The Totem Single-Ring Ordering and Membership Protocol

Y. AMIR, L. E. MOSER, P. M. MELLIAR-SMITH, D. A. AGARWAL, P. CIARFELLA
University of California, Santa Barbara

Fault-tolerant distributed systems are becoming more important but, in existing systems, maintaining the consistency of replicated data is quite expensive. The Totem single-ring protocol supports consistent concurrent operations by placing a total order on broadcast messages. This total order is derived from the sequence number in a token that circulates around a logical ring imposed on a set of processors in a broadcast domain. The protocol handles reconfiguration of the system when processors fail and restart or the network partitions and remerges. Extended virtual synchrony ensures that processors deliver messages and configuration changes to the application in a consistent total order system-wide. An effective flow control mechanism enables the Totem single-ring protocol to achieve message ordering rates significantly higher than the best prior total ordering protocols.

Categories and Subject Descriptors: C.2.1 [**Computer-Communications 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: Protocols, Performance, Reliability

Additional Key Words and Phrases: Flow control, membership, reliable delivery, token passing, total ordering, virtual synchrony

---

Earlier versions of the Totem single-ring protocol appeared in the Proceedings of the IEE International Conference on Information Engineering, Singapore (December 1991) and in the Proceedings of the IEEE 13th International Conference on Distributed Computing Systems, Pittsburgh, PA (May 1993).
This research was supported by NSF Grant No. NCR-9016361, ARPA Contract No. N00174-93-K-0097, and Rockwell CMC/State of California MICRO Grant No. 92-101.
Authors' Addresses: Y. Amir, Computer Science Department, The Hebrew University of Jerusalem, Israel; L. E. Moser and P. M. Melliar-Smith, Department of Electrical and Computer Engineering, University of California, Santa Barbara, CA 93106; D. A. Agarwal, Lawrence Berkeley National Laboratory, Berkeley, CA 94720; P. Ciarfella, Cascade Communications Corporation, 5 Carlisle Road, Westford, MA 01886.

1. INTRODUCTION

Fault-tolerant distributed systems are becoming more important due to the increasing demand for more reliable operation and improved performance. Maintaining the consistency of replicated data and coordinating the activities of cooperating processors present substantial problems, which are made more difficult by concurrency, asynchrony, fault-tolerance, and real-time performance requirements. Existing fault-tolerant distributed systems that address these problems are difficult to program, and expensive in the number of messages broadcast and/or computations required. Recent protocols for fault-tolerant distributed systems [2; 5; 9; 12; 17] employ the idea of placing a partial or total order on broadcast messages to simplify the application programs and to reduce the communication and computation costs.

The Totem single-ring protocol supports high-performance fault-tolerant distributed systems that must continue to operate despite network partitioning and remerging, and processor failure and restart. Totem provides totally ordered message delivery with low overhead, high throughput, and low latency using a logical token-passing ring imposed on a broadcast domain. The key to its high performance is an effective flow control mechanism. Totem also provides rapid detection of network partitioning and processor failure together with reconfiguration and membership services. Its novel mechanisms prevent delivery of messages in different orders in different components of a partitioned network, and provide accurate information about which processors have delivered which messages. Earlier versions of the Totem single-ring protocol are described in [3; 11].

Programming the application is considerably simplified if messages are delivered in total order rather than only in causal order, or if messages are delivered in causal order rather than only in FIFO order. In prior systems, delivery of messages in total order has been more expensive than delivery of messages in causal order, and delivery of messages in causal order has been more expensive than FIFO delivery. The Totem single-ring protocol can, however, deliver totally ordered messages with high throughput at no greater cost than causally ordered messages or, indeed, than reliable point-to-point FIFO messages. A total order on messages simplifies the application programming by reducing the risk of inconsistency when replicated data are updated, and by resolving the contention for shared resources within the system, such as the claiming of locks.

In Totem, messages are delivered in agreed order, which guarantees that processors deliver messages in a consistent total order and that, when a processor delivers a message, it has already delivered all prior messages originated within its current configuration. Totem also provides delivery in safe order, which guarantees additionally that, when a processor delivers a message, it has determined that every processor in the current configuration has received and will deliver the message unless that processor fails. Delivery of a message in agreed or safe order is requested by the originator of the message.

Delivery in a consistent total order is not easy to achieve in distributed systems that are subject to processor failure and network partitioning. A failing processor, or a group of processors that have become isolated, may deliver messages in an order that is different from the order determined by other processors. As long as those processors remain failed or isolated, these inconsistencies are not apparent,

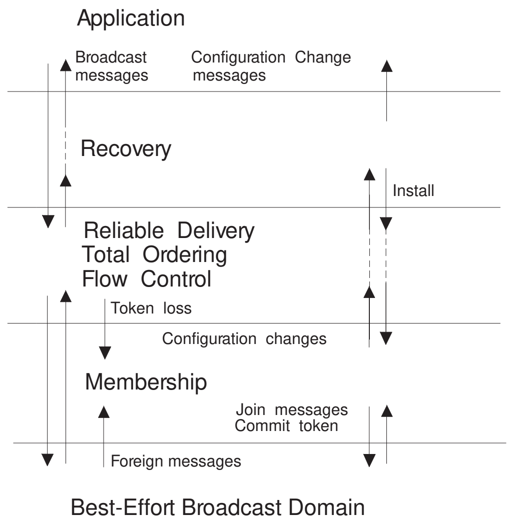

Fig. 1: The Totem single-ring protocol hierarchy.

but as soon as a processor is repaired and readmitted to the system, or as soon as the components of a partitioned system are remerged, the inconsistencies in the message order may become manifest and recovery may be difficult. The Totem protocol cannot guarantee that every processor is able to deliver every message but it does guarantee that, if two processors deliver a message, they deliver the message in the same total order.

The application programs may also need to know about configuration changes. Different processors may learn of a configuration change at different times, but they must form consistent views of the configuration change and of the messages that precede or follow the configuration change. Birman [5] devised the concept of virtual synchrony, which ensures that processors deliver messages consistently in the event of processor fail-stop faults. We have generalized this concept to extended virtual synchrony [14], which applies to systems in which the network can partition and remerge, and in which processors can fail and restart with stable storage intact.

The Totem single-ring protocol is designed to operate over a single broadcast domain, such as an Ethernet. It uses the Unix UDP service, which provides a best-effort multicast service over such media. Other media that provide a best-effort multicast service, such as ATM or the Internet MBone, can be used to construct the broadcast domain needed by Totem.

The software architecture of the Totem single-ring protocol is shown in Figure 1. The arrows on the left represent the passage of messages through the protocol hierarchy, while the arrows on the right represent Configuration Change messages and configuration installation. Using a logical token-passing ring imposed on the physical broadcast domain, the single-ring protocol provides reliable totally ordered messsage delivery and effective flow control. On detection of token loss, or on receiving a message from a processor not on its ring, a processor invokes the membership protocol to form a new ring using Join messages and a Commit token transmitted over the broadcast domain. The membership protocol activates the recovery protocol with the proposed configuration change. The recovery protocol uses the single-ring ordering protocol to recover missing messages. The processor then installs the new ring, by delivering Configuration Change messages to the application.

2. RELATED WORK

Our work on the Totem protocol is based on our combined experience with two systems: the Trans and Total reliable ordered broadcast and membership protocols [12; 16] and the Transis group communication system [1; 2].

The Trans protocol uses positive and negative acknowledgments piggybacked onto broadcast messages and exploits the transitivity of positive acknowledgments to reduce the number of acknowledgments required. The Total protocol, layered on top of the Trans protocol, is a fault-tolerant total ordering protocol that continues to order messages provided that a resiliency constraint is met. The membership protocol, layered on top of the Total protocol, ensures that each change in the membership occurs at the same logical time in each processor, corresponding to a position in the total order. The Totem protocol was developed to address the computational overhead of Trans and Total, and is intended for local-area networks with fast and highly reliable communication.

The Transis group communication system provides reliable ordered group multicast and membership services. Transis initially based its ordering protocol on the Trans protocol but, more recently, has also included the Totem protocol for message ordering. Unlike other prior protocols, the Transis membership protocol supports remerging of a partitioned network, and maintains a consensus view of the membership of each component, rather than a global consensus view of the entire system. The Totem membership protocol uses the idea, first proposed for Transis, that the membership can be reduced in size to ensure termination.

In [7] Chang and Maxemchuk described a reliable broadcast and ordering protocol that uses a token-based sequencer strategy. Unlike Totem, which requires that a processor must hold the token to broadcast a message, their protocol allows processors to broadcast messages at any time. The processor holding the token is responsible for broadcasting an acknowledgment message that includes a sequence number for each message acknowledged. A processor that has not received a message sends a negative acknowledgment to request retransmission by the processor that acknowledged the message. While the latency is good at low loads, it increases at high loads and in the presence of a failed processor.

More closely related to Totem is the TPM protocol of Rajagopalan and McKinley [18], which also uses a token to control broadcasting and sequencing of messages. The TPM protocol provides the safe delivery but not the agreed delivery that Totem provides. In the absence of processor failure and network partitioning, TPM requires on average two and one-half token rotations for safe delivery, whereas Totem requires two token rotations. In the event of network partitioning, only the component containing a majority of the processors continues to operate; processors in the other components block. In contrast, Totem handles network partitioning and remerging by allowing each component of a partitioned system to continue operating, not just the component that contains a majority of the processors.

Birman's Isis system [5], and the more recent Horus system [19], have focused on process groups and the application program interface. Isis provides BCAST or unordered messages, CBCAST or causally ordered messages, and ABCAST or totally ordered messages. A vector clock strategy is used to ensure causal ordering, and a token-based sequencer, similar to that of Chang and Maxemchuk, is used to provide total ordering. Isis introduced the important idea of virtual synchrony in which reconfiguration messages are ordered relative to other messages so that a consistent view of the system is maintained as the system changes dynamically. The user interfaces of both Transis and Totem were inspired by the Isis user interface.

The Psync protocol of Peterson, Buchholz and Schlichting [17] constructs a partial order on messages that can be converted into a total order. Isis, Trans, and Transis employ a similar strategy. In contrast, Totem constructs a total order on messages directly without constructing a partial order first. Mishra, Peterson and Schlichting [13] have developed a membership protocol based on the partial order of Psync.

In [9] Kaashoek and Tanenbaum describe group communication in the Amoeba distributed operating system. One processor, called the sequencer, is responsible for placing a total order on messages. Processors send point-to-point messages to the sequencer, which assigns sequence numbers to messages and broadcasts them to the other processors. Messages are recovered by sending a request to the sequencer for retransmission. Group membership functions are also provided. Performance is excellent for very short messages, but deteriorates for long messages and also if high resilience to processor failure and network partitioning is required.

The excellent performance of the Totem single-ring protocol is achieved using a flow control strategy that limits message loss due to buffer overflow at the receivers. Related to this flow control strategy are the sliding-window strategy, the FDDI token rotation time limit, and the flow control mechanism of Transis. The use of a token in combination with a window for flow control on a broadcast medium works much better than either mechanism in isolation and, as far as we know, has not been investigated in prior work.

3. THE MODEL

We consider a distributed system built on a broadcast domain consisting of a finite number of processors that communicate by broadcasting messages. We use the term *originate* to refer to the first broadcast of a message generated by the application. Each broadcast of a message is received immediately or not at all by a processor in the broadcast domain and, thus, messages may have to be retransmited to achieve reliable delivery. A processor receives all of its own broadcast messages.

The broadcast domain may become partitioned so that processors in one component of the partitioned system are unable to communicate with processors in another component. Communication between separated components can subsequently be reestablished.

Each processor within the system has a unique identifier and stable storage. Processors can incur fail stop, timing, or omission faults. A processor that is excessively slow, or that fails to receive a message an excessive number of times, can be regarded as having failed. Failed processors can be repaired, and are reconfigured into the system when they restart. If a processor fails and restarts, its identifier does not change and all or part of its state may have been retained in stable storage.

There are no malicious faults, such as faults in which processors generate incorrect messages or in which the communication medium undetectably modifies messages in transit.

Imposed on the broadcast domain is a logical token-passing ring. The token is a special message transmitted point-to-point. The token may be lost by not being received by a processor on the ring. Each ring has a *representative*, chosen deterministically from the membership when the ring is formed, and an identifier that consists of a ring sequence number and the identifier of the representative. To ensure that ring sequence numbers and hence ring identifiers are unique, each processor records its ring sequence number in stable storage.

We use the term *ring* to refer to the infrastructure of Totem, and the term *configuration* to represent the view provided to the application. The *membership* of a configuration is a set of processor identifiers. The minimum configuration for a processor consists of the processor itself. A *regular* configuration has the same membership and identifier as its corresponding ring. A *transitional* configuration consists of processors that are members of a new ring coming directly from the same old ring; it has an identifier that consists of a "ring" sequence number and the identifier of the representative.

We distinguish between the terms "receive" and "deliver," as follows. A processor *receives* messages that were broadcast by processors in the broadcast domain, and a processor *delivers* messages in total order to the application.

Two types of messages are delivered to the application. *Regular* messages are generated by the application for delivery to the application. *Configuration Change* messages are generated by the processors for delivery to the application, without being broadcast, to terminate one configuration and to initiate another. The identifiers of the regular and Configuration Change messages consist of configuration identifiers and message sequence numbers.

We define a causal order that is a modification of Lamport's causal order [10] in that it applies to messages rather than events and is constrained to messages originated within a single configuration. This allows remerging of a partitioned network and joining of a failed processor without requiring all messages in the history to be delivered. Causal and total orders[1] are defined on sets of messages, as follows:

Causal Order for Configuration C. The reflexive transitive closure of the "precedes" relation defined for all processors p that are members of C, as follows:

—Message m_1 precedes message m_2 if processor p originated m_1 in configuration C before p originated m_2 in C.

—Message m_1 precedes message m_2 if processor p originated m_2 in configuration C and p delivered m_1 in C before originating m_2.

[1]A total order on a set S is a relation $\leq$ that satisfies the reflexive ($x \leq x$), transitive (if $x \leq y$ and $y \leq z$, then $x \leq z$), anti-symmetric (if $x \leq y$ and $x \neq y$, then $y \not\leq x$), and comparable properties ($x \leq y$ or $y \leq x$). A partial order on a set S is a relation $\leq$ defined on S that satisfies the reflexive, transitive, and anti-symmetric properties. To adhere to standard mathematical practice in which partial and total orders are reflexive, "before" must be regarded as non-strict, *i.e.* an event occurs before itself.

The causal order is assumed to be anti-symmetric and, thus, is a partial order.[2]

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

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

Note that some processors in configuration C may not deliver all messages of the Delivery Order for Configuration C.

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

—For each processor p and each configuration C of which p is a member, the Configuration Change message delivered by p that initiates C precedes every message m delivered by p in C.

—For each processor p and each configuration C of which p is a member, every message m delivered by p in C precedes the Configuration Change message delivered by p that terminates C.

In [4] we prove that the Delivery Order for Configuration C is a total order and that the Global Delivery Order is a total order.

4. SERVICES

The objective of the Totem single-ring protocol is to provide the application with reliable totally ordered message delivery and membership services, as defined below.

4.1 Membership Services

The Totem membership protocol provides the following properties:

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

Consensus. All of the processors that install a configuration determine that the members of the configuration have reached consensus on the membership.[3]

Termination. If a configuration ceases to exist for any reason, such as processor failure or network partitioning, then every processor of that configuration either installs a new configuration by delivering a Configuration Change message or fails before doing so. The Configuration Change message contains the identifier of the configuration it terminates, the identifier of the configuration it initiates, and the membership of the configuration it initiates.

Configuration Change Consistency. If processors p and q install configuration C_2 directly after C_1, then p and q both deliver the same Configuration Change message to terminate C_1 and initiate C_2.

[2]This property cannot be proved. That the physical world is anti-symmetric must be an assumption.

[3]This does not violate the impossibility result of Fischer, Lynch and Paterson [8] because the membership protocol allows the membership to decrease in order to reach consensus.

4.2 Reliable Ordered Delivery Services

The Totem total ordering protocol provides the following properties, which hold for all configurations C and for all processors p in C:

Reliable Delivery for Configuration C

—Each message m has a unique identifier.

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

—If processor p originates message m, then p will deliver m or will fail before delivering a Configuration Change message to install a new regular configuration.

—If processor p is a member of regular configuration C and no configuration change ever occurs, then p will deliver in C all messages originated in C.

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

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

Reliable delivery defines the basic requirements on message delivery, in particular, which messages a processor must deliver within a configuration.

Delivery in Causal Order for Configuration C

—Reliable delivery for configuration C.

—If processor p delivers messages m_1 and m_2, and m_1 precedes m_2 in the causal order for configuration C, then p delivers m_1 before p delivers m_2.

Causal delivery imposes an ordering constraint to ensure that the delivery order respects Lamport causality within a configuration.

Delivery in Agreed Order for Configuration C

—Delivery in causal order for configuration C.

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

Agreed delivery requires that all processors deliver messages within a configuration in the same total order. Moreover, when a processor delivers a message, it must have delivered all preceding messages in the total order for the configuration.

Delivery in Safe Order for Configuration C

—Delivery in agreed order for configuration C.

—If processor p delivers message m in regular configuration C in safe order, then every member of C has installed C.

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

Delivery of a message in safe order requires that a processor has determined that all of the processors in the configuration have received the message. This determination is typically based on acknowledgments from the processors indicating that they have received the message and all of its predecessors in the total order. Once a processor has acknowledged receipt of a safe message, it is required to deliver the message unless it fails. Note that this requirement does not guarantee that a processor delivers the message in the same configuration as all of the other processors. Totem uses a Configuration Change message to notify the application of the membership of the configuration within which delivery is guaranteed as safe.

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.

Virtual synchrony was devised by Birman [5] to ensure that view (configuration) changes occur at the same point in the message delivery history for all operational processors. Processors that are members of two successive views must deliver exactly the same set of messages in the first view. A failed processor can only be readmitted to the system as a new processor. Thus, failed processors are not constrained as to the messages they deliver or their order, and messages delivered by a failed processor have no effect on the system. If the system partitions, only processors in one component, the primary component, continue to operate; all of the other processors are deemed to have failed.

Extended virtual synchrony extends the concept of virtual synchrony to systems in which all components of a partitioned system continue to operate and can subsequently remerge, and to systems in which failed processors can be repaired and can rejoin the system with stable storage intact. Two processors may deliver different sets of messages, when one of them has failed or when they are members of different components, but they must not deliver messages inconsistently. In particular, if processor p delivers message m_1 before p delivers message m_2, then processor q must not deliver message m_2 before q delivers message m_1.[4]

Extended virtual synchrony requires that the properties of delivery in agreed and safe order must be satisfied. If processor p delivers message m as safe in configuration C, then every processor in C has received m and will deliver m before it installs a new regular configuration, unless that processor fails. This is achieved by installing a transitional configuration with a reduced membership, within which any remaining messages from the prior configuration are delivered, while honoring the agreed and safe delivery guarantees. Thus, Totem delivers two Configuration Change messages, the first to introduce a smaller transitional configuration and the second to introduce the new regular configuration. In [14] we demonstrated that virtual synchrony can be implemented on top of the more general extended virtual synchrony property provided by Totem.

Proofs of correctness for the Totem single-ring protocol based on the service properties defined above can be found in [4].

[4]Note, however, that a definition based on pairwise delivery of messages does not suffice. We must ensure that the Global Delivery Order has no cycles of any length.

5. THE TOTAL ORDERING PROTOCOL

The Totem single-ring ordering protocol provides agreed and safe delivery of messages within a broadcast domain. Imposed on the broadcast domain is a logical token-passing ring. The token controls access to the ring; only the processor in possession of the token can broadcast a message. A processor can broadcast more than one message for each visit of the token, subject to the constraints imposed by the flow control mechanisms described in Section 8. When no processor has a message to broadcast, the token continues to circulate. Each processor has a set of input buffers in which it stores incoming messages. The flow control mechanisms avoid overflow of these input buffers.

Each message header contains a sequence number derived from a field of the token; thus, there is a single sequence of message sequence numbers for all processors on the ring. Delivery of messages in sequence number order is agreed delivery. Safe delivery uses an additional field of the token, the *aru* field, to determine when all processors on the ring have received a message.

We now describe the Totem single-ring ordering protocol with the assumptions that the token is never lost, that processor failures do not occur, and that the network does not become partitioned; however, messages may be lost. In Section 6 we relax these assumptions and extend the protocol to handle token loss, processor failure and restart, and network partitioning and remerging.

5.1 The Data Structures

Regular Message

Each regular message contains the following fields:

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

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

Regular Token

To broadcast a message on the ring, a processor must hold the regular token, referred to also as the token. The token contains the following fields:

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

The *seq* field of the token provides a single total order of messages for all processors on the ring. The *aru* field is the basic acknowledgment mechanism that determines if a message can be delivered as safe.

Local Variables

Each processor maintains several local variables, including

- *my_token_seq*: The value of the *token_seq* when the processor forwarded the token last.
- *my_aru*: The sequence number of a message such that the processor has received all messages with sequence numbers less than or equal to this sequence number.
- *my_aru_count*: The number of times that the processor has received the token with an unchanged *aru* and with the *aru* not equal to *seq*.
- *new_message_queue*: The queue of messages originated by the application waiting to be broadcast.
- *received_message_queue*: The queue of messages received from the communication medium waiting to be delivered to the application.

A processor updates *my_token_seq* and *my_aru_count* as it receives tokens, and updates *my_aru* as it receives messages. When it transmits a message, the processor transfers the message from *new_message_queue* to *received_message_queue*. When it determines that a message has become safe, the processor no longer needs to retain the message for future retransmission and, thus, can discard the message from *received_message_queue*.

5.2 The Protocol

On receipt of the token, a processor completely empties its input buffer, either delivering the messages or retaining them until they can be delivered in order. It then broadcasts requested retransmissions and new messages, updates the token, and transmits the token to the next processor on the ring. For each new message that it broadcasts, the processor increments the *seq* field of the token and sets the sequence number of the message to this value.

Each time a processor receives the token, it compares the *aru* field of the token with *my_aru*. If *my_aru* is smaller, the processor replaces the *aru* with *my_aru* and sets the *aru_id* field of the token to its identifier. If the *aru_id* equals the processor's identifier, it sets the *aru* to *my_aru*. (In this case, the processor had set the *aru* on the last visit of the token and no other processor changed the *aru* during the token rotation.) Whenever the *seq* and the *aru* are equal, the processor increments *aru* and *my_aru* in step with *seq*, and sets the *aru_id* to a null value (a value that is not the id of any processor).

If the *seq* field of the token is greater than its *my_aru*, the processor has not received all of the messages that have been broadcast on the ring, so it augments the *rtr* field of the token with the missed messages. If the processor has received messages that appear in the *rtr* field, it retransmits those messages before broadcasting new messages. When it retransmits a message, the processor removes the sequence number of the message from the *rtr* field.

If a processor has received a message m and has delivered every message with sequence number less than that of m and if the originator of m requested agreed delivery, then the processor delivers m in agreed order. If, in addition, the processor forwards the token with the *aru* field greater than or equal to the sequence number of m on two successive rotations and if the originator of m requested safe delivery, then m is safe and the processor delivers m in safe order.

The total ordering protocol is, of course, unable to continue when the token is lost; a token retransmission mechanism has been implemented to reduce the probability of token loss. Each time a processor forwards the token, it sets a Token Retransmission timeout. If a processor receives a regular message or the token, it cancels the Token Retransmission timeout. On a Token Retransmission timeout, the processor retransmits the token to the next processor on the ring and then resets the timeout.

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

6. THE MEMBERSHIP PROTOCOL

The Totem single-ring ordering protocol is optimized for high performance under failure-free conditions, but depends on a membership protocol to resolve processor failure, network partitioning, and loss of all copies of the token. The membership protocol detects such failures and reconstructs a new ring on which the total ordering protocol can resume operation.

The objective of the membership protocol is to ensure consensus, in that every member of the configuration agrees on the membership of the configuration, and termination, in that every processor installs some configuration with an agreed membership within a bounded time unless it fails within that time. The membership protocol also generates a new token and recovers messages that had not been received by some of the processors when the failure occurred.

6.1 The Data Structures

Join Message

A Join message contains a set of identifiers of processors being considered for membership in the new ring by the processor broadcasting the Join message and also a set of identifiers of processors that it regards as having failed. These are contained in the *proc_set* and *fail_set* fields of the Join message defined below:

- *type*: Join.
- *sender_id*: The processor identifier of the sender.
- *proc_set*: The set of identifiers of processors that the sender is considering for membership in a new ring.
- *fail_set*: The set of identifiers of processors that the sender has determined to have failed.
- *ring_seq*: The largest ring sequence number of a *ring_id* known to the sender.

Join messages differ from regular messages in that a processor may broadcast a Join message without holding the token; moreover, Join messages are not retransmitted or delivered to the application.

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

Configuration Change Message

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

- *ring_id*: The identifier of the regular configuration if this message initiates a regular configuration, or the identifier of the preceding regular configuration if this message initiates a transitional configuration.
- *seq*: 0 if this message initiates a regular configuration, or the largest sequence number of a message delivered in the preceding regular configuration if this message initiates a transitional configuration.
- *conf_id*: The identifier of the old transitional configuration from which the processor is transitioning if this message initiates a regular configuration, or the identifier of the transitional configuration to which the processor is transitioning if this message initiates a transitional configuration.
- *memb*: The membership of the configuration that this message initiates.

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

A Configuration Change message may describe a change from an old configuration to a transitional configuration or from a transitional configuration to a new configuration. Configuration Change messages differ from regular messages in that they are generated locally at each processor and are delivered directly to the application without being broadcast.

Commit Token

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

- *memb_list*: A list containing a processor identifier, old ring *ring_id*, old ring *my_aru*, *high_delivered*, and *received_flg* fields for each member of the new ring.
- *memb_index*: The index of the processor in *memb_list* that last forwarded the Commit token.

For each processor identifier in *memb_list*, the *high_delivered* field is the largest sequence number of a message that the processor has delivered on the old ring. The *received_flg* field indicates that the processor has already received all of the messages possessed by other processors in its transitional configuration.

On the first rotation of the Commit token around the new ring, each processor sets its old *ring_id*, old ring *my_aru*, *high_delivered*, and *received_flg* fields. It also updates *memb_index*. The remaining fields are set by the representative when it creates the Commit token.

Local Variables

Each processor maintains several local variables, including

- *my_ring_id*: The ring identifier in the most recently accepted Commit token.
- *my_memb*: The set of identifiers of processors on the processor's current ring.
- *my_new_memb*: The set of identifiers of processors on the processor's new ring.
- *my_proc_set*: The set of identifiers of processors that the processor is considering for membership of a new ring.
- *my_fail_set*: The set of identifiers of processors that the processor has determined to have failed.
- *consensus*: A boolean array indexed by processors and indicating whether each processor is committed to the processor's *my_proc_set* and *my_fail_set*.

Stable storage is required to store a processor's ring sequence number, *my_ring_id.seq*. This stable storage is read only when a processor recovers from a failure, and is written when a configuration change occurs.

6.2 The Protocol

The membership protocol can be described by a finite state machine with seven events and four states, as illustrated in Figure 2.

6.2.1 *The Seven Events of the Membership Protocol*

Receiving a Foreign Message. Such a message was broadcast by a processor that is not a member of the receiving processor's ring, and activates the membership protocol in the receiving processor.

Receiving a Join Message. This informs the receiver of the sender's proposed membership and may cause the receiver to enlarge its *my_proc_set* or *my_fail_set*.

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

Token Loss Timeout. This timeout indicates that a processor did not receive the token or a regular message within the required amount of time and activates the membership protocol.

Token Retransmission Timeout. This timeout indicates that a processor should retransmit the token because it has not received the token or a regular message broadcast by another processor on the ring.

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

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

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

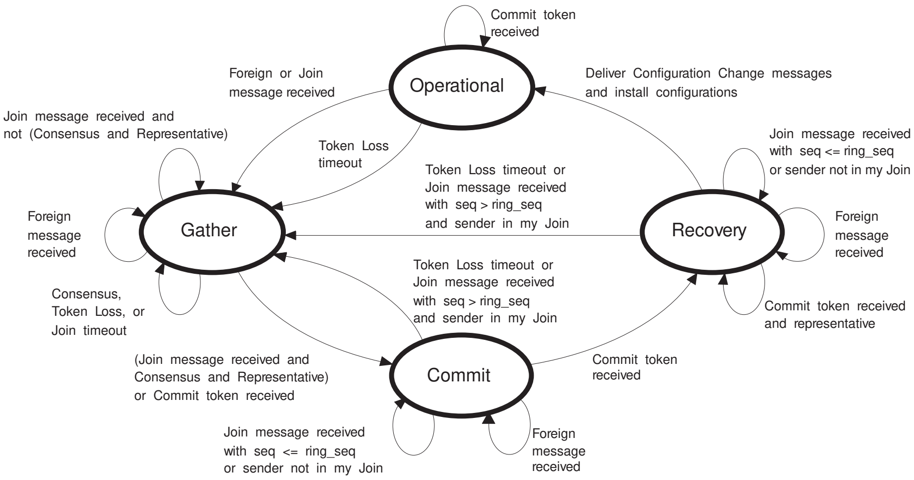

Fig. 2: The finite state machine for the membership protocol.

6.2.2 *The Four States of the Membership Protocol*

Operational State. In the Operational state (Figure 3), messages are broadcast and delivered in agreed or safe order, as requested by the originator of the message. Since processor failure and network partitioning result in loss of the token, the mechanism for detecting failures is the Token Loss timeout. When the Token Loss timeout expires or when a processor receives a Join message or a foreign message, the processor invokes the protocol for the formation of a new ring and shifts to the Gather state (Figure 7).

A processor buffers a message for retransmission until the message has been acknowledged by the other processors on the ring. If a processor repeatedly fails to receive a particular message, then the other processors buffer that message and all subsequent messages until that message is received. Consequently, a processor cannot be allowed to fail to receive messages indefinitely. When its local variable *my_aru_count* reaches a predetermined constant, a processor determines that some other processor has failed to receive, namely the processor whose identifier is in the *aru_id* field of the token. It then includes that processor's identifier in its *my_fail_set*, shifts to the Gather state, and broadcasts a Join message.

Gather State. In the Gather state (Figure 4), a processor collects information about operational processors and failed processors, and broadcasts that information in Join messages. When a processor receives a Join message, the processor updates its *my_proc_set* and *my_fail_set*. If its *my_proc_set* and *my_fail_set* have changed, the processor abandons its previous membership, broadcasts a Join message containing the updated sets, and resets the Join and Consensus timeouts. The Join timeout is shorter than the Consensus timeout and is used to increase the probability that Join messages from all currently operational processors are received during a single round of consensus.

A processor reaches consensus when it has received Join messages with *proc_set* and *fail_set* equal to its *my_proc_set* and *my_fail_set*, respectively, from every processor in the difference of those sets, *i.e.* *my_proc_set* $-$ *my_fail_set*. It then no

```
Regular token received:
  if token.ring_id ≠ my_ring_id or token.token_seq <= my_token_seq then
    discard token
  else
    determine how many messages I'm allowed to broadcast by flow control
    update retransmission requests
    broadcast requested retransmissions
    subtract retransmissions from allowed to broadcast
    for as many messages as allowed to broadcast do
      get message from new_message_queue
      increment token.seq
      set message header fields and broadcast message
    update my_aru
    if my_aru < token.aru or my_id = token.aru_id or token.aru_id = invalid then
      token.aru := my_aru
      if token.aru = token.seq then
        token.aru_id := invalid
      else token.aru_id := my_id
    if token.aru = aru in token on last rotation and token.aru_id ≠ invalid then
      increment my_aru_count
    else my_aru_count := 0
    if my_aru_count > fail_to_rcv_const and token.aru_id = my_id then
      add token.aru_id to my_fail_set
      call Shift_to_Gather
    else
      update token.rtr and token flow control fields
      increment token.token_seq
      forward token
      reset Token Loss and Token Retransmission timeouts
      deliver messages that satisfy their delivery criteria

Regular message received:
  cancel Token Retransmission timeout if set
  add message to receive_message_queue
  update retransmission request list
  update my_aru
  deliver messages that satisfy their delivery criteria

Token Loss timeout expired:
  call Shift_to_Gather

Token Retransmission timeout expired:
  retransmit token
  reset Token Retransmission timeout

Foreign message from processor q received:
  add message.sender_id to my_proc_set
  call Shift_to_Gather

Join message from processor q received:
  same as in Gather state except call Shift_to_Gather regardless of Join message's content

Commit token received:
  discard the Commit token
```

Fig. 3: The pseudocode executed by a processor in the Operational state.

```
Regular token or regular message received:
  same as in Operational state

Foreign message from processor q received:
  if q not in my_proc_set then
     add message.sender_id to my_proc_set
     call Shift_to_Gather

Join message from processor q received:
  if my_proc_set = message.proc_set and my_fail_set = message.fail_set then
     consensus[q] := true
      if for all r in my_proc_set − my_fail_set
            consensus[r] = true and my_id = smallest id of my_proc_set − my_fail_set then
        token.ring_id.seq := (maximum of my_ring_id.seq and Join ring_seqs) + 4
        token_memb := my_proc_set − my_fail_set
        call Shift_to_Commit
     else return
  else if message.proc_set subset of my_proc_set and
           message.fail_set subset of my_fail_set then return
  else if q in my_fail_set then return
  else
     merge message.proc_set into my_proc_set
     if my_id in message.fail_set then
        add message.sender_id to my_fail_set
     else
        merge message.fail_set into my_fail_set
     call Shift_to_Gather

Commit token received:
  if my_proc_set − my_fail_set = token.memb and token.seq > my_ring_id then
     call Shift_to_Commit

Join timeout expired:
  broadcast Join message with my_proc_set, my_fail_set, seq = my_ring_id.seq
  set Join timeout

Consensus timeout expired:
  if consensus not reached then
     for all r such that consensus[r] ≠ true do
         add r to my_fail_set
     call Shift_to_Gather
  else
     for all r do
        consensus[r] := false
     consensus[my_id] := true
     set Token Loss timeout

Token Loss timeout expired:
  execute code for Consensus timeout expired in Gather state
  call Shift_to_Gather
```

Fig. 4: The pseudocode executed by a processor in the Gather state.

longer accepts incoming Join messages. A processor is also considered to have reached consensus when it has received a Commit token with the same membership as *my_proc_set* − *my_fail_set*. The processors in that difference constitute the membership of the proposed new ring. If the Consensus timeout expires before a processor has reached consensus, it adds to *my_fail_set* all of the processors in *my_proc_set* from which it has not received a Join message with *proc_set* and *fail_set* equal to its own sets, returns to the Gather state, and tries to reach consensus again by broadcasting Join messages.

When a processor has reached consensus, it determines whether it has the lowest processor identifier in the membership and, thus, is the representative of the proposed new ring. If it is the representative, the processor generates a Commit token. It determines the *ring_id* of the new ring, which is composed of a ring sequence number equal to four plus the maximum of the ring sequence numbers in any of the Join messages used to reach consensus and its own ring sequence number. (The sequence number two less than that of the new ring is used as the transitional configuration identifier.) The representative also determines the *memb_list* of the Commit token, which specifies the membership of the new ring and the order in which the token will circulate, with the representative placed first. It then transmits the Commit token and shifts to the Commit state (Figure 7).

When a processor other than the representative has reached consensus, if it has not received the Commit token, the processor sets the Token Loss timeout, cancels the Consensus timeout, and continues in the Gather state waiting for the Commit token. If the Token Loss timeout expires, the processor returns to the Gather state and tries to reach consensus again. On receiving the Commit token, the processor compares the proposed membership, given by the *memb_list* field of the Commit token, with *my_proc_set* − *my_fail_set*. If they differ, the processor discards the Commit token, returns to the Gather state, and repeats the attempt to form a new ring. If they agree, the processor extracts the *ring_id* for the new ring from the Commit token, sets the fields in its entry of *memb_list*, increments the *memb_index* field, and shifts to the Commit state.

Commit State. In the Commit state (Figure 5), the first rotation of the Commit token around the proposed new ring confirms that all members in the *memb_list* of the Commit token are committed to the membership. It also collects information needed to determine correct handling of the messages from the old ring that still require retransmission when the membership protocol was invoked.

The second rotation of the Commit token disseminates the information collected in the first rotation. On receiving the Commit token for the second time, a processor shifts to the Recovery state (Figure 7) and writes the sequence number, *my_ring_id.seq*, for its new ring into stable storage.

Recovery State. In the Recovery state (Figure 6), when the representative receives the Commit token after its second rotation, it converts the Commit token into the regular token for the new ring, replacing the *memb_list* and *memb_index* fields by the *rtr* field. At this point, the new ring is formed but not yet installed, and the execution of the recovery protocol begins.

The processors use the new ring to retransmit messages from their old rings that must be exchanged to maintain agreed and safe delivery guarantees. In one

```
Regular token received: discard token

Regular message received: same as in Operational state

Foreign message received: discard message

Join message from processor q received:
  if q in my_new_memb and message.ring_seq ≥ my_ring_id.seq then
     execute code for receipt of Join message in Gather state
     call Shift_to_Gather

Commit token received:
  if token.seq = my_ring_id.seq then
     call Shift_to_Recovery

Join timeout expired: same as in Gather state

Token Loss timeout expired: call Shift_to_Gather
```

Fig. 5: The pseudocode executed by a processor in the Commit state.

atomic action, each processor delivers the exchanged messages to the application along with Configuration Change messages, installs the new ring, and shifts to the Operational state (Figure 7). The recovery protocol is described in more detail in Section 7.

When a processor starts or restarts, it first forms and installs a singleton ring, consisting of only the processor itself. The processor then broadcasts a Join message containing the value of *my_ring_id.seq*, obtained from its stable storage, and shifts to the Gather state.

The membership protocol described above is guaranteed to terminate in bounded time because *proc_set* and *fail_set* increase monotonically within a fixed finite domain, because timeouts bound the time that a processor spends in each of the states, and because an additional failure (which increases the *fail_set*) is forced to prevent the repetition of a proposed membership. In the base case, the membership, *proc_set* − *fail_set*, consists of a single processor identifier.

7. THE RECOVERY PROTOCOL

The objective of the recovery protocol is to recover the messages that had not been received when the membership protocol was invoked, and to enable the processors transitioning from the same old configuration to the same new configuration to deliver the same messages from the old configuration. The recovery protocol also provides message delivery guarantees, and thus maintains extended virtual synchrony, during recovery from failures. Maintenance of extended virtual synchrony is essential to applications such as fault-tolerant distributed databases.

7.1 The Data Structures

The recovery protocol uses the following data structures in addition to those above.

Regular Token Field

The regular token has the following additional field:

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

```
Regular token received:
  same as in Operational state except get messages from retrans_message_queue
  instead of new_message_queue and before forwarding the token execute:

  if retrans_message_queue is not empty then
      if token.retrans_flg = false then
         token.retrans_flg := true
  else if token.retrans_flg = true and I set it then
         token.retrans_flg := false
  if token.retrans_flg = false then
     increment my_retrans_flg_count
  else my_retrans_flg_count := 0
  if my_retrans_flg_count = 2 then
     my_install_seq := token.seq
  if my_retrans_flg_count ≥ 2 and my_aru ≥ my_install_seq and my_received_flg = false then
     my_received_flg := true
     my_deliver_memb := my_trans_memb
  if my_retrans_flg_count ≥ 3 and token.aru ≥ my_install_seq on last two rotations then
     call Shift_to_Operational

Regular message received:
  reset Token Retransmission timeout
  add message to receive_message_queue
  update my_aru
  if retransmitted message from my old ring then
     add to receive_message_queue for old ring
     remove message from retrans_message_queue for old ring

Foreign message from processor q received:
  discard message

Join message from processor q received:
  if q in my_new_memb and message.ring_seq ≥ my_ring_id.seq then
     execute code for receipt of Join message in Commit state
     execute code for Token Loss timeout expired in Recovery state

Commit token received by new representative:
  convert Commit token to regular token
  if retrans_message_queue is not empty then
     token.retrans_flg := true
  else token.retrans_flg := false
  forward regular token
  reset Token Loss and Token Retransmission timeouts

Token Loss timeout expired:
  discard all new messages received on the new ring
  empty retrans_message_queue
  determine current old ring aru (it may have increased)
  call Shift_to_Gather

Token Retransmission timeout expired:
  retransmit token
  reset Token Retransmission timeout
```

Fig. 6: The pseudocode executed by a processor in the Recovery state.

Local Variables

The recovery protocol also depends on the following local variables:

- *my_new_memb*: The set of identifiers of processors on the processor's new ring.
- *my_trans_memb*: The set of identifiers of processors that are transitioning from the processor's old ring to its new ring.
- *my_deliver_memb*: The set of identifiers of processors whose messages the processor must deliver in the transitional configuration.
- *low_ring_aru*: The lowest aru for the old ring for processors in *my_deliver_memb*.
- *high_ring_delivered*: The highest message sequence number such that some processor delivered the message with that sequence number as safe on the old ring.
- *my_install_seq*: The largest new ring sequence number of any old ring message transmitted on the new ring.
- *retrans_message_queue*: A queue of messages from the old ring awaiting retransmission to ensure that all remaining processors from the old ring have the same set of messages.
- *my_retrans_count*: The number of successive token rotations on which the processor has received the token with *retrans_flg* false.

7.2 The Protocol

A processor executing the recovery protocol takes the following steps:

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

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

(3) Deliver the first Configuration Change message, which initiates the transitional configuration.

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

(5) Deliver the second Configuration Change message, which initiates the new regular configuration.

(6) Shift to the Operational state.

Steps 2 through 6 involve no communication with other processors and are performed as one atomic action. The pseudocode executed by a processor to complete these steps is given in Figure 6.

Exchange of Messages from the Old Ring

In the first step of the recovery protocol, each processor determines the lowest *my_aru* of any processor from its old ring that is also a member of the new ring. The processor then broadcasts on the new ring every message for the old ring that it has received and that has a sequence number greater than the lowest *my_aru*. The *retrans_flg* field in the token is used to determine when all old ring messages have been retransmitted. This exchange of messages ensures that each processor receives as many messages as possible from the old ring.

```
Shift_to_Gather:
   broadcast Join message containing my_proc_set, my_fail_set, seq = my_ring_id.seq
   cancel Token Loss timeout and Token Retransmission timeouts
   reset Join and Consensus timeouts
   for all r in my_proc_set do
     consensus[r] := false
   consensus[my_id] := true
   state := Gather
```

```
Shift_to_Commit:
   update memb_list in Commit token with my_ring_id, my_aru, my_received_flg, my_high_delivered
   my_ring_id := Commit token ring_id
   forward Commit token
   cancel Join and Consensus timeouts
   reset Token Loss and Token Retransmission timeouts
   state := Commit
```

```
Shift_to_Recovery:
   forward Commit token for the second time
   my_new_memb := membership in Commit token
   my_trans_memb := members on old ring transitioning to new ring
   if for some processor in my_trans_memb received_flg = false then
     my_deliver_memb := my_trans_memb
     low_ring_aru := lowest aru for old ring for processors in my_deliver_memb
     high_ring_delivered := highest sequence number of message delivered for old ring
        by a processor in my_deliver_memb
     copy all messages from old ring with sequence number > low_ring_aru
        into retrans_message_queue
   my_aru := 0
   my_aru_count := 0
   reset Token Loss and Token Retransmission timeouts
   state := Recovery
```

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

Fig. 7: The pseudocode executed by a processor when shifting between states.

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

Delivery of Messages on the Old Ring

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

The processor sorts the messages for the old ring that were broadcast on the new ring into the order of their sequence numbers on the old ring, and delivers messages in order until it encounters a gap in the sorted sequence or a message requiring safe delivery with a sequence number greater than *high_ring_delivered*. Messages beyond this point cannot be delivered as safe on the old ring, but may be delivered in a transitional configuration.

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

Delivery of Messages in the Transitional Configuration

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

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

Failure of Recovery

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

When a processor delivers a message in safe order in a transitional configuration, it must have a guarantee that each of the other members of the configuration will deliver the message before it installs the new ring, unless that processor fails. If a processor does not install the new ring, it will proceed in due course to install a dif-

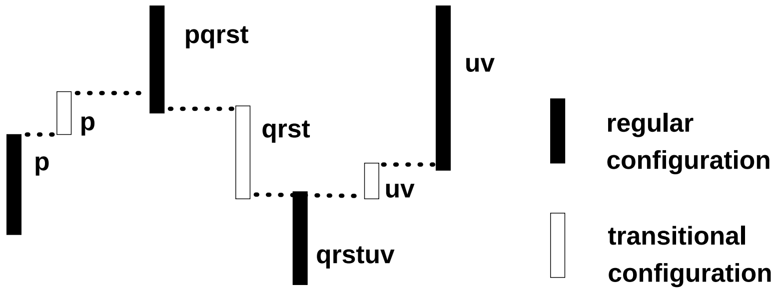

Fig. 8: Regular and transitional configurations. The vertical lines represent total orders of messages, and the dashed horizontal lines represent Configuration Change messages.

ferent new ring with a corresponding transitional configuration. It must deliver the message in that transitional configuration in order to honor the delivery guarantee.

Thus, if a processor finds the *received_flg* in the Commit token set to true for every processor in *my_trans_memb*, it must retain the old ring messages originated by members of *my_deliver_memb* and deliver them as safe in *my_trans_memb*, the transitional configuration for the new ring that it actually installs. Note that *my_trans_memb* is a subset of *my_deliver_memb* and that *my_trans_memb* must decrease on successive passes through the Recovery state before a new ring is installed.

7.3 An Example

As shown in Figure 8, a ring containing processors p, q, r, s and t partitions, so that p becomes isolated while q, r, s and t merge into a new ring with u and v. Processors q, r, s and t successfully complete the recovery protocol and deliver two Configuration Change messages, one to switch from the regular configuration $\{p, q, r, s, t\}$ to the transitional configuration $\{q, r, s, t\}$ and one to switch from the transitional configuration $\{q, r, s, t\}$ to the regular configuration $\{q, r, s, t, u, v\}$.

Processors q, r, s and t may not be able to deliver all of the messages originated in the regular configuration $\{p, q, r, s, t\}$, because they may not have received some of the messages from p before p became isolated; however, it can be guaranteed that they deliver all of the messages originated by a processor in the transitional configuration $\{q, r, s, t\}$. Similarly, processors q, r, s and t may not be able to deliver a message as safe in the regular configuration $\{p, q, r, s, t\}$ because they may have no information as to whether p had received the message before it became isolated; however, it can be guaranteed that they deliver the message as safe in the transitional configuration $\{q, r, s, t\}$. The first Configuration Change message separates the messages for which delivery guarantees can be provided in the regular configuration $\{p, q, r, s, t\}$ from the messages for which delivery guarantees apply in the reduced transitional configuration $\{q, r, s, t\}$.

Extended virtual synchrony does not, of course, solve all the problems of maintaining consistency in a fault-tolerant distributed system, but it does provide a foundation upon which these problems can be solved. Consider, for example, the set of messages delivered by processor p. Prior to the first Configuration Change message delivered by p to terminate the regular configuration $\{p, q, r, s, t\}$, there are no missing messages. In the transitional configuration, p delivers all remaining messages originated by itself and also other messages that have become safe. There may, however, be messages broadcast by processors q, r, s and t that are not available to, and are not delivered by, processor p. After the second Configuration Change message, p is a member of the regular configuration $\{p\}$, and p does not deliver messages from the other processors.

When p rejoins the other processors in some subsequent configuration, the application programs must update their states, using application-specific algorithms, to reflect activities that were not communicated while the system was partitioned. The Configuration Change messages warn the application that a membership change occurred, so that the application programs can take appropriate actions based on the membership change. Extended virtual synchrony guarantees a consistent order of message delivery, which is essential if the application programs are to reconcile their states following repair of a failed processor or remerging of a partitioned network.

8. THE FLOW CONTROL MECHANISM

The Totem protocol is designed to provide high performance under high load. The performance measures we consider are throughput (messages ordered per second) and latency (delay from message origination to delivery in agreed or safe order). Effective flow control is required to achieve the desired performance.

With point-to-point communication, positive acknowledgment protocols, such as the sliding-window protocol, have been refined to provide excellent flow control. However, with broadcast and multicast communication, positive acknowledgment protocols result in excessive numbers of acknowledgments. Rate-controlled protocols have attracted attention recently, but have the disadvantage for broadcast and multicast communication that the transmission rate must be set for each processor individually rather than for the multicast group. With bursty communication, the maximum transmission rate for a processor must be set to a value that is unacceptably low, even when other processors have few messages to transmit.

A basic characteristic of reliable ordered broadcast and multicast protocols is that the rate of broadcasting messages cannot exceed the rate at which the slowest processor can receive messages. At higher rates of broadcasting, the input buffer of the slowest processor will become full and messages will be lost. In our experience, this is the primary cause of message loss. Retransmission of lost messages increases the message traffic and reduces the effective transmission rate.

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

When the rate of broadcasting is not equally spread across all processors, the protocol can be modified to allow the token to visit more than once per token rotation those processors that have the highest transmission rates, and to allow those processors to transmit more messages on each visit. To minimize the latency, the rate at which the token visits a processor should be approximately proportional to the square root of the rate at which the processor broadcasts [6].

8.1 The Data Structures

Regular Token Fields

The flow control mechanism depends on two fields of the regular token:

- *fcc*: A count of the number of messages broadcast by all processors during the previous rotation of the token.
- *backlog*: The sum of the number of new messages waiting to be transmitted by each processor on the ring at the time at which that processor forwarded the token during the previous rotation.

Flow Control Constants

The flow control mechanism also depends on two global constants:

- *window_size*: The maximum number of messages that all processors are allowed to broadcast in any token rotation.
- *max_messages*: The maximum number of messages that each processor is allowed to broadcast during one visit of the token.

These constants can be determined analytically from the number of processors and their characteristics, or can be negotiated during ring formation. The constant *max_messages* may be different for different processors.

Local Variables

Each processor maintains the following local variables:

- *my_trc*: The number of messages broadcast by this processor on this rotation of the token (my this rotation count).
- *my_pbl*: The number of new messages waiting to be transmitted by this processor when it forwarded the token on the previous rotation (my previous backlog).
- *my_tbl*: The number of new messages waiting to be transmitted by this processor when it forwards the token on this rotation (my this backlog).

The values of *my_pbl* and *my_tbl* are limited by the amount of buffer space available for messages awaiting transmission.

8.2 The Algorithm

The value of *my_trc*, the number of messages broadcast by this processor on this token rotation, is subject to the following constraints:

$my\_trc \leq max\_messages$: The number of messages broadcast by this processor must not exceed the maximum number it is allowed to broadcast during one visit of the token.

$my\_trc \leq window\_size - fcc$: The number of messages broadcast by this processor must not exceed the window size minus the number of messages broadcast in the previous rotation of the token.

$my\_trc \leq window\_size \times my\_tbl/(backlog + my\_tbl - my\_pbl)$: The number of messages broadcast by this processor must not exceed its fair share of the window size, based on the ratio of its backlog to the sum of the backlogs of all the processors as they released the token during the previous rotation.

The backlog mechanism achieves a more uniform transmission rate for individual processors under high, but not overloaded, traffic conditions than does the simple window-size mechanism used by FDDI.

9. IMPLEMENTATION AND PERFORMANCE

The Totem single-ring reliable ordering and membership protocol has been implemented in the C programming language on a network of Sun 4/IPC workstations connected by an Ethernet. The implementation uses only standard Unix features and is highly portable. It has been transferred from our Sun workstations to DEC and SGI workstations and has worked with little modification.

The implementation uses the UDP broadcast interface in the Unix operating system SunOS 4.1.1 with Sun's default kernel allocations. One UDP socket is used for all broadcast messages, and a separate UDP socket is used by each processor to receive the token from its predecessor on the ring. The input buffers are a combination of the buffers managed by the physical Ethernet controller and those managed by the Unix UDP service. Four bytes of stable storage are required to store the ring sequence number.

We have measured the performance of our implementation on a network of five Sun 4/IPC workstations, each ready to broadcast at all times with minimal extraneous load on the processors and on the Ethernet. For each measurement, the *window_size* and *max_messages* were adjusted to the maximum values for which message loss is negligible in order to maximize throughput.

The throughput was measured for equal numbers of messages broadcast by all processors on the ring and also for all messages broadcast by a single processor. As the left graph of Figure 9 shows, there was little difference in the throughput. With 1024 byte messages, more than 800 messages are ordered per second. For smaller messages, over 1000 messages are ordered per second. The highest prior rates of message ordering for reliable totally ordered message delivery for 1024 byte messages are about 300 messages per second for the Transis system using the same equipment and for the Amoeba system using equipment of similar performance.

We have also investigated the latency from origination to delivery of a message in agreed and safe order; a detailed analysis can be found in [15]. The right graph of Figure 9 shows the mean latency to agreed and safe delivery for Poisson arrivals at lower, more typical loads for 1024 byte messages. At low loads (*e.g.*, 400 ordered messages per second which is much more than the maximum throughput for prior protocols), the latency to agreed delivery is under 10 milliseconds. Even at 50% useful utilization of the Ethernet (625 ordered messages per second), the latency to agreed delivery is still only about 13 milliseconds. In general, the latency to agreed delivery is approximately half the token rotation time and the latency to safe delivery is approximately twice the token rotation time, except at very high loads where the latency is dominated by queueing delays in the buffers.

Other performance characteristics of interest are the time to recover from token loss and the time to execute the membership protocol and reconfigure the system when failures occur. With the token retransmission mechanism enabled, the time to return to normal operation after loss of the token is on average 16 milliseconds. With the token retransmission mechanism disabled, loss of the token triggers a Token Loss timeout and forces a complete reformation of the membership. The

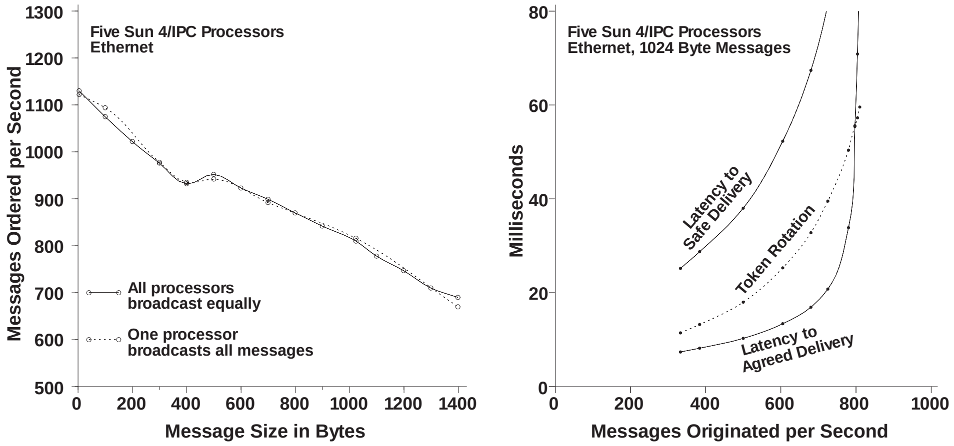

Fig. 9: To the left, the throughput as a function of message size. To the right, the latency to agreed and safe delivery as a function of load.

time to form a new ring, generate a new token, recover messages from the old ring, and return to normal operation is on average the Token Loss timeout plus 40 milliseconds, with the Token Loss timeout set to 100 milliseconds.

10. CONCLUSION

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

The membership protocol handles processor failure and recovery, and network partitioning and remerging. Extended virtual synchrony ensures consistent actions by processors that fail and are repaired with stable storage intact and in networks that partition and remerge. A recovery protocol that maintains extended virtual synchrony during recovery after a failure has been provided.

The flow control mechanism avoids message loss due to buffer overflow and provides significantly higher throughput than prior total ordering protocols. Given the high performance of Totem, there is no need to provide a weaker message ordering service, such as causally ordered delivery, because totally ordered agreed delivery can be provided at no greater cost. Moreover, applications can be programmed more easily and more reliably with totally ordered messages.

Continuing work on Totem is exploiting the single-ring protocol to provide more general services. Agreed and safe delivery services, as well as membership services, are being provided to multiple rings interconnected by gateways. It remains to be investigated whether the exceptional performance of the Totem single-ring protocol can be sustained when Totem is extended to multiple rings.

REFERENCES

[1] Amir, Y., Dolev, D., Kramer, S., and Malki, D. Membership algorithms in broadcast domains. In *Proceedings of the 6th International Workshop on Distributed Algorithms* (Haifa, Israel, November 1992), pp. 292–312.

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

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

[4] Amir, Y., Moser, L. E., Melliar-Smith, P. M., Agarwal, D. A., and Ciarfella, P. The Totem single-ring ordering and membership protocol. Tech. Rep. 94-19, Department of Electrical and Computer Engineering, University of California, Santa Barbara, August 1994.

[5] Birman, K. P., and Renesse, R. V. *Reliable Distributed Computing with the Isis Toolkit*. IEEE Computer Society Press, March 1994.

[6] Boxma, O. J., Levy, H., and Weststrate, J. A. Optimization of polling systems. In *Performance '90, Proceedings of the 14th IFIP WG 1.3 International Symposium on Computer Performance Modelling, Measurement and Evaluation* (Edinburgh, UK, September 1990), pp. 1–17.

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

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

[9] Kaashoek, M. F., and Tanenbaum, A. S. Group communication in the Amoeba distributed operating system. In *Proceedings of the IEEE 11th International Conference on Distributed Computing Systems* (Arlington, TX, May 1991), pp. 882–891.

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

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

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

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

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

[15] Moser, L. E., and Melliar-Smith, P. M. Probabilistic bounds on message delivery for the Totem single-ring protocol. In *Proceedings of the 15th IEEE Real-Time Systems Symposium* (San Juan, Puerto Rico, December 1994), pp. 238–248.

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

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

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

[19] van Renesse, R., Hickey, T. M., and Birman, K. P. Design and performance of Horus: A lightweight group communications system. Tech. Rep. 94-1442, Cornell University, Department of Computer Science, August 1994.