Fast Message Ordering and Membership Using a Logical Token-Passing Ring

Y. Amir, L. E. Moser, P. M. Melliar-Smith, D. A. Agarwal, P. Ciarfella
Department of Electrical and Computer Engineering
University of California, Santa Barbara, CA 93106

Abstract

Many protocols exist to support the maintenance of consistency of data in fault-tolerant distributed systems; these protocols are quite expensive and thus have not been widely adopted. The Totem protocol supports consistent concurrent operations by placing a total order on broadcast messages. This total order is achieved by including a sequence number in a token circulated around a logical ring that is imposed on a set of processors in a broadcast domain. A membership algorithm handles reconfiguration, including restarting of a failed processor and remerging of a partitioned network. Effective flow-control allows the protocol to achieve message ordering rates two to three times higher than the best prior protocols.

1 Introduction

As distributed systems become more widely used, improved performance and more reliable operation will be required. Of primary importance are the problems of maintaining the consistency of data and of coordinating the activities of processors in the network. These problems are difficult when asynchronism, fault-tolerance and performance must be taken into account.

Protocols exist that address these problems in fault-tolerant asynchronous distributed systems. These protocols are, however, quite expensive in the number of messages broadcast and/or the computation required. Recent designs for fault-tolerant distributed systems [5, 13, 17] employ the idea of placing a partial order or a total order on broadcast messages. This approach simplifies the design of the application but depends heavily on efficient fault-tolerant protocols to keep the overhead reasonable.

The Totem protocol provides fast reliable totally ordered delivery of messages in a *broadcast domain* in which every message is transmitted to all processors in the domain. Reliable totally ordered delivery is achieved by circulating a token around a logical ring imposed on the broadcast domain. Only the processor in possession of the token can broadcast a message to the other processors on the ring. Under low loads, token-passing protocols, such as Totem, can exhibit longer latency from generation to ordering of a message. Under high loads, the token-based flow-control strategy of Totem provides about three times the throughput of, and lower latency than, prior total ordering protocols.

The Totem protocol also determines processor membership of the logical ring and handles token loss, processor failure and restart, and partitioning and remerging of the ring. Unless the token is lost, processors can continue to broadcast messages during reconfiguration up to the point at which the token for the old ring is replaced by the token for the new ring. Also important are the provisions for virtual synchrony and extended virtual synchrony which ensure that messages are not lost or incorrectly ordered as a result of reconfiguration.

This work is based on prior experience with the Trans and Total protocols [13, 15, 16], which were effective but computationally expensive and led to the design of the Totem protocol [1, 12]. Ideas from the Trans protocol also influenced the development of the Transis system [2, 3]. The work presented here extends the membership algorithm of Transis to handle ring formation and token generation and combines that algorithm with the single-ring total ordering algorithm of Totem to obtain a highly efficient fault-tolerant total ordering protocol.

2 Related Work

An early reliable broadcast and ordering protocol by Chang and Maxemchuk [6] uses a token-passing strategy. In their protocol the processor that holds the token acknowledges messages, whereas in Totem the processor that holds the token broadcasts messages. Chang and Maxemchuk have also provided a membership and token recovery algorithm. Typically, between two and three messages are required to order a message in an optimally loaded system. While the latency is reasonable 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]. Like Totem, TPM uses a token on a logical ring for broadcasting and retransmission of messages. They have also provided a membership and token regeneration protocol. In the absence of processor failure and network partitioning,

The address of Y. Amir is Computer Science Department, The Hebrew University of Jerusalem, Israel. The address of P. Ciarfella is Digital Equipment Corporation, Networks Engineering, Littleton, MA.

the TPM protocol achieves safe delivery (defined in Section 3) but requires, on average, two and one-half token rotations. TPM addresses network partitioning, but delivers more messages than are allowed by causal delivery and virtual synchrony.

The ISIS system of Birman and Joseph [4] is based on the idea of ordering messages multicast to process groups. Recent versions [5] have adopted a token-passing protocol, similar to that of Chang and Maxemchuck, for total ordering. ISIS has focused on the application program interface and provides several types of services of increasing cost, reliability and synchronization: BCAST or unordered messages, CBCAST or causally ordered messages, and ABCAST or totally ordered messages. ISIS has also established the idea of virtual synchrony as important to maintaining consistency in fault-tolerant distributed systems.

Peterson, Buchholz and Schlichting [17] have devised the Psync protocol, which constructs a partial order on messages that can be converted into a total order. In contrast, the Totem protocol constructs a total order on messages directly without constructing a partial order first. Mishra, Peterson and Schlichting [14] have developed a membership algorithm based on the partial order of Psync.

An early use of synchronous behavior in the communication system to support fault-tolerant distributed applications was the work of Cristian, Aghili, Strong and Dolev [8]. They fabricated an atomic broadcast from unreliable message communication using loosely synchronized clocks and timeouts with an upper bound on message transit time. Cristian [7] has also developed a membership algorithm using a strategy similar to that used for atomic broadcast. The Totem protocol likewise uses a timeout mechanism to detect token loss and processor failure.

The approach of Kopetz *et al* [9] is even more synchronous than that of Cristian, being specifically directed towards real-time applications. The result is higher performance and less flexibility. Kopetz [10] has used this synchronous approach to provide membership services in addition to reliable ordered message delivery. The Totem protocol provides equally high performance with greater flexibility.

Related to our flow-control strategy are the sliding-window strategy and the token rotation time limit of FDDI. The use of a token and a window for flow control on a broadcast medium has not been investigated in prior work known to us.

3 Services

The Totem protocol provides reliable totally ordered message delivery and membership services to higher layers of the communication protocol hierarchy, referred to here as the application. The protocol accommodates message loss (including token loss), processor failure and restart, and network partitioning and remerging. Malicious faults are assumed not to occur. In addition, we assume that the probability that a processor receives a broadcast message is greater than some positive constant. We use the terms receive and deliver as follows. A processor *receives* a message that was broadcast by a processor in the broadcast domain and a processor *delivers* messages in order to the application.

3.1 Reliable Ordered Delivery Services

The Totem protocol provides two types of reliable ordered message delivery, *agreed* and *safe*. Both of these services provide total ordering of messages. Safe delivery requires, in addition, that the processor delivering the message know that all other processors in the configuration have received and will deliver the message. Safe delivery is the best that any communication subsystem can provide and may be required by the application. However, safe delivery incurs a larger latency than agreed delivery. In Totem, the mean latency from generation of a message to agreed delivery is approximately half the token rotation time, while the mean latency to safe delivery is approximately one and one-half token rotation times.

Reliable ordered delivery services can be categorized into five levels of service: basic, fifo, causal, agreed and safe. The *basic* level of service guarantees that messages are delivered to the application (without regard for ordering). Every processor that receives a basic message can deliver that message immediately. The *fifo* level of service ensures that messages originated by a given processor are delivered in the order in which they were originated. However, messages from different processors can be arbitrarily interleaved. The *causal* level of service, which is taken from Lamport [11], is the reflexive transitive closure of the relation:

- Message m precedes message m' if processor p delivers m before p sends m'.
- Message m precedes message m' if processor p sends m before p sends m'.

We now provide the definitions of agreed and safe delivery.

Definition. A message m is delivered by a processor p in *agreed* order in configuration C if and only if (1) p is a member of C, p has received m, and m was originated by a member of C or of a configuration that precedes C, (2) p does not deliver two different messages at the same position in agreed order in C nor does it deliver one message at two different positions in agreed order in C, (3) p has delivered all messages that precede m in causal order in C, (4) p has delivered all messages that precede m in agreed order in C, and (5) for any other processor q in C, if p delivers m before n in agreed order in C, then q does not deliver n before m in agreed order in C.

Definition. A message m is delivered by a processor p in *safe* order in configuration C if and only if it is delivered by p in agreed order in C and if p knows that all processors in C have received and will deliver m.

It is easy to see that safe service is also agreed service, that agreed service is also causal service, that causal service is also fifo service and that fifo service is also basic service.

Many systems provide some of these levels of service. In particular, the ISIS system [5] provides CBCAST and ABCAST service, corresponding approximately to causal and agreed service, respectively. ISIS also provides an all-stable service, corresponding approximately to safe service. The Transis system [2] provides causal, agreed and safe levels of service.

The use of one of the lower levels of service, i.e. basic, fifo or causal rather than agreed, has been justified for other protocols by the earlier delivery achieved. In Totem, a processor can deliver a message broadcast on the ring in agreed order immediately upon receipt, provided that it has already delivered all messages on the ring that precede that message in the order. Safe delivery provides a higher level of service than agreed delivery, and does incur a latency penalty. Thus, we provide agreed and safe delivery and no others.

3.2 Membership Services

In a distributed system processors may fail and restart and the network may partition and remerge. Thus, the need exists for a service that maintains consensus on the set of processors currently connected and working. Some distributed applications may need to know about configuration changes. This information can be provided by configuration change messages, which are delivered in order along with broadcast messages. Such a message is regarded as the terminal message of an existing configuration and the initial message of a new configuration. The configuration change messages define a partial order on the configurations. These messages support our implementation of virtual synchrony, a concept introduced by Birman [5]. We define virtual synchrony as follows:

Definition. *Virtual synchrony* requires that if processors p and q are both members of consecutive configurations C and D, then p and q determine exactly the same agreed order of messages in C.

Virtual synchrony suffices for systems like ISIS that do not allow network partitioning and only consider processors that fail and never recover or regard repaired processors as new processors. If a configuration C partitions into configurations D and E, the existing concept of virtual synchrony might allow configurations D and E to deliver messages in inconsistent orders. Such behavior can be observed in existing systems and causes problems for database designers. Similar problems can arise if a processor fails. In effect, immediately prior to the failure, the processor becomes isolated and may order messages inconsistently with the other processors. Consequently, we extend the definition of virtual synchrony to constrain the behavior of processors that are members of non-consecutive configurations.

Definition. *Extended virtual synchrony* is virtual synchrony with the additional requirement that, if processors p and q both order messages m and n, then they determine the same relative order of m and n.

Note that messages m and n are not necessarily ordered by p and q in the same configuration or even in consecutive configurations.

As a configuration changes due to processor failure or network partitioning, it is inevitable that some messages may be received by some processors but not by others. Care is required to ensure that delivery of messages still meets the requirements of causal, safe and agreed orders. In Totem, a processor delivers two configuration change messages to the application. These messages not only inform the application of the changes but also define the configurations for which virtual synchrony holds.

4 The Total Ordering Algorithm

The Totem single-ring ordering algorithm operates on a broadcast domain that consists of a finite number of processors. Each processor has a unique identifier, which does not change if a processor fails and restarts. A configuration or ring consists of an ordered set of processors within the broadcast domain. Each ring has a unique ring identifier consisting of the smallest identifier of any processor on the ring and the sequence number of the first message broadcast on that ring; at any point in time a processor is a member of only one ring. Each message has an identifier consisting of the identifier of the processor originating the message, the identifier of the ring on which the message was originated, and a sequence number.

For the total ordering algorithm described in this section we assume that the token is never lost, that the ring does not become partitioned and that processor failures do not occur; however, messages may be lost. In Section 5 we relax these assumptions and extend the algorithm to handle token loss, processor failure and restart, and partitioning and remerging of the ring.

The total ordering of messages is achieved by using a single sequence of message sequence numbers for all processors on the ring and by including the sequence number of the last message broadcast in the token. The token contains the following fields:

- *type*: Regular.
- *seq*: The highest sequence number of any message that has been broadcast on the ring, i.e. a high-water mark. The *seq* is initially one less than the sequence number of the first message on the ring.
- *aru*: A sequence number (all-received-up-to) such that all processors on the ring have received all messages up to (and including) the message with this sequence number, i.e. a low-water mark. It is used to control the discarding of messages that have been received by all processors on the ring and that will, therefore, not need to be retransmitted. The *aru* is initially one less than the sequence number of the first message broadcast on the ring.
- *rtr*: A retransmission request list, containing one or more retransmission requests.
- *flg*: A flag that is used during reconfiguration to achieve extended virtual synchrony. The *flg* is initially set to false.

- *fcc*: The number (flow control count) of messages actually broadcast by all processors on the ring in the last rotation of the token, including retransmissions.

Each processor maintains a local variable *my_aru* containing the sequence number of the message such that it has received all messages with sequence numbers at most equal to that sequence number. Initially, a processor sets *my_aru* to be one less than the sequence number of the first message on the ring. As the processor receives messages, it updates *my_aru*. Each processor maintains a list of messages that it has received; messages that are safe can be discarded from this list.

On receipt of the token, a processor completes the processing of messages in its input buffer, broadcasts messages, updates the token and transmits it to the next processor on the ring. For each new message it broadcasts, a processor increments the *seq* field of the token and sets the sequence number of the new message to this *seq*.

Whether broadcasting a message or not, a processor compares the *aru* field of the token with *my_aru* and, if *my_aru* is smaller, replaces *aru* with *my_aru*. If the processor previously lowered the *aru* and the token returned with the same value as it set, then it sets *aru* equal to *my_aru*. If *seq* and *aru* are equal, then it increments *aru* and *my_aru* in step with *seq*.

If the *seq* field of the token indicates that messages have been broadcast that it has not yet received, a processor sets or augments the *rtr* field. If the processor has received messages that appear in the *rtr* field then, for each such message, it generates an independent random variable to determine whether it should retransmit that message before broadcasting new messages. (This randomization increases overall system reliability.) When it retransmits a message, the processor removes the sequence number of that message from the *rtr* field.

The *flg* field is used to achieve extended virtual synchrony, as described in Section 5. The *fcc* field provides the flow control for the protocol, as described in Section 6.

If a processor has received and delivered every message with sequence number less than that of message m, then it can deliver m in agreed order. If a processor can deliver m in agreed order and if on two successive rotations, it releases the token with an *aru* no less than the sequence number of m, then it can determine that m is safe. The proof of correctness of the basic total ordering algorithm of Totem has been omitted.

5 The Membership Algorithm

The membership algorithm presented here is used in conjunction with the total ordering algorithm of Section 4. The algorithm handles all aspects of processor membership, including processor failure and restart, token loss, and network partitioning and remerging.

The algorithm differs from other membership algorithms in several important ways. The algorithm uses a single representative for each ring that is being merged; this representative negotiates the membership of the new ring on behalf of the other processors on the old ring and should not be regarded as a leader or master of the old or new rings. While the new ring is being formed, the old ring is used as long as possible to broadcast messages. Before it is installed, the new ring is used to recover messages for the old ring that must be delivered to achieve extended virtual synchrony.

The membership algorithm uses three types of special messages:

- Attempt Join message broadcast by a representative initiating the membership algorithm to form a new ring from two or more rings.
- Join message broadcast by a representative proposing a set of representatives of old rings and also a set of failed representatives (a subset of the set of representatives). The proposed new ring will be formed by the representatives in the first set but not in the second.
- Configuration Change message that enumerates the membership of a new ring or of an interim configuration consisting of processors on both old and new rings.

The Attempt Join and Join messages have no sequence numbers and are not delivered to the application. The Configuration Change messages are used to ensure extended virtual synchrony. They are generated locally at each processor and are delivered directly to the application without being broadcast.

In forming a new ring the representative of that ring generates a Form token. The Form token, which differs from the regular token of the total ordering algorithm, contains the following fields:

- *type*: Form.
- *form_token_id*: The Form token identifier, which consists of the identifier of the representative of the new ring and a Form token sequence number. This sequence number is necessary to ensure that there is a unique Form token for the proposed new ring. It is determined by the representative and incremented with each Form token the representative generates.
- *join_list*: A sorted list of the representatives' identifiers.
- *join_index*: The index of the representative in the join_list that has just released the Form token. This index is used to connect the old rings together into a new ring (see Figure 2).
- *memb_list*: A list containing all the identifiers of all the members of the new ring according to their position on the new ring. For each of these members, this list also contains its old ring identifier and *my_aru*.
- *high_seq*: The larger of the highest sequence number of a message received from any of the old rings and of the highest first sequence number of any new ring containing a member of this ring. This will be the first sequence number of the new ring.

These fields are updated as the Form token circulates around the new ring, except for the *type* field, the *form_token_id* field and the *join_list*, which are set by the representative of the new ring.

5.1 The State Diagram

The membership algorithm is defined in terms of the state diagram shown in Figure 1. The definitions of events and states are given below.

5.1.1 Definition of Events

There are five membership events, namely:

- Receiving a *foreign* message, which can be a
  - Regular message broadcast by a processor that is not a member of the ring, or
  - Attempt Join message, or
  - Join message.
- Receiving a Form token. On the first receipt of the Form token a processor of the proposed new ring updates the Form token; on the second receipt it obtains the updated information that the other processors supplied.
- Token loss timeout. This timeout indicates that a processor did not receive either the token or a message from a processor on the ring in that amount of time.
- Gather timeout. This timeout is used to bound the time for gathering representatives to form a new ring.
- Commit timeout. This timeout indicates that a processor participating in the formation of a new ring failed to determine that consensus had been reached.

5.1.2 Definition of States

There are five states, namely:

- Operational state. This is the regular state of the ring in which the basic total ordering algorithm operates with no membership changes.
- Gather state. In the Gather state the representatives that will constitute the new ring are collected. This is done by gathering as many Attempt Join and Join messages as possible before the Gather timeout expires.
- Commit state. In the Commit state the representatives attempt to reach consensus on the set of representatives whose rings will be combined to form the proposed new ring.
- Form state. In the Form state the path of the token of the proposed new ring is determined and information about the members of that ring is exchanged.
- Recover state. In the Recover state messages that have to be delivered for the old rings are recovered to ensure that extended virtual synchrony is achieved. When extended virtual synchrony is achieved, the proposed new ring is installed as the new ring.

5.1.3 Formation of a New Ring

We first explain the membership algorithm without considering the effects of further processor failure or token loss during operation of the algorithm. Those effects are examined in Section 5.1.5.

The membership algorithm is invoked when a token loss is detected or when a foreign message is received by a processor on the ring. A processor that starts or restarts first forms a singleton configuration containing only itself and then broadcasts an Attempt Join message. A non-representative in the Operational state ignores foreign messages.

In the Operational state the ordering of messages proceeds according to the basic total ordering algorithm of Totem. When a foreign message is received by a representative, the representative initiates an Attempt Join message, advertizing its intention to form a bigger ring. It then shifts to the Gather state.

The Gather state allows time for the representative to collect together as many representatives as possible to form a new ring. The representative remains in the Gather state until the Gather timeout expires. It then broadcasts a Join message, containing the identifiers of the representatives it has collected in the Gather state, and shifts to the Commit state.

In the Commit state the representatives reach consensus about the set of representatives that will participate in the formation of the new ring. Each time that a representative broadcasts a Join message, it is committed to the set of representatives and the set of failed representatives in that message. If another representative broadcasts a Join message that contains at least one representative in either of those sets that is not in the corresponding sets of the first Join message, then the first Join message is effectively cancelled. (This is essential because the second representative will never agree on the sets of representatives in the first Join message.) The first representative will then broadcast, and commit to, a new Join message that contains the union of the sets of representatives and the union of the sets of failed representatives in the two Join messages.

A consensus is reached when there exists a set of representatives and a set of failed representatives, enumerated in a Join message, such that each of the non-failed representatives has broadcast a Join message with those two sets of representatives. When such a representative receives all those Join messsages, it knows that consensus has been reached. If the Commit timeout expires, no consensus has been reached. In this case the representative inserts into the set of failed representatives all representatives from which it has not received the required Join message. The representative then broadcasts a revised Join message, restarts the Commit timeout, and tries again to form a new ring.

The representative for the proposed new ring, chosen deterministically from among the representatives when consensus is reached, generates a Form token. The Form token circulates through all members of the proposed new ring along a cycle determined by the increasing order of identifiers of the representatives (see Figure 2). Every member of the proposed new ring

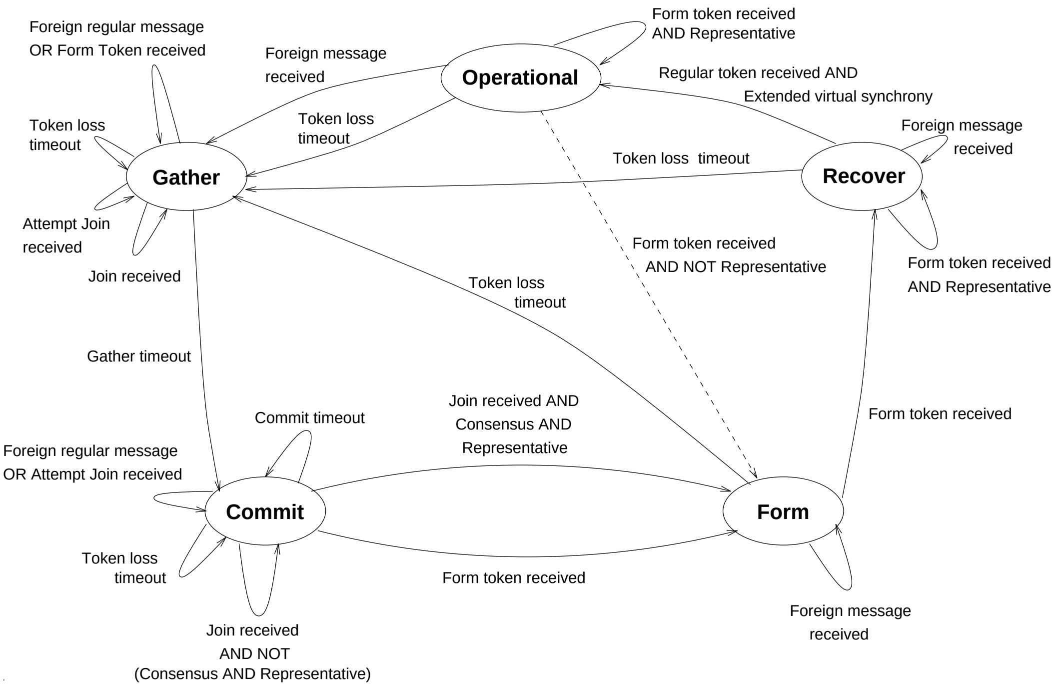

Figure 1: The state diagram for the membership algorithm.

shifts to the Form state as it forwards the Form token. This includes the non-representatives, which shift from the Operational state to the Form state, as shown by the dashed line in Figure 1. Having entered the Form state, a processor ignores new messages broadcast on the old ring and consumes the regular token for the old ring if it subsequently receives it.

After one rotation of the Form token, the representative of the new ring knows all of the information needed for the new ring and shifts to the Recover state. After two rotations of the Form token, all other processors on the proposed new ring have this information and have also shifted to the Recover state. This information includes the new ring identifier, which consists of the identifier of the representative of the new ring and *high_seq*, as well as the *my_aru* and old ring identifier of any processor that is a member of the proposed new ring. From this information each processor can calculate an interim configuration and a low *my_aru* for the processors in that interim configuration.

On receiving the Form token after its second rotation, the representative of the new ring consumes the Form token and transmits the regular token for the new ring in its place. At this point the new ring is formed but not yet installed, and the operation needed to achieve extended virtual synchrony begins.

5.1.4 Achieving Extended Virtual Synchrony

The objective of extended virtual synchrony is to provide strong guarantees about message delivery to the application, even in the presence of failures. We must guarantee that

- All messages that are delivered to the application are delivered in causal order.
- The messages delivered in agreed order in configuration C include all messages originated by processors in C up through the message delivered with the largest sequence number.
- The messages delivered in safe order in configuration C are delivered to the application in every processor of C unless that processor fails.
- All messages originated by a processor are delivered to the application in that processor unless that processor fails.

These guarantees allow an application, such as a fault-tolerant distributed database, to operate correctly despite configuration changes.

The extended virtual synchrony algorithm consists of six steps:

1. The processors of the new ring, that were members of the same old ring, exchange messages to ensure that they all have the same set of messages broadcast on the old ring but not yet delivered.
2. Each processor delivers to the application those messages that can be delivered in agreed or safe order on the old ring.
3. The processor delivers the first Configuration Change message, which enumerates the processors in a smaller interim configuration consisting of the processors on both the old and new rings.
4. The processor delivers to the application further messages that could not be delivered in agreed or safe order on the old ring but that can be delivered in agreed or safe order in the smaller interim configuration.

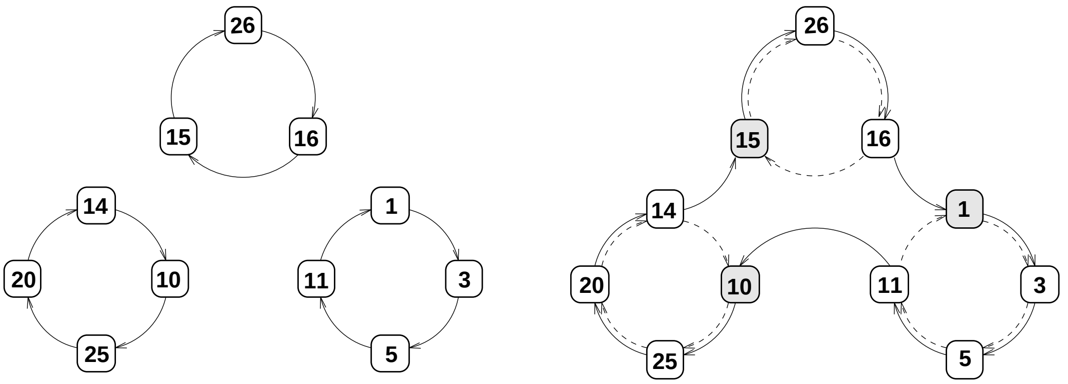

Figure 2: Determination of the cycle along which the token will circulate for the proposed new ring. The representatives of the old rings are shown shaded.

5. The processor delivers a second Configuration Change message, which enumerates the processors on the new ring.

6. The processor then shifts to the Operational state, and broadcasts and delivers messages for the new ring.

To implement the first step of the extended virtual synchrony algorithm, each processor that is a member of both an old ring and the new ring determines the lowest *my_aru* of any processor from its old ring that is also a member of the new ring. It then broadcasts on the new ring every message for the old ring that it has received and that has a sequence number greater than the lowest *my_aru*. This ensures that each processor receives as many messages from its old ring as possible.

Each such message is broadcast with the old ring identifier and an old ring sequence number, as well as with a new ring identifier and a new ring sequence number. The new ring sequence numbers are used to ensure that messages are received, and the old ring sequence numbers are used to order messages as messages of the old ring. Messages from an old ring are not delivered to the application by any processor that was not a member of the old ring.

Completion of this message exchange is determined by the recovery flag *flg* in the token, which is initialized to false by the representative of the new ring, and by a variable *my_install_seq* in each processor. A processor changes *flg* from false to true if it has more messages to retransmit when it releases the token. A processor changes *flg* from true to false if it set *flg* to true and now has no further messages to retransmit.

When it receives the token on two successive rotations with *flg* set to false, a processor sets *my_install_seq* to *seq*. When *my_aru* is at least equal to *my_install_seq*, the processor has received all of the messages of the old ring that have been broadcast on the new ring.

If any of the messages to be delivered in the interim configuration require safe delivery, then the processor must not deliver the first Configuration Change message until it knows that all processors on the ring have determined a value for *my_install_seq* and have received all messages with lower sequence numbers. This is guaranteed when it has seen the *aru* field at least equal to *my_install_seq* in two successive token rotations and the *flg* field set to false three times.

The processor then moves to the second step. It 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 that is not safe on the old ring. It then delivers the first of the two Configuration Change messages, which enumerates the processors in the interim 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 the interim configuration or that need to be delivered to satisfy safe delivery. The processor then delivers the second Configuration Change message, which enumerates the processors on the new ring, and shifts to the Operational state.

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

5.1.5 Token Loss and Processor Failure

We do not distinguish between processor failure and token loss because a failed processor cannot forward the token to the next processor on the ring and thus the consequence of processor failure is token loss. If the token reaches a processor that has determined that the token is lost, the processor consumes the token.

The simplest token loss event occurs in the Operational state. On expiration of the token loss timeout, a processor regards itself as a representative, representing only itself but retaining its existing ring id, and proceeds to the Gather state.

Token loss can also occur to a representative in the Gather or Commit states. If, in either of these states, the token of the representative's existing ring is lost, the representative continues the operation of the membership algorithm, retaining its existing ring identifier but representing only itself.

Loss of the Form token or of the regular token for the new ring can also occur in the Form or Recover states. In these states the old ring is no longer operational and the new ring has not yet been installed, and the processor returns to the Gather state. To ensure termination, the algorithm guarantees, by placing a processor in the set of failed processors, that the next membership on which consensus is reached in the Commit state is smaller than the previous membership. An attempt is made to exclude the processor most likely to have lost the token.

If the token is lost during the Recover state, some processors may have installed the new ring while others have not. During the subsequent recovery the processors that installed the new ring must regard that ring as their old ring. Processors that have not installed the new ring must regard the old ring as their old ring. Thus, each processor must preserve its old ring identifier until it is able to install a new ring.

When a processor delivers a message as safe in an interim configuration, it guarantees that every other member of the interim configuration will also deliver that message unless that other processor fails. If the token is lost in the Recover state, some processors may not install the new configuration and thus will not deliver messages in the interim configuration. These processors will proceed in due course to form another new ring with an associated interim configuration. There may be processors in the original interim configuration that are not in this subsequent interim configuration. Because of the safe delivery requirement, messages from those processors must still be delivered in this subsequent interim configuration.

Consequently, for safe delivery, if a processor has lost the token late in the Recover state, then it must retain messages that it would have delivered in the first interim configuration because some other processor may have already installed the new ring. The processor must ensure that when it is able to install a new ring such messages are delivered in the interim configuration between the old ring and the installed new ring. The proof of correctness of the membership algorithm has been omitted.

6 Flow Control

A basic characteristic of reliable broadcast and multicast protocols is that the rate of broadcasting messages cannot exceed the rate at which the slowest processor can receive messages. At faster rates of broadcasting, the input buffer of the slowest receiver will become full and messages will be lost. Retransmission of these messages will result in an increase in message traffic and a reduction in the useful transmission rate.

Because modern transmission media, such as FDDI, are capable of very high transmission rates and because many processors may contribute to the transmission rate, a broadcast protocol without an effective flow-control mechanism will easily overwhelm the slowest receiver. The flow-control mechanism should allow high transmission rates when all processors are able to process messages at that rate. However, to prevent buffers from overflowing and messages from being lost, the mechanism must dynamically restrict the flow of messages to the rate at which messages can be processed.

For point-to-point protocols, positive acknowledgment mechanisms such as the sliding-window mechanism have been refined to provide excellent flow control. However, positive acknowledgment protocols are unattractive when messages are broadcast because of the excessive number of acknowledgments. Rate-controlled protocols have attracted much attention recently, particularly for applications that require high transmission rates. The disadvantage of rate control for broadcast protocols is that the aggregate rate of all transmitters must be controlled. For applications with bursty communication patterns, the maximum transmission rate must be set to a low value to guarantee that all processors receive messages at or below that rate.

The response time of the flow-control mechanism is determined by the size of the message buffer in each processor. Even if all processing stops, a processor can continue to receive messages and thus other processors can continue to broadcast messages until that processor's buffer becomes full. To ensure that buffer overflow does not occur, during the interval from the time at which a processor's buffer becomes empty and the next flow-control message (token for the Totem protocol) transmitted by that processor, the other processors must not broadcast more messages than that processor's buffer can contain. Consequently, each processor must transmit flow-control information at least once during any sequence of messages that is sufficient to fill its buffer.

This requirement is met by the Totem protocol. As each processor receives the token, it must empty its input buffer before passing on the token. If a processor is unable to process messages at the rate at which they are broadcast, many messages may still be in its input buffer when it receives the token. This processor must not transmit the token until it has processed those messages and emptied its input buffer. Thus, the rate of broadcasting is reduced to the rate at which messages can be processed by the slowest processor.

Our flow-control algorithm depends on the *fcc* field of the token, which keeps a count of the number of messages broadcast by all processors on the previous rotation of the token, and on the *window_size*, the maximum number of messages that all processors are allowed to broadcast in any token rotation. The number of messages that a processor is allowed to broadcast in this token rotation is bounded by the difference $window\_size - fcc$ and by the maximum number of messages it is allowed to broadcast in any token rotation.

On receipt of the token, a processor first processes all messages in its input buffer and broadcasts messages in its output buffer, up to the maximum number

Figure 3: The number of ordered broadcast messages per second and the utilization of the Ethernet for various message sizes for a network of five Sun 4/IPC processor running the Totem protocol.

Figure 4: The mean token rotation time and the mean latency from generation of a message until it is delivered in agreed order by all processors on the ring as a function of load.

permitted by the flow control mechanism. The processor then sets *fcc* to *fcc* plus the number of messages it broadcast during this token visit minus the number of messages it broadcast during the previous token visit. The processor then transmits the token to the next processor on the ring.

Assuming that the window size is no larger than the buffer size for any of the processors, it is easy to demonstrate that this protocol guarantees that no input buffer can overflow. Experience with the Totem protocol and other broadcast and multicast protocols has shown that effective flow control, capable of preventing message loss due to buffer overflow even at high message rates, is essential to the attainment of high throughput. Existing broadcast protocols must be throttled at relatively low rates of broadcasting to avoid high rates of message loss, particularly when the traffic is bursty. The circulating token flow-control mechanism of the Totem protocol has been demonstrated to ensure negligible rates of message loss even for very high rates of broadcasting.

7 Implementation and Performance

The Totem single-ring total ordering and membership protocol has been implemented as 1500 lines of C code on a network of Sun 4/IPC workstations connected by an Ethernet. The implementation uses the standard UDP broadcast interface within the Unix operating system (SunOS 4.1.1). One UDP socket is used for all broadcast messages, and a separate UDP socket is used by each processor to receive the token from its predecessor on the ring. The version of the code currently operating detects token loss and reforms the ring. Failed processors are removed from the ring, new and restarted processors can join the ring, and rings can partition and remerge.

Measurements from this first implementation show excellent performance. Figure 3 shows the number of messages ordered per second for messages of various sizes. These measurements were made on a network of five Sun 4/IPC workstations with the *window_size* set to the maximum value for which message loss is negligible. Note that with 1000 byte messages, 830 messages were ordered per second. Reducing the message size to 600 bytes resulted in over 1000 ordered messages per second. The highest prior rates for asynchronous fault-tolerant ordered broadcast messages of which we are aware are about 250 to 300 messages per second for the Transis protocol.

A concern about token-passing protocols, such as Totem, is that the token-passing overhead reduces the data transmission rate available for messages. Figure 3 shows the useful utilization (excluding transmission of the token) of the Ethernet achieved by the Totem protocol. With large messages, a utilization of about 70% is achieved, which may be compared to the approximately 65% utilization that can be achieved by TCP transmitting messages point-to-point with the same equipment.

While Figure 3 depicts the maximum data transmission rates measured using Totem over the Ethernet, Figure 4 considers performance at lower, more typical loads with Poisson arrivals of messages and 1000 byte messages. It shows the token rotation time and the latency from generation of a message until it is delivered in agreed order by all processors on the ring. At low loads (e.g. 400 ordered messages per second which is much more than the maximum throughput for prior protocols), the latency achieved by Totem is under 10 milliseconds. Even at 50% useful utilization of the Ethernet (625 messages per second), the latency is still only about 13 milliseconds. Note that the latency is close to half the token rotation time except at very high loads where queueing delays dominate.

8 Conclusion

The single-ring total ordering protocol of Totem provides fault-tolerant agreed and safe delivery of messages within a broadcast domain. The use of a token circulating around a logical ring provides excellent flow control and permits substantially higher throughput than existing total ordering protocols. There is no need to provide a weaker message ordering service, such as partially ordered causal delivery, because totally ordered agreed delivery can be provided at no extra cost.

The membership algorithm of Totem handles processor failure and recovery and also network partitioning and remerging. The concept of virtual synchrony has been extended to ensure consistent actions by processors that fail and are repaired and in networks that partition and remerge.

Continuing work on Totem is exploiting the single-ring protocol to provide more general services. Agreed and safe delivery services will be provided to process groups, as will membership services, across rings interconnected by gateways. It remains to be investigated whether the exceptional performance of the Totem single-ring protocol can be sustained in wide-area networks.

Acknowledgments

This work was supported by NSF grant NCR-9016361. We acknowledge the contribution of Danny Dolev, Shlomo Kramer and Dalia Malki who, with Yair Amir, developed the membership protocol for Transis from which our membership protocol is derived.

References

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

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

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

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

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

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

[7] F. Cristian. Reaching agreement on processor group membership in synchronous distributed systems. *Distributed Computing*, 4(4):175–187, April 1987.

[8] F. Cristian, H. Aghili, R. Strong, and D. Dolev. Atomic broadcast: From simple message diffusion to byzantine agreement. In *Proceedings of the IEEE Symposium on Fault Tolerant Computing*, pages 200–206, Ann Arbor, Michigan, June 1985.

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

[10] H. Kopetz, G. Grünsteidl, and J. Reisinger. Fault-tolerant membership service in a synchronous distributed real-time system. In *Proceedings of the International Working Conference on Dependable Computing for Critical Applications*, pages 167–174, August 1989.

[11] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, pages 558–565, July 1978.

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

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

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

[15] L. E. Moser, P. M. Melliar-Smith, and V. Agrawala. Membership algorithms for asynchronous distributed systems. In *Proceedings of the IEEE 11th International Conference on Distributed Computing Systems*, pages 480–488, Arlington, TX, May 1991.

[16] L. E. Moser, P. M. Melliar-Smith, and V. Agrawala. Asynchronous fault-tolerant total ordering algorithms. To appear in *SIAM Journal of Computing*.

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

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