestamp 9.

4.2 The Topology Maintenance Algorithm

The total-ordering algorithm described above depends on knowledge of the topology of the interconnected rings. If a new ring becomes connected to processor p, p must be informed so that it can add the new ring to its topology. Moreover, p must wait for messages from the new ring; otherwise, p will prematurely deliver messages with higher timestamps. Similarly, if a ring becomes disconnected from p, p must be informed so that it can



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

Fig. 7. An example of message ordering. The rings are indicated by circles and the processors and gateways by 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.

delete that ring from its topology; otherwise, p will wait for messages from that ring and stop delivering messages.

When a topology change occurs, it is important that the effect of the topology change be consistent throughout the set of processors that were previously able to, and can still, communicate with each other. Even though the processors learn of the topology change at different physical times, they must agree on the same logical time for the topology change, and must agree on the sets of messages to be delivered before and after the topology change. Local Configuration Change messages and global Topology Change messages are timestamped and are delivered in timestamp order along with the regular messages.

Because it takes time for a message to traverse the network, it is possible that a message will be affected by a fault reported by a Configuration Change message whose timestamp is greater than the message timestamp. If a gateway becomes faulty, rings on the far side of the gateway may become inaccessible. For each such ring, there is a last message that was received from that ring before the fault. No message with a larger timestamp can be received from that ring, until connectivity is restored. Consequently, to ensure a consistent view of the topology and to prevent blocking of the message-ordering algorithm, for each such ring, the topology maintenance algorithm generates a Topology Change message reporting the disconnection of the ring at that timestamp.

The multiple-ring topology maintenance algorithm employs the following local data structures and message types.

4.2.1 *Local Data Structures.* Each processor and gateway maintains:

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

Each gateway maintains:

—*topology*: The gateway's view of the current topology, 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, which is initialized to −1 when the edge is first added to *topology*. When a configuration change results in deletion of a gateway, this timestamp is changed to that of the Configuration Change message, marking that node for deletion and indicating when it will be deleted from the graph.

4.2.2 *Message Types*.

4.2.2.1 *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 a gateway after the gateway has forwarded all of the messages from the old ring. The Configuration Change message is delivered as an agreed message by the single-ring protocol to the multiple-ring protocol and by the multiple-ring protocol to the process group layer. It contains the following fields:

—*timestamp*: The highest timestamp of a message delivered by a processor in the new single-ring membership prior to the Configuration Change message.

—*src_ring id*: The identifier of the new ring.

—*type*: Configuration Change.

—*conf_id*: The identifier of the old ring.

—*memb_list*: A list of the processor identifiers of the membership of the new configuration.

—*gateways*: A vector containing an entry for each processor in *memb_list*. An entry contains a 1 if the associated processor is a gateway and a 0 otherwise.

—*gateway_ids*: A list containing the *gateway_id* of each gateway on the new ring.

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

4.2.2.2 *Network Topology Message*. A Network Topology message is transmitted by a gateway on a directly attached ring that experienced a configuration change. The Network Topology message is not delivered to the application and is not forwarded by the multiple-ring protocol. It informs the other gateways and processors on the ring about the part of the network connected to the ring by the gateway. The Network Topology message is necessary when the components of a partitioned network remerge or when gateways or processors are added to the ring. It contains the following fields:

—*timestamp*: The timestamp of the associated Configuration Change message.

—*src_sender_id*: The processor identifier (on the ring that experienced the configuration change) of the gateway that originated the message.

—*type*: Network Topology.

—*gateway_id*: The processor identifier (on the ring that did not experience the topology change) of the gateway that originated the message.

—*topology*: The gateway's current view of the topology including all rings to which it is connected, except the ring that experienced the configuration change.

4.2.2.3 *Topology Change Message*. A Topology Change message is transmitted 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 total order along with the regular messages. A Topology Change message is also generated and delivered locally by a processor (not a gateway) when it receives a Configuration Change message for its own 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 transmitted with a request for agreed delivery. It contains the following fields:

—*timestamp*: The *timestamp* of the corresponding Configuration Change message if the Topology Change message adds one or more rings; otherwise, the *max_timestamp* of the ring to be deleted.

—*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 consists of 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 that ring.

—*type*: Topology Change.

—*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 has been received from that ring.

—*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 first four fields constitute the identifier of the Topology Change message.

Messages are delivered into the global total order defined by the lexicographical order on message-ordering timestamps, i.e., (*timestamp*, *src-*

```
if ring_table[msg.src_ring_id] exists then
  discard message
  return
endif
if amgateway then
  forward Configuration Change message
endif
add entry for msg.src_ring_id to ring_table
   ring_table[msg.src_ring_id].recv_msgs := empty list
   ring_table[msg.src_ring_id].max_timestamp := msg.timestamp
   ring_table[msg.src_ring_id].min_timestamp := msg.timestamp
add msg to ring_table[msg.src_ring_id].recv_msgs
add row for new ring to guarantee_array with entries equal to msg.timestamp
if msg.src_ring_id = ring_id of directly connected ring then
  update neighbors with msg.gateway_ids
  mark all new entries in neighbors with msg.timestamp
  copy my_guarantee_vector for msg.conf_id into gvmsg.guarantee_vector
  if amgateway then
    for each gateway_id in msg.gateway_ids do
      if gateway_id is in current topology then
        mark gateway in topology for deletion at msg.timestamp
      endif
    endfor
  endif
endif
update guarantee_array row for msg.conf_id with gvmsg.guarantee_vector
```

Fig. 8. Algorithm executed by a processor or gateway on receipt of a Configuration Change message and an associated Guarantee Vector message.

_ring_id, *type*, *conf_id*). The timestamps are integers with the usual ordering; the ring identifiers *src_ring_id* and *conf_id* are lexicographically ordered pairs (*ring_seq*, *rep_id*), and the message types are ordered by the relation: Regular < Configuration Change < Topology Change.

4.2.3 *The Events of the Topology Maintenance Algorithm.* There are five topology events:

—*Receipt of a Configuration Change Message*: The Configuration Change message is created by the single-ring membership algorithm when a failed processor is removed from the configuration, a new or recovered processor is added, the network partitions, or the components of a partitioned network remerge. On receipt of a Configuration Change message, a processor adds a data structure for the new ring to the *ring_table* and the *guarantee_array*, and inserts the Configuration Change message into the *recv_msgs* list corresponding to the source ring of the message. If a gateway receives a Configuration Change message indicating that a ring has become disconnected, the gateway transmits a Topology Change message deleting that ring. Pseudocode is shown in Figure 8.

—*Delivery of a Configuration Change Message*: To handle remerging of a partitioned network, a gateway directly attached to a ring that experi-

```
if msg.src_ring_id ≠ directly connected ring ring_id then
  wait for Topology Change message
else
  if amgateway then
    for each gateway_id in msg.gateway_ids do
      if gateway_id is in topology then
        delete gateway from topology
      endif
    endfor
    send Network Topology message
  endif
  if have neighboring gateways then
    collect Network Topology messages from all gateways in msg.gateway_ids
  else
    add msg.src_ring_id to Topology Change message.new_rings
    insert msg.conf_id in Topology Change message.del_rings
    if amgateway then
      insert added gateways in Topology Change message.new_gateways
      send Topology Change message
    endif
  endif
  deliver Configuration Change and Topology Change messages
endif
```

Fig. 9. Algorithm executed by a processor or gateway when a Configuration Change message is the lowest entry in *cand_msgs*, i.e., the next message to be delivered.

enced a configuration change exchanges topology information with the other processors and gateways on that ring by transmitting a Network Topology message. The Network Topology message describes the component of the network that is connected to the ring by the gateway, based on the gateway's current topology and on the Configuration Change message. Pseudocode is shown in Figure 9.

—*Receipt of a Network Topology Message*: When a gateway has received Network Topology messages from all of the other gateways on the newly formed ring, it transmits a Topology Change message. The Topology Change message serves to inform the other processors and gateways in the network of the change in the topology. Pseudocode is shown in Figure 10.

—*Receipt of a Topology Change Message*: When a processor or gateway receives a Topology Change message that it has not previously received, it accepts the message, inserts the message into the *recv_msgs* list corresponding to the source of the message, and forwards the message to the rest of the network. Duplicate Topology Change messages are discarded. Pseudocode is shown in Figure 11.

—*Delivery of a Topology Change Message*: When the Topology Change message is delivered, the local view of the topology is updated. The Topology Change messages are delivered in total order along with the regular messages and, thus, are delivered by the processors and gate-

```
store as neighbor topology
if have Network Topology msgs from all gateways on new ring then
  combine neighbor topologies to determine rings in network
  for each ring_id in new rings not already in ring_table do
    add ring_id to Topology Change message.new_rings
  endfor
  insert deleted rings in Topology Change message.del_rings
  discard neighbor topologies
  if amgateway then
    insert added gateways in Topology Change message.new_gateways
    send Topology Change message
  endif
  add Topology Change message to ring_table[msg.src_ring_id].recv_msgs
endif
```

Fig. 10. Algorithm executed on receipt of a Network Topology message by a processor or gateway.

```
if msg.timestamp < ring_table[msg.src_ring_id].min_timestamp or
    msg is already in ring_table[msg.src_ring_id].recv_msgs then
  discard msg
  return
endif
if amgateway then
  forward msg
  for each gateway identifier in msg.new_gateways do
    if gateway is in topology then timestamp := msg.timestamp
  endfor
endif
for each ring_id in msg.new_rings not already in ring_table do
  add entry to ring_table for ring_id
    ring_table[ring_id].recv_msgs := empty list
    ring_table[ring_id].max_timestamp := msg.timestamp
    ring_table[ring_id].min_timestamp := msg.timestamp
  add row for new ring to guarantee_array with entries equal to msg.timestamp
endfor
add msg to ring_table[msg.src_ring_id].recv_msgs
ring_table[msg.src_ring_id].max_timestamp := msg.timestamp
```

Fig. 11. Algorithm executed by a processor or gateway on receipt of a Topology Change message.

ways in a consistent global total order. The topology information at the gateways is thus updated consistently. Pseudocode is shown in Figure 12.

4.2.4 *Handling a Single Topology Change.* First we describe the steps taken by a processor to handle a single topology change without considering further topology changes during execution of the topology maintenance algorithm. Receipt of a Configuration Change message generated by the single-ring protocol signals a topology change. The gateways are responsible for determination of the topology change caused by the configuration change and for dissemination of this information to the other processors

```
for each ring_id in msg.del_rings do
  delete ring_table[ring_id]
  if amgateway then
    delete ring_id and connected gateways from topology
  endif
  delete column and row for ring_id from guarantee_array
  delete entry for ring_id from my_guarantee_vector
endfor
for each ring_id in msg.new_rings do
  if amgateway then
    add ring to topology
  endif
endfor
if amgateway then
  for each gateway_id in msg.new_gateways do
    delete gateway_id from topology
    add gateway_id as edge in topology
  endfor
endif
deliver Topology Change message
```

Fig. 12. Algorithm executed by a processor or gateway on delivery of a Topology Change message.

and gateways in the network. A gateway determines the new topology by exchanging topology information with the other gateways on the ring. The actions taken by the processors and gateways when a topology change occurs are described in more detail below.

4.2.4.1 *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 membership. In the total order of messages, the Configuration Change message precedes all messages originated on the new ring. In essence, the Configuration Change message places a marker in the message order for the topology change.

On receipt of a Configuration Change message, a processor sets its *max_timestamp* for the old ring to the timestamp of the Configuration Change message. No messages with higher timestamps will subsequently be received from the old ring. Advancing *max_timestamp* ensures that message ordering will not be obstructed by the lack of such messages.

A processor adds an entry for the new ring to its *ring_table* with a *min_timestamp* and *max_timestamp* equal to the timestamp of the Configuration Change message. It also adds a row to the *guarantee_array* for the new ring and sets all entries in that row to the timestamp of the Configuration Change message. These steps ensure that the new data structures are not subsequently set to inappropriate values that could delay or even deadlock message ordering. The processor then places the Configuration Change message in the *recv_msgs* list for the new ring. The processor also updates *neighbors* using the *gateway_ids* of the Configuration Change message.

A gateway also updates the edges in its *topology*; for each gateway identifier in *gateway_ids*, the gateway marks the edge for that gateway with the timestamp of the Configuration Change message. The edge is thus known to connect the two rings as of the timestamp of the Configuration Change message. The timestamps on the edges are used to determine the rings that must be deleted from the topology to allow delivery of the Configuration Change message. A gateway delays updating its *topology* until the Configuration Change message is the lowest entry in *cand_msgs*, i.e., the next message to be delivered. This ensures that the topology change occurs at the same logical time at all of the processors and gateways.

A gateway transmits 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 has delivered all of the messages in *recv_msgs* for that ring. The ring can then be deleted because messages are forwarded in order, and all messages that should have been forwarded from the ring were forwarded ahead of the Configuration Change message that indicated the disconnection.

A Topology Change message to delete a ring has a *timestamp* equal to the *max_timestamp* of the ring to be deleted, a *src_ring_id* equal to the *ring_id* of that ring, and contents indicating that the ring is to be deleted.

4.2.4.2 *Delivery of a Configuration Change Message*. When a Configuration Change message is the next message to be delivered, each gateway on the ring that experienced the configuration change transmits a Network Topology message on that ring. The Network Topology message indicates the gateway's connected topology, excluding the old ring and the gateways connected to it. The gateway waits to transmit the Network Topology message until the Configuration Change message is the next message to be delivered, to ensure that its *topology* has been updated to the timestamp of the Configuration Change message. It does not forward the Network Topology message.

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 identifier as the Configuration Change message (or Network Topology messages from all gateways on the ring with the same timestamp as the Configuration Change message) and has generated a Topology Change message.

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

When a gateway has received Network Topology messages from all of the gateways on the new ring, it merges the topologies in the messages into its *topology*. For each ring or gateway newly added to its *topology*, it records

that ring or gateway in the fields *new_rings* or *new_gateways* of a Topology Change message. It 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 transmits the Topology Change message, which has a timestamp equal to the timestamp of the Configuration Change message (and of the Network Topology messages) and source ring identifier equal to the ring identifier of the new ring.

4.2.4.4 *Receipt of a Topology Change Message*. When a processor or gateway receives a Topology Change message, it adds the message to the *recv_msgs* list corresponding to the source ring of the message, unless the message is a duplicate in which case it discards the message. If a processor or gateway accepts the Topology Change message, it adds data structures to its *ring_table* and *guarantee_array* for the previously unknown rings in the *new_rings* list of the Topology Change message. Each ring is added to the *ring_table* with an empty *recv_msgs* list. *Min_timestamp*, *max_time-stamp*, and the entries of the row are set to the timestamp of the Topology Change message. Each ring is also added to the *seen_table*.

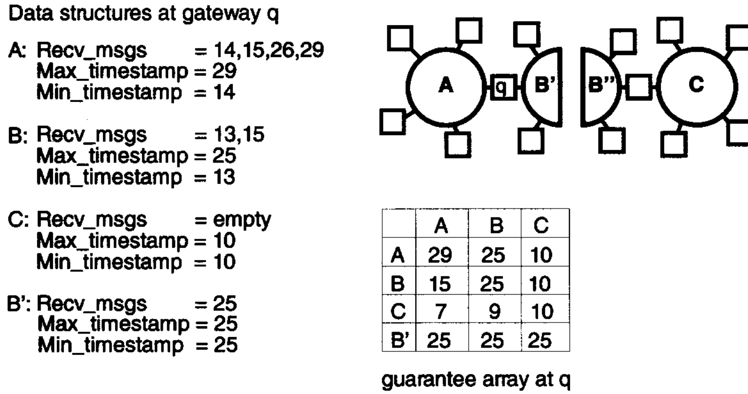
A gateway also marks the edges corresponding to *new_gateways* in its *topology*, with the timestamp of the Topology Change message indicating that those edges will be deleted from the topology at that timestamp. The gateway then forwards the Topology Change message to the rest of the network.

4.2.4.5 *Delivery of a Topology Change Message*. When a Topology Change message is the next message to be delivered, a processor or gateway deletes the entries corresponding to the rings in *del_rings* from its *guarantee_array*, *my_guarantee_vector*s, and *ring_table*. It adds a row to the *guarantee_array* for each ring in *new_rings*, and initializes the time-stamp for each entry in that row to the timestamp of the Topology Change message.

A gateway also replaces the existing gateways in its *topology* with the gateways in *new_gateways*, adds the rings in *new_rings* to its *topology*, and deletes the rings in *del_rings* from its *topology*.

After a processor or gateway has completed processing a Topology Change message, it delivers the message to the application.

4.2.4.6 *Message Ordering during a Topology Change*. It is essential that a newly connected gateway should forward only those messages that have timestamps greater than the timestamp of the Configuration Change message that added the gateway to the topology. Thus, when a Configuration Change message for a directly attached ring is pending (received but not delivered), a processor discards (does not place in its *recv_msgs* lists) any message that has a timestamp less than the timestamp of the Configuration Change message and that was forwarded onto the ring by a new gateway. A processor or gateway buffers any message that has a timestamp greater than the timestamp of the Configuration Change message and that was broadcast by a new gateway, until after it has delivered the Configu-



	A	B	C
A	29	25	10
B	15	25	10
C	7	9	10
B'	25	25	25

Fig. 13. An example of network partitioning, where ring B has partitioned into rings B' and B''. The rings are indicated by circles and the processors by squares. The data structures at gateway q are also shown.

ration Change message. Once it has delivered the Configuration Change message and the associated Topology Change message, it adds the buffered messages to its *recv_msgs* lists. A gateway also forwards those messages in order.

To deliver a message in safe order, a processor or gateway must wait until it knows that all processors and gateways in its current topology have received the message. If the network has partitioned and the processor or gateway does not know that the message has been received by all of the processors on a disconnected ring (because it has not received a Guarantee Vector message from that ring), the processor or gateway must delete that ring from the topology before it can deliver the message in safe order.

4.2.5 *Example.* Returning to the example in Figure 7, consider what happens if ring B partitions into rings B' and B'', as shown in Figure 13. The Configuration Change message generated by the processors on ring B' has timestamp 25. When a processor on ring A or ring B' receives this Configuration Change message, it adds an entry for B' to its *ring_table* and a row for B' to its *guarantee_array*. The setting of *max_timestamp*, *min_-timestamp*, and the *guarantee_array_entries* for B' to 25 ensures that these new entries do not obstruct the delivery of messages with lower time-stamps. The data structures at gateway q, after it has received the Configuration Change message, are shown in Figure 13.

When gateway q receives the Configuration Change message, it adds the message to its *recv_msgs* list for ring B' and increases the *max_timestamp* for ring B to 25 and increases the corresponding entry in the row for ring B in the *guarantee_array*. It then forwards the Configuration Change message onto ring A. It cannot yet deliver the Configuration Change message, because it has not yet received messages beyond timestamp 10 from ring C. It determines whether ring C would still be reachable, were it to delete

from the topology all of the gateways marked for deletion. Since such deletions would leave C unreachable, q deletes ring C from its topology. This is necessary because q may never receive messages from C with timestamps greater than 10, as there is no forwarding path for those messages. Thus, q transmits a Topology Change message indicating that ring C should be deleted at timestamp 10. When the processors on rings A and B' deliver this Topology Change message, they delete ring C. This allows those processors and gateway q to deliver messages beyond timestamp 10, in particular the message with timestamp 13 from ring B, the message with timestamp 14 from ring A, the messages with timestamp 15 from rings A and B, and the Configuration Change message with timestamp 25.

When the Configuration Change message is the lowest entry in *cand-_msgs* and thus ready to be delivered, gateway q transmits a Network Topology message with timestamp 25 on ring B'. The message contains the part of the topology to the left of gateway q, in this case just ring A. If there were new processors on ring B', this would also inform them of that part of the topology. Because q is the only gateway on ring B', it does not wait for additional Network Topology messages from other gateways on ring B'. Instead, q transmits immediately on ring A a Topology Change message with timestamp 25, indicating the addition of ring B', the addition of gateway q, and the deletion of ring B.

The Configuration Change message initiating ring B' informs the processors on ring B' of the gateways on B'. When such a processor has received Network Topology messages from all of the gateways on ring B', in this case just q, it can determine that the current topology consists of ring A and the new ring B'.

When a processor on ring A or ring B' delivers the Topology Change message, it deletes the entry for ring B from its *ring_table*. In Figure 13 all entries in the row of the *guarantee_array* for ring B' are set to 25 because no messages have yet been transmitted on that ring. The column for ring B' will be inserted when the first message originated on ring B' is received.

4.2.6 *Handling Multiple Concurrent Topology Changes*. The multiple-ring membership algorithm does not have control over the order or timing of topology changes. Topology Change messages and further Configuration Change messages may arrive before the current Configuration Change message is delivered. While a processor is waiting for a Network Topology message from a gateway, the processor may receive a Configuration Change message indicating that the gateway is no longer present on the ring. Processing must nevertheless proceed, even though the Network Topology message may never be received. We now describe the handling of further Configuration Change messages that arrive before a pending Configuration Change message has been delivered. The rules are individually

simple and evident, but in combination their consequences are quite complex.

4.2.6.1 *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 has a Configuration Change message pending, a processor may now need fewer Network Topology messages for the prior Configuration Change message, because some gateways may no longer be connected to the ring introduced by the most recent Configuration Change message.

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

4.2.6.3 *Message Ordering*. If there are multiple Configuration Change messages pending for a single ring, a processor buffers all of the messages forwarded by new gateways. It removes a message from the buffer when it delivers the Configuration Change message that adds the new gateway that forwarded the message. Messages with timestamps less than that of the Configuration Change message are discarded.

A processor uses the pending Configuration Change messages to determine whether the ring corresponding to the lowest entry in *cand_msgs* has become disconnected so that it will accept no further messages from that ring. If the lowest entry in *cand_msgs* requested safe delivery, the processor uses the pending Configuration Change messages to determine whether a ring that did not yet guarantee the message as safe has become disconnected. Otherwise, the ordering of messages proceeds as in the case of a single topology change.

4.2.7 *Example*. We now illustrate how the multiple-ring protocol maintains consistent message ordering despite multiple concurrent topology changes. Consider the network in Figure 14(i), where there are six rings A, B, C, D, E, and F with five gateways a, b, c, d, and e. The first change is the merging of rings A and F into the new ring A'. The Configuration Change message generated by the processors on ring A' reports the configuration change and has timestamp 120 (Figure 14(ii)). The second change is the partitioning of ring B into rings B' and B''. Two Configuration Change messages are generated, one by the processors on ring B' with timestamp 125 (Figure 14(iii)) and the other by the processors on ring B''

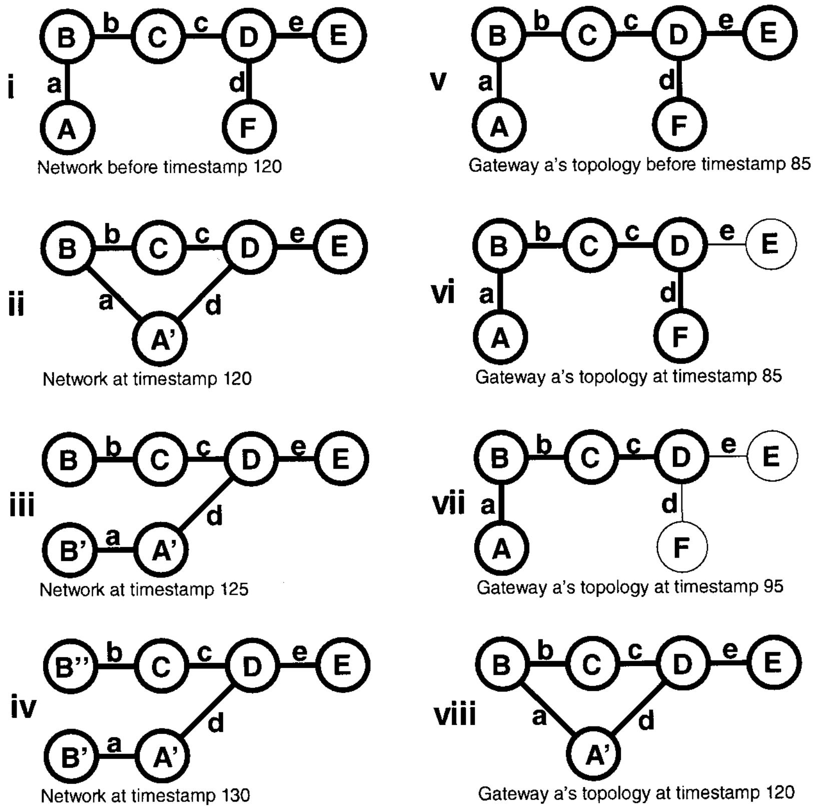


Fig. 14. An example of deletion of rings. The rings are indicated by circles and the gateways by line segments. At timestamp 120, rings A and F merge and become ring A'. At timestamp 125, ring B' is formed when ring B partitions, and at timestamp 130 ring B'' is formed by the remaining processors from ring B. At timestamp 125, gateway a's *recv_msgs* list for ring E is empty, and its *max timestamp* for ring E is 85. Gateway a then deletes ring E from its *topology* at timestamp 85, and ring F at timestamp 90. Gateway a deletes ring A and adds rings A' and E at timestamp 120.

with timestamp 130 (Figure 14(iv)). The gateways forward these Configuration Change messages to the other processors in the network.

In this example, the messages from the processors on ring E are originally forwarded to gateway a via ring B. At timestamp 120, there is an additional forwarding path to gateway a from ring E via ring A'. At timestamp 125, there is only one path to gateway a from ring E via ring A'. Surprisingly, even though there is a continuous connection between gateway a and ring E, it is nevertheless essential for gateway a to delete ring E temporarily from its topology, as shown at the right of Figure 14 and explained below.

The guarantee vectors at gateway a , after it has received a Configuration Change message with timestamp 125 for ring B', are shown in Figure

	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

Fig. 15. Example guarantee vectors. Some of the guarantee vectors at gateway a after it has received the Configuration Change message with timestamp 125, indicating the formation of ring B' (Figure14(iii)).

15. Because gateway a has not yet delivered the Configuration Change messages for rings A' and B', it has not yet changed its view of the topology. Gateway a buffers the messages forwarded by gateway d to ring A' with timestamps greater than 120. It does not forward or add the buffered messages to its *recv_msgs* lists until its ordering timestamp reaches 120 and it has delivered the Configuration Change and Topology Change messages for ring A'. Gateway a discards messages forwarded by gateway d to ring A' with timestamps less than 120. This is essential for correct ordering of messages. If a were to forward messages with timestamp less than 120 onto ring B, those messages might reach ring B before messages traveling over the previous path through ring C. In particular, they might arrive before a message with a lower timestamp and, thus, might be delivered by a processor on ring B before that processor became aware that the message with the lower timestamp exists.

In this example, we assume that, at timestamp 125, gateway a's *recv_msgs* list for ring E is empty and that its *max_timestamp* for ring E is 85, indicating that 85 was the timestamp of the last message that it received from ring E. Gateway a's view of the topology prior to timestamp 85 is shown in Figure 14(v). When it receives the Configuration Change message for ring B' with timestamp 125, gateway a knows that it will no longer receive messages from ring E via gateway b. Gateway a will not receive messages from ring E via gateway d with timestamp less than 120, because gateways a and d were not connected by ring A' prior to timestamp 120. The processors on ring E may have broadcast messages with timestamps between 85 and 120, but gateway a will never receive those messages. Consequently, gateway a must delete ring E from its topology at timestamp 85 (Figure 14(vi)) to ensure that it will be able to deliver messages beyond timestamp 85 and to avoid inconsistencies.

To inform the other processors (Figure 14(iii)) of the deletion, gateway a transmits a Topology Change message with timestamp 85, stating that ring E should be deleted. Processors on ring B', and those on ring A' that were formerly on ring A, will order this Topology Change message, but proces-

sors on ring A' that were formerly on ring F (including gateway d) will discard it, because the timestamp 85 of the Topology Change message is less than the timestamp 120 at which rings A and F merged to form ring A'. Similarly, when gateway a's ordering timestamp reaches 90, it transmits a Topology Change message deleting ring F (Figure 14(vii)). It does not delete rings B, C, and D at this timestamp because its *recv_msgs* lists for those rings contain messages up through timestamp 120.

Gateway a waits until its ordering timestamp has reached 120 and then transmits a Network Topology message on ring A', indicating that its current topology outside of A' consists of rings B, C, and D. When gateway a receives the Network Topology message with timestamp 120 from gateway d, it is informed of the new connection to rings B, C, D, and E via gateway d, and it adds ring E back into the topology. Gateway a then transmits a Topology Change message with timestamp 120 (Figure 14(viii)), indicating deletion of ring A and addition of rings A' and E. When a processor or gateway orders this Topology Change message, it can proceed to order messages beyond timestamp 120.

5. IMPLEMENTATION AND PERFORMANCE

The code for the Totem multiple-ring protocol has been written in the C programming language and runs in user space on Sun workstations, running SunOS 4.x and Solaris 2.x, on 10 and 100 Mbit/sec Ethernets. The code uses standard System V calls, standard network interfaces, and does not preempt the regular process scheduling of the operating system. The code of Totem is easily ported to other platforms.

5.1 Performance Measurements

The performance measurements given here are for Totem running on Sun 1170 and Sun 2200 workstations, running Solaris 2.5.1, on 100 Mbit/sec Ethernets. The measurements were taken for a single ring having nine processors, a two-ring topology having four processors on each ring with a gateway between the two rings, and a three-ring topology having three processors on each ring with two gateways (a gateway between the first and second rings and a gateway between the second and third rings). For each topology, the processors are Sun 1170s, and the gateways are Sun 2200s. All of the processors generate and receive messages; the gateways receive and forward, but do not generate, messages. To represent process group locality and selective forwarding of messages, the system has an additional parameter, the *forwarding probability*. At each gateway, as each message is considered, it is forwarded to the next ring with this probability.

For the measurements, a message size of 1KB was used; the message size is the actual payload in an Ethernet packet, and does not include either the 46 bytes of UDP/IP/Ethernet packet header or the 68 bytes of Totem header. All of the performance results are based on the transmission of

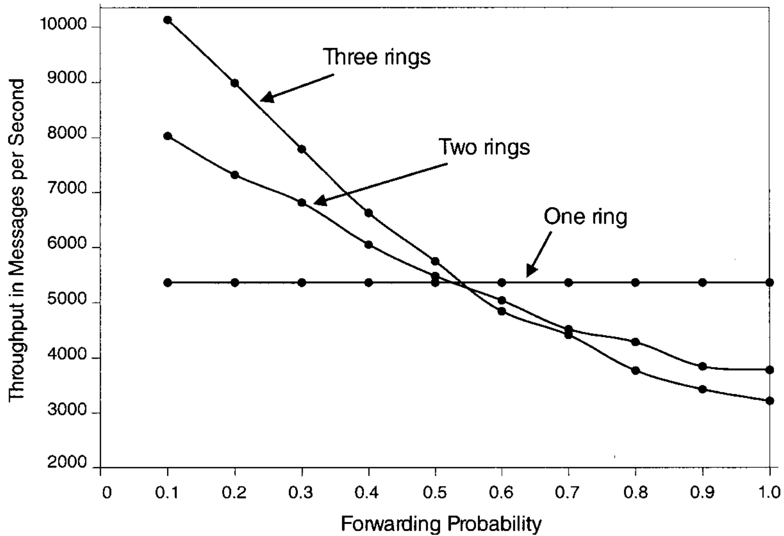


Fig. 16. Throughput for the three topologies as a function of forwarding probability.

100,000 messages. During a typical run, 100 to 1,000 messages out of the 100,000 messages were lost and had to be retransmitted.

All of the measurements were done for a deterministic message generation process. Each processor broadcasts messages at the maximum rate allowed by the flow control mechanisms, and has its full window size of messages to broadcast each time it receives the token. No measurements were made for a Poisson message generation process because the overheads and inaccuracies of the Unix timer delay mechanism precludes the use of such a process, even at 1,000 messages per second.

Figure 16 compares the throughput for the three topologies, as a function of the probability that any particular message is forwarded by a gateway. The effect of process group locality on the performance of the multiple-ring topologies is evident from these graphs. For high process group locality and thus low forwarding probability, the multiple-ring topologies achieve higher throughputs than the single-ring topology. With a forwarding probability of 0.1, the throughput for a single ring is 5,337 messages/sec, for the two-ring topology 7,993 messages/sec, and for the three-ring topology 10,086 messages/sec. If the forwarding probability is higher, or the processing required to forward a message through a gateway takes longer, the performance advantages of the multiple-ring topologies are reduced.

When considering latency rather than throughput, the advantage of the multiple-ring topologies is even more evident. Figure 17 compares the latencies to agreed and safe delivery for the three topologies, as a function of throughput, for a forwarding probability of 0.1. As the top graph of the figure shows, the latency to agreed delivery increases slowly as the throughput increases. The same slow increase can be extended to higher throughputs, beyond saturation of the single-ring topology by use of two-

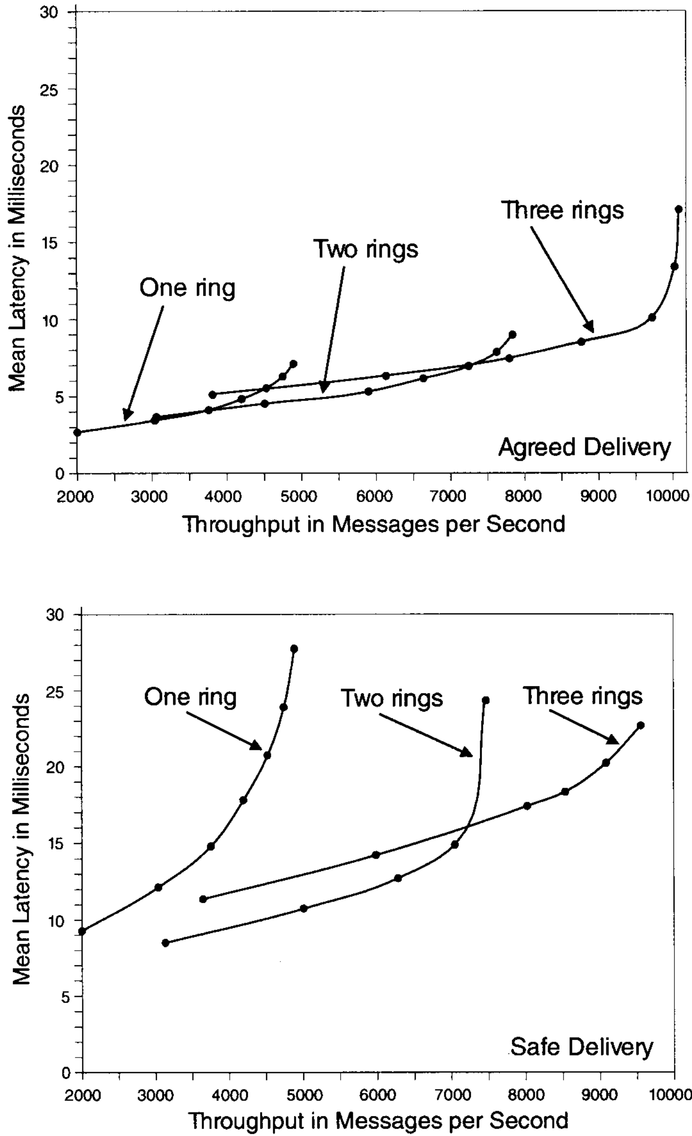


Fig. 17. The mean latencies to agreed and safe delivery obtained from measurement for one ring with nine processors, two rings with eight processors and a gateway, and three rings with nine processors and two gateways in a linear topology. The message generation process is deterministic, and the forwarding probability is 0.1.

ring and three-ring topologies. As the bottom graph of the figure shows, the latency to safe delivery exhibits similar characteristics.

With the small number of processors available in our laboratory, the advantage of multiple-ring topologies is most evident for low forwarding

probabilities. At higher forwarding probabilities, the benefits of the multiple-ring topologies become more evident for larger numbers of processors.

5.2 Performance Analysis

5.2.1 *Mean Latency Analysis.* We now consider a simple analysis of the reduction in the mean latency to agreed delivery that results from structuring the system into multiple rings to exploit process group locality and selective forwarding of messages. We introduce the following notation:

N Number of processors (not gateways)

M Number of rings

n Diameter of the topology in rings

m Number of processors (not gateways) on a ring

k Number of gateways on a ring

p Probability of forwarding a message through a gateway

t Mean time to forward a message through a gateway

r Mean number of rings on which a message is transmitted

l Mean latency to deliver a message on a particular ring

L Mean latency to deliver a message in the topology

For this analysis we assume that the events of forwarding messages through different gateways are independent of each other. Moreover, we assume that, for each source ring, the network structure embeds a complete binary tree rooted at that ring. The nodes represent all of the rings in the topology, and the edges represent gateways (though not necessarily all of them); thus, $M = 2^n - 1$ and $m = N/M = N/(2^n - 1)$. The messages generated by the root node are forwarded down the tree. This analysis is intended solely for the purpose of demonstrating scalability and does not consider all possible network structures.

The mean number r of rings on which a message is transmitted is given by

$$r = 1 + 2p + 4p^2 + \ \ldots\ + 2^{n-1} = \frac{1 - 2^n p^n}{1 - 2p}.$$

The mean latency l to deliver a message at a particular processor on a particular ring (single-ring latency) is proportional to the number of processors (and gateways) on that ring multiplied by the mean number of rings on which a message is transmitted plus the mean time to forward a message through the gateway and is given by

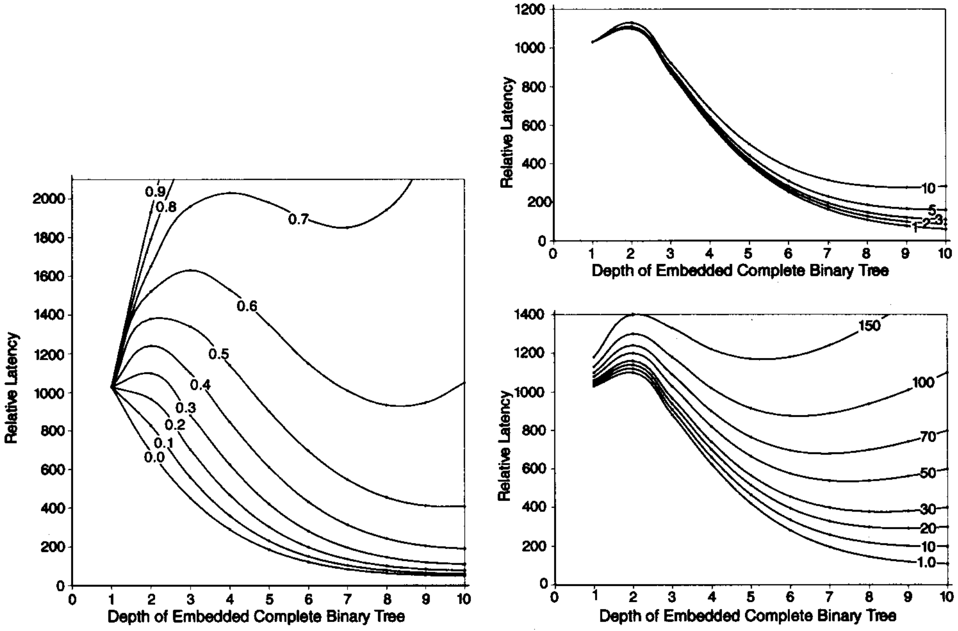


Fig. 18. At the left, the mean latency L as a function of n for various values of t with $N = 1024$, $k = 3$, and $p = 0.3$. At the top right, the mean latency L as a function of n for various values of p with $N = 1024$, $k = 3$, and $t = 1.0$. At the bottom right, the mean latency L as a function of n for various values of k with $N = 1024$, $p = 0.3$, and $t = 1.0$.

$$l = c((m + k)r + t) = c\left(\left(\frac{N}{2^n - 1} + k\right)\left(\frac{1 - 2^n p^n}{1 - 2p}\right) + t\right),$$

where c is a proportionality constant. Here the factor $m + k$ reflects the size of each ring and, thus, the traffic generated on the ring and the time to deliver messages on the ring. The factor r reflects the increased traffic on the ring from messages originated on other rings and forwarded to this ring.

The mean latency L to deliver a message at a particular processor in the topology (multiple-ring latency) is given by

$$L = nl = nc((m + k)r + t) = nc\left(\left(\frac{N}{2^n - 1} + k\right)\left(\frac{1 - 2^n p^n}{1 - 2p}\right) + t\right).$$

Here the factor n reflects the time to gather messages with the current timestamp, up the binary tree of depth n, so that the message can be delivered.

Figure 18 explores the values of L as n, p, k, and t are varied. At the left of the figure, L is shown as a function of n for various values of p with $N = 1024$, $k = 3$, and $t = 1.0$. For $p < 0.5$ (exploiting process group local-

ity and selective forwarding), the smallest values of the latency are achieved with the largest values of n. A value of p greater than 0.5 is disadvantageous because the gateways filter few messages, and the same messages are transmitted on many rings, increasing the load on those rings. At the top right of the figure, L is shown as a function of n for various values of k with $N = 1024$, $p = 0.3$, and $t = 1.0$. Changing the value of k does not have a major effect on the latency, but larger values of k become disadvantageous for large values of n because each ring contains few processors but many gateways. Larger values of k correspond to more tightly interconnected networks. A typical value of k is 3, although there is little effect on L for values of k between 1 and 5. At the bottom right of the figure, L is shown as a function of n for various values of t with $N = 1024$, $k = 3$, and $p = 0.3$. As the time t through a gateway increases, the latency increases but, for values of t less than 20, the smallest delay is achieved for the largest values of n. A typical value of t is 1.0.

There is little interaction between p, k, and t in the formula for L. For $p < 0.5$, $k < 5$, and $t < 20$, the smallest latency is achieved for the largest values of n, corresponding to rings containing few processors. Substituting $n = \log_2(M + 1) = \log_2((N/m) + 1)$ into the formula for L, we obtain

$$L = c \log_2\left(\frac{N}{m} + 1\right)\left((m + k)\left(\frac{1 - \left(\left(\frac{N}{m} + 1\right)p^{\log_2\left(\frac{N}{m}+1\right)}\right)}{1 - 2p}\right) + t\right).$$

From the investigation above, m should be as small as possible (i.e., n as large as possible) while keeping the value of p less than 0.5. Such a structure exploits process group locality and selective forwarding of messages. Thus, by keeping m constant, L scales logarithmically with N.

5.2.2 *Probability Density Function Analysis*. The mean latency can be misleading, since some messages may incur much longer latencies than others. Figure 19 shows the probability density functions for the latencies to agreed and safe delivery, calculated using a detailed analytic model [Thomopoulos et al. 1998]. Results are given for 40 processors in one-ring, two-ring, and four-ring topologies when the forwarding probability is 0.3. As these graphs indicate, with multiple-ring topologies, not only is the mean latency reduced, but also the variation in the latency is reduced, a desirable characteristic for soft real-time systems.

5.3 Dominant Factors Affecting Performance

Today's multicast group communication protocols, operating over LANs, are limited by the processor rather than by the communication medium. 100Mbit/sec Ethernet exceeds the capability of quite expensive worksta-

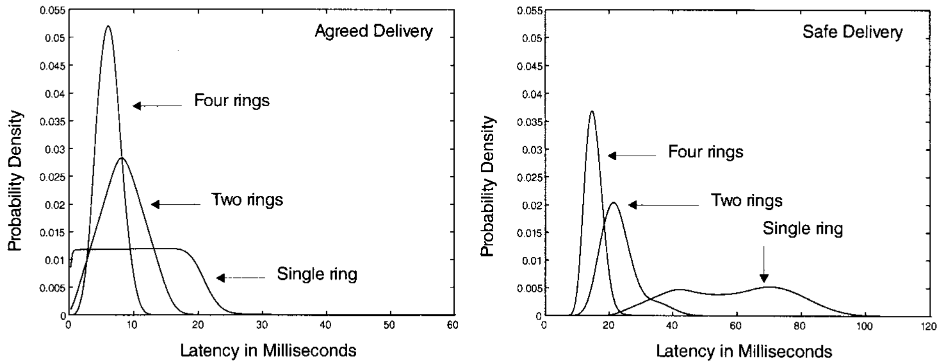


Fig. 19. The pdfs for the latencies to agreed and safe delivery for 40 processors on a single ring, on two-rings with a gateway, and on four rings in a ring-of-rings topology with four gateways. The message generation process is Poisson, the forwarding probability is 0.3, and the aggregate throughput is 2000 new messages/sec.

tions, even when those workstations are devoted 100% to communication. Because most users will wish to allocate the bulk of their processing resources to the application, rather than to the group communication protocol, it is unlikely for several years to come that processors will become fast enough to render inadequate the 1Gbit/sec Ethernet currently being developed.

In the past, much attention has been given to buffer management and message copying. Elaborate communication kernels have been designed to minimize the cost of these activities. However, recent operating systems have optimized commonly used buffer management system calls, such as *bcopy* and *malloc*. The time spent in these calls is now quite small and is not a significant factor in the performance of Totem. Rather, the performance of Totem is determined almost entirely by context switching. Context switches from user space into the kernel and back to user space are made whenever a message is received. Processing the token in the single-ring protocol requires four context switches, two to receive it and two to transmit it, instead of the two context switches for a regular message. The performance of the gateways, which are the bottleneck of the multiple-ring protocol, is restricted primarily by the increased number of context switches incurred by servicing two rings. Unfortunately, faster processors do not necessarily provide proportionately faster context switching.

6. CONCLUSION

The Totem multiple-ring protocol provides reliable totally ordered delivery of messages across multiple local-area networks interconnected by gateways. It achieves a consistent global total order of messages systemwide, in systems that are subject to network partitioning and remerging and to processor failure and recovery. It scales well within a local area and achieves a latency that increases logarithmically with system size. Mes-

sages destined for processes in a particular process group are forwarded by the gateways only to those parts of the network containing members of the process group. When the system is structured so that the process groups exhibit a high degree of locality, messages need not be forwarded across the entire network, and high throughput and low latency are achieved.

ACKNOWLDEGMENTS

The authors wish to thank the anonymous reviewers for their constructive comments, which have greatly improved this article. The authors also wish to thank E. Thomopoulos for the graphs of the probability density functions for the latencies given in Figure 19.

REFERENCES

AGARWAL, D. A. 1994. Totem: A reliable ordered delivery protocol for interconnected local-area networks. Ph.D thesis, Department of Electrical and Computer Engineering, University of California, Santa Barbara, CA.

AGARWAL, D. A., MOSER, L. E., MELLIAR-SMITH, P. M., AND BUDHIA, R. K. 1995. A reliable ordered delivery protocol for interconnected local-area networks. In *Proceedings of the International Conference on Network Protocols* (Tokyo, Japan, Nov.), 365–374.

ALVAREZ, G., CRISTIAN, F., AND MISHRA, S. 1998. On-demand asynchronous atomic broadcast. In *Proceedings of the 5th IFIP International Working Conference on Dependable Computing for Critical Applications* (Urbana-Champaign, IL, Sept. 1995), R. K. Iyer, M. Morganti, W. K. Fuchs, and V. Gligor, Eds. IEEE Computer Society Press, Los Alamitos, CA, 119–137.

AMIR, Y., DOLEV, D., KRAMER, S., AND MALKI, D. 1992. Transis: A communication subsystem for high availability. In *Proceedings of the 22nd IEEE International Symposium on Fault-Tolerant Computing* (Boston, MA, July). IEEE Press, Piscataway, NJ, 76–84.

AMIR, Y., MOSER, L. E., MELLIAR-SMITH, P. M., AGARWAL, D. A., AND CIARFELLA, P. 1995. The Totem single-ring ordering and membership protocol. *ACM Trans. Comput. Syst. 13*, 4 (Nov.), 311–342.

BIRMAN, K. P. AND VAN RENESSE, R. 1994. *Reliable Distributed Computing with the Isis Toolkit*. IEEE Computer Society Press, Los Alamitos, CA.

CHANG, J.-M. AND MAXEMCHUK, N. F. 1984. Reliable broadcast protocols. *ACM Trans. Comput. Syst. 2*, 3 (Aug.), 251–273.

CRISTIAN, F. AND MISHRA, S. 1995. The pinwheel asynchronous atomic broadcast protocols. In *Proceedings of the 2nd International Symposium on Autonomous Decentralized Systems* (Phoenix, AZ, Apr.). IEEE Computer Society Press, Los Alamitos, CA, 215–221.

DOLEV, D., KRAMER, S., AND MALKI, D. 1993. Early delivery totally ordered multicast in asynchronous environments. In *Proceedings of the 23rd IEEE International Symposium on Fault-Tolerant Computing* (Toulouse, France, June). IEEE Computer Society Press, Los Alamitos, CA, 544–553.

EZHILCHELVAN, P. D., MACEDO, R. A., AND SHRIVASTAVA, S. K. 1995. Newtop: A fault-tolerant group communication protocol. In *Proceedings of the 15th IEEE International Conference on Distributed Computing Systems* (Vancouver, Canada, May/June). IEEE Computer Society Press, Los Alamitos, CA, 296–306.

JIA, W., KAISER, J., AND NETT, E. 1996. Fault-tolerant group communication. *IEEE Micro 16*, 2 (Apr.), 59–67.

KAASHOEK, M. F. AND TANENBAUM, A. S. 1991. Group communication in the Amoeba distributed operating system. In *Proceedings of the 11th IEEE International Conference on Distributed Computing Systems* (Arlington, TX, May). IEEE Computer Society Press, Los Alamitos, CA, 436–447.

LAMPORT, L. 1978. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM 21*, 7, 558–565.

MELLIAR-SMITH, P. M. AND MOSER, L. E. 1993. Trans: A reliable broadcast protocol. *IEE Trans. Commun. 140*, 6 (Dec.), 481–493.

MELLIAR-SMITH, P. M., MOSER, L. E., AND AGRAWALA, V. 1990. Broadcast protocols for distributed systems. *IEEE Trans. Parallel Distrib. Syst. 1*, 1 (Jan.), 17–25.

MISHRA, S., PETERSON, L. L., AND SCHLICHTING, R. D. 1993. Consul: A communication substrate for fault-tolerant distributed programs. *Distrib. Syst. Eng. 1*, 2 (Dec.), 87–103.

MOSER, L. E., AMIR, Y., MELLIAR-SMITH, P. M., AND AGARWAL, D. A. 1994. Extended virtual synchrony. In *Proceedings of the 14th IEEE International Conference on Distributed Computing Systems* (Poznan, Poland, June). IEEE Computer Society Press, Los Alamitos, CA, 56–65.

MOSER, L. E., MELLIAR-SMITH, P. M., AGARWAL, D. A., BUDHIA, R. K., AND LINGLEY-PAPADOPOULOS, C. A. 1996. Totem: A fault-tolerant multicast group communication system. *Commun. ACM 39*, 4 (Apr.), 54–63.

MOSER, L. E., MELLIAR-SMITH, P. M., AND AGRAWALA, V. 1993. Asynchronous fault-tolerant total ordering algorithms. *SIAM J. Comput. 22*, 4 (Aug.), 727–750.

RAJAGOPALAN, B. AND MCKINLEY, P. K. 1989. A token-based protocol for reliable, ordered multicast communication. In *Proceedings of the 8th IEEE Symposium on Reliable Distributed Systems* (Seattle, WA, Oct.). IEEE Computer Society Press, Los Alamitos, CA, 84–93.

RODRIGUES, L. E. T., FONSECA, H., AND VERISSIMO, P. 1996. Totally ordered multicast in large-scale systems. In *Proceedings of the 16th IEEE International Conference on Distributed Computing Systems* (Hong Kong, May). IEEE Computer Society Press, Los Alamitos, CA, 503–510.

THOMOPOULOS, E., MOSER, L. E., AND MELLIAR-SMITH, P. M. 1998. Analyzing the latency of the Totem multicast protocols. In *Proceedings of the 6th International Conference on Computer Communications and Networks* (Las Vegas, NV, Sept.), 42–50.

VAN RENESSE, R., BIRMAN, K. P., AND MAFFEIS, S. 1996. Horus: A flexible group communications system. *Commun. ACM 39*, 4 (Apr.), 76–83.

WHETTEN, B., MONTGOMERY, T., AND KAPLAN, S. 1995. A high performance totally ordered multicast protocol. In *Theory and Practice in Distributed Systems* (Dagstuhl Castle, Germany, Sept. 1994), K. Birman, F. Mattern, and A. Schiper, Eds. Springer-Verlag, Berlin, Germany, 33–57.

Received: July 1997; revised: January 1998; accepted: February 1998