Broadcast Protocols for Distributed Systems

P. M. MELLIAR-SMITH, MEMBER, IEEE, LOUISE E. MOSER, MEMBER, IEEE, AND VIVEK AGRAWALA

Abstract—**We present an innovative approach to the design of fault-tolerant distributed systems that avoids the several rounds of message exchange required by current protocols for consensus agreement. The approach is based on broadcast communication over a local area network, such as an Ethernet or a token ring, and on two novel protocols, the Trans protocol, which provides efficient reliable broadcast communication, and the Total protocol, which with high probability promptly places a total order on messages and achieves distributed agreement even in the presence of fail-stop, omission, timing, and communication faults. Reliable distributed operations such as locking, update and commitment, typically require only a single broadcast message rather than the several tens of messages required by current algorithms.**

Index Terms—**Agreement problem, broadcast communication, communication protocols, distributed systems, fault tolerance, local area networks, total order.**

I. Introduction

MANY important activities in a distributed system involve simultaneous coordination of several processors. Among these are scheduling and load balancing, synchronization, process migration, remote procedure calls, nested atomic transactions, access to distributed information, locking, update and commitment, and transaction logging. All of these activities require agreement among processors as to which processor should undertake, or has undertaken, an action.

Unfortunately, significant problems exist in the design of algorithms for reaching asynchronous distributed agreement when processors can fail. Aside from some strong impossibility results [13], [14], [21], existing algorithms are very expensive, requiring for a group of five processors, 40 messages, and perhaps 40 acknowledgments to reach agreement with no failures and more messages in the presence of processor failures or communication errors [23]. Thus, all of the activities above, activities that are essential to distributed systems and distributed applications, are rather expensive.

We present a novel efficient approach to asynchronous distributed agreement that is based on broadcast communication. The basic strategy consists of

• an efficient broadcast (or multicast) protocol, the Trans protocol, which ensures that every message broadcast or received by any working processor is received by every working processor, and

• an efficient protocol, the Total protocol, which with high probability promptly places a total order on broadcast messages, ensuring that even in the presence of faults all working processors agree on exactly the same sequence of broadcast messages.

Manuscript received April 21, 1989; revised August 29, 1989. This work was supported in part by the National Science Foundation Grant CCR89-08515.

The authors are with the Department of Electrical and Computer Engineering and the Department of Computer Science, University of California, Santa Barbara, CA 93106.

IEEE Log Number 8931912.

It is easy to demonstrate that placing a total order on broadcast messages, so that every working processor processes the same messages in the same order, provides an immediate solution to the agreement problem. Once this total order is determined, distributed actions can be carried out using simple sequential fault-tolerant algorithms. The strategy is very efficient: for example, locking records in a distributed database typically requires only a single broadcast message to claim a lock and a single broadcast message to release it. Based on this strategy, it is possible to design simple and efficient but very robust distributed systems.

A. Existing Agreement Protocols

The first areas of computer science to directly address the problems of reaching agreement in a fault-tolerant system were those of distributed databases [1] and remote procedure calls [4]. In neither case were good solutions immediately forthcoming and it soon became apparent that the general problem of reaching agreement in a system subject to faults underlay many of the difficulties encountered. Subsequently, it was shown that the problem of reaching agreement is not merely hard but actually impossible in an asynchronous system [14]. Asymptotic protocols were devised that reach agreement with high probability but with correspondingly high costs [5], [6], [23].

The most detailed existing description of a reliable atomic broadcast protocol is that of Chang and Maxemchuk [7], [8]. All messages pass through an intermediary node, called the token site; an elegant token-passing protocol is used to detect failures at the token site, to select a new token site, and to retransmit messages affected by the failure. Typically, about three messages are required for each broadcast message, and the latency is reasonable at low loads but increases at high loads. As with almost all of the other protocols described here, the need to recognize a failed processor, and to reconfigure the system to exclude it, introduces long delays when a processor fails.

Kopetz [17], [18] developed and implemented a practical atomic multicast protocol for real-time systems. His Mars system is fully synchronous and uses a TDMA broadcast medium with simple algorithms and low overhead. Failure of a processor must be detected but introduces no delay to other messages. The design is very suitable for real-time systems, but a fully synchronous approach is rather inflexible for transaction processing and other asynchronous applications.

The HAS system of Cristian [11] is based on fabricating an atomic broadcast from unreliable message communication. To avoid the impossibility results associated with fully asynchronous systems, his system uses loosely synchronized clocks and timeouts with an upper bound on message transit time. More flexible is the V system [9] which employs broadcasting but makes no guarantee of delivery or recovery. Higher efficiency is obtained at the cost of passing much of the work of recovery on to the application program. Cheriton [10] also investigated multicast protocols, accepting the inevitability of multiple acknowledgments but demonstrating that careful optimization of the protocol can reduce the costs to a more acceptable level.

Luan and Gligor [19] devised an atomic broadcast protocol based on a variation of three-phase commit that uses voting to avoid blocking. Their algorithm requires over $4n$ messages per atomic broadcast in a system of size n, but under high load conditions many messages can be ordered per execution of the protocol so that the overhead can become quite low. The latency of the algorithm remains high however. The algorithm can operate without explicitly detecting the failure of processors. Garcia-Molina and Spauster [15] have also devised algorithms for atomic multicast based on point-to-point communication over a spanning tree.

Quite close in concept to our approach is the ISIS system of Birman and Joseph [2], [3], which is based on the idea of broadcasting and placing a total order on broadcast messages. ISIS differs, however, in that its developers did not have an efficient broadcast acknowledgment protocol, such as the Trans protocol, available to them, and their total order protocol, due to Skeen is less efficient than the Total protocol. To recover reasonable efficiency, ISIS employs the ingenious ideal of virtual synchronous application programs, but the overhead costs are still high.

Peterson, Buchholz, and Schlichting [24] devised the Psync protocol which also uses an approach similar to ours, constructing a partial order which is then converted into a total order. However, their algorithms are weaker than Trans and Total, requiring the system to be partially synchronous and to block until a failed processor is detected and removed from the configuration.

B. Context of the Broadcast Protocols

Our broadcast communication model is intended to match typical local area networks, such as the Ethernet or the token ring. Processors are selected to broadcast at random from among the processsors seeking to use the communication medium. A processor's broadcast message is received immediately or not at all by the other processors. Broadcast messages are assumed to satisfy the requirements of the Trans and Total protocols described below.

A reception fault occurs when a processor fails to receive a broadcast message. Reception faults are caused relatively infrequently by the physical communication mechanisms and rather more frequently by exceeding the processing and message buffering capacity of the processor. The choice of processors at which a reception fault occurs is assumed to be random rather than malicious. Network partitioning faults are accommodated, and the protocols continue to operate uninterrupted in a component of the partition with at least $2n/3$ processors.

The model also assumes that processors are subject to fail-stop, omission, and timing faults but not to malicious faults. A processor that has failed makes no further broadcasts, while a working processor continues to broadcast, although not necessarily within any fixed time constraint. In a partitioned system, the processors in a component of the partition appear to have failed to processors in the other components.

Operating systems, particularly Unix, are prone to pauses of a few seconds during which little happens even though the processor has not failed and normal processing will resume. Protocols that are required to detect processor failures in order to make decisions must accept occasional pauses during which the whole system stops briefly or, alternatively, abort processors and incur frequently the overhead of determining a new configuration.

The Trans and Total protocols do not attempt to detect processor failures because, as data link layer protocols, they must make decisions very quickly, typically within a few milliseconds or tens of milliseconds. Rather, as fault-tolerant protocols, they determine a total order promptly with high probability even in the presence of failed processors, recovering automatically from transient processor failures and transient network partitioning. However, for effective implementation, detection of failed processors by a higher-level protocol of the protocol hierarchy is required. For example, the Trans protocol retains copies of messages until they have been received and acknowledged by all processors in the configuration. Consequently, the protocol must be informed that a processor has failed and has been removed from the configuration so that message buffer space can be recovered.

The design of the Trans and Total protocols assumes that the communication interface will include an interface processor and buffer space sufficient to receive, buffer, process, and acknowledge every message delivered by the communication medium. Much of the processing costs of these protocols will be carried by the communication controller rather than by the main processor, and only messages intended for the main processor need be delivered to it even though the communication controller processes every message. Although simple Ethernet controllers do not have this capability, controllers that accept and process every message can readily be designed. Bearing in mind the importance of communication in high-performance distributed systems, the expense of a capable communications interface processor, while not negligible, should be compared to the costs of sophisticated display and disk controllers in modern computers.

II. The Trans Broadcast Protocol

Many distributed computer systems use a communication mechanism that is physically a broadcast medium, such as an Ethernet, token ring, 1553 bus, or packet radio system. The advantage of a broadcast communication medium is that it makes distribution of a message simultaneously to several destinations physically possible. Existing standard communication protocols do not allow distributed systems to make use of the broadcast capability of the physical communication medium. Rather, existing protocols require all messages to be point-to-point from a single source to a single destination.

The Trans broadcast protocol [20] uses a combination of positive and negative acknowledgment strategies to achieve efficient reliable broadcast or multicast communication. Messages can be broadcast simultaneously to many destinations without the need for explicit acknowledgment by every recipient. An Observable Predicate for Delivery determines which processors have received a message, even though they did not acknowledge it directly.

Trans provides reliable communication despite a noisy communication medium and processor fail-stop, omission, or timing faults. Multicast, rather than fully broadcast, communication is readily achieved by operating several subnets over the same local area network, a standard feature provided by existing protocols. Alternatively, a destination address list may be used to denote the processors for which the message is intended; other processors will typically receive and possibly acknowledge the message but will otherwise ignore it.

Within the ISO protocol hierarchy, the primary responsibility for ensuring reliable transmission across the broadcast medium lies with the data link layer [12]. The Trans protocol is directed towards that layer of the hierarchy and provides services appropriate to that layer only. While Trans can determine whether a processor has acknowledged receipt of a message, it relies on a higher-level protocol to determine network membership following a failure.

By the performance measures of number of messages and utilization of the communication medium, the Trans protocol is clearly better than typical point-to-point protocols whenever the application requires broadcasting or multicasting. The most significant performance advantages of Trans result, however, from its use in conjunction with the Total protocol to achieve agreement in fault-tolerant distributed systems.

A. The Protocol

The basic idea behind the Trans protocol is that acknowledgments for broadcast messages are piggybacked on messages that are themselves broadcast and typically seen by all other processors. The operation of Trans is illustrated in the following scenario:

—Processor P broadcasts a message.

—The message from processor P is received uncorrupted by processor Q.

—Processor Q includes a positive acknowledgment for P's message in Q's next message.

—Processor R on receiving Q's message is aware that P's message has been acknowledged and that there is no need for R also to acknowledge it in its next message; instead processor R acknowledges Q's message.

—If processor R has not received the message from P, the message from Q alerts R of this loss and, therefore, R includes a negative acknowledgment for P's message in R's next message.

We now give a property-theoretic definition of the Trans protocol. We wish to define just those properties of the protocol that are necessary for correct operation and to avoid confusing them with the details of one specific implementation that is in no way preferable to any other. Thus, we define the data link layer protocol Trans by means of the following requirements:

Message Format

- Each message is broadcast with a header in which there is a message identifier containing the identity of the broadcasting processor and a message sequence number. Other fields of the header, such as a message destination address list for multicasting, may be present but do not play a part in the protocol.
- Retransmissions are identical to the original transmission. The retransmitted message thus contains exactly the same acknowledgments, positive and negative, as the prior transmission of the message. Note that the retransmission can be broadcast by any processor, not just by the processor that originated the message.
- To avoid large delays in a lightly loaded system, if a processor has no messages pending, it may construct a null message to carry acknowledgments. The acceptable delay before transmitting a null message may differ for positive and negative acknowledgments.

Data Structures

Each processor maintains

- An *acknowledgment list* of message identifiers with positive and negative acknowledgments. The acknowledgments in this list are transmitted with the next message the processor broadcasts.
- A *received list* of messages the processor has received uncorrupted, or has broadcast in the recent past, and may need to rebroadcast. Messages are retained in this list until there is no possibility of retransmission being required.
- A *pending retransmissions list* of message identifiers. The processor received each of these messages and also a negative acknowledgment for each of them. These messages will be retransmitted by the processor unless it observes that they have already been broadcast by some other processor.

Sending a Message

- When a processor prepares a message for broadcast, it appends its acknowledgment list to the message. Positive acknowledgments that are appended to the message are removed from the acknowledgment list, but negative acknowledgments are retained. If there are too many acknowledgments to append to the current message, negative acknowledgments are given priority over positive acknowledgments.

Receiving a Message

When a processor receives a message,

- It adds the message identifier with a positive acknowledgment to its acknowledgment list and adds the message to its received list. If the message identifier is present with a negative acknowledgment in the acknowledgment list, it deletes the identifier from that list. If the message is present in the retransmissions list, it deletes the message from that list as well.
- If a positive acknowledgment is appended to the message, the processor deletes from its acknowledgment list any

matching positive acknowledgment. If the acknowledgment is for a message that is not in its received list (i.e., for a message that it has not received), it adds the identifier for that message with a negative acknowledgment to its acknowledgment list.

- If a negative acknowledgment is appended to the message then, if the acknowledged message is already on its received list, the processor adds the message identifier to its retransmissions list; otherwise, it adds the message identifier to its acknowledgment list with a negative acknowledgment unless the identifier is already present.
- A processor can also recognize that it has not received a message when it receives a message with a sequence number more than one greater than the largest sequence number of a message in its acknowledgment list from the same source. Again, one or more identifiers with negative acknowledgments are added to its acknowledgment list. If there is a large discontinuity in sequence numbers, it may be preferable not to attempt to recover the missing messages at the data link level, but rather to refer the problem to the network level of the protocol hierarchy.

Retransmission Timeout

- If a processor has not received a positive acknowledgment for a message it broadcast within some time interval, it adds the message to its retransmissions list.

Pruning the Received List

- A broadcast message with the appended acknowledgments is retained in the received list until the processor has determined (using the Observable Predicate for Delivery) that all of the processors in the configuration have received that message.

The protocol described above is reliable against momentary transmission failures. It can operate over networks that are connected but not completely connected and even where the interconnection topology changes dynamically. However, under such circumstances, the efficiency of the protocol is adversely affected.

B. Examples

As an example of the operation of Trans, consider the following message sequence in which upper-case letters represent messages (we do not bother to denote the source of the message directly), lower-case letters represent acknowledgments, and overhead bars denote negative acknowledgments.

$$A \quad Ba \quad Cb \quad Dc \quad E\bar{c}d \quad Cb \quad Fec$$

Here message A is only acknowledged by message B. Other processors that are aware of the presence of B's acknowledgment do not acknowledge A in their subsequent messages. It is this feature that reduces the number of acknowledgments required. Note that the positive acknowledgment of C that accompanies message D alerts a processor that it did not receive message C and, thus, causes the negative acknowledgment of C that accompanies message E. This negative acknowledgment triggers a retransmission of C with precisely the acknowledgments that accompanied the original transmission; thus, the retransmission cannot acknowledge message E that caused the retransmission. The processor broadcasting message F also acknowledges message E in addition to message C; in doing so, it implicitly acknowledges messages B and D and, through B, message A as well. Thus, each message will contain typically only a few acknowledgments but will implicitly acknowledge many other messages through the transitivity of positive acknowledgments.

The effect of missing several messages can be seen in this next example.

$$A \quad Ba \quad Cb \quad Dc \quad E\bar{c}d \quad Cb \quad F\bar{b}ec \quad Ba \quad Gfb$$

Here the processor broadcasting message E received neither message C nor B, but is informed by message D only of the loss of C. When C is retransmitted with a positive acknowledgment for B, the processor becomes aware that it missed B too and transmits a further negative acknowledgment with message F. Thus, a short sequence of missing messages can be recovered quite quickly and easily; of course, this technique is inappropriate for recovery from an extended processor failure.

The simple linear sequence of acknowledgments shown above may be rather optimistic. Checking the cyclic redundancy code, manipulating the acknowledgment queues, and constructing message packets all take time, but efficient use of the communication medium requires that the next message be transmitted with as little delay as possible. Thus, the idealized expectation that reception of a message will be reflected in the acknowledgments that accompany the next message is unrealistic and is not required by the Trans protocol. Delays in broadcasting acknowledgments and the broadcasting of extra acknowledgments, either positive or negative, have no logical effect on the protocol and only a small effect on performance, as shown in the next example which assumes that no message is acknowledged by the next broadcast message.

$$A \quad B \quad Ca \quad Dab \quad Ebc \quad Fcd \quad G\bar{c}de \quad Hef \quad Ca \quad Igh \quad Jghc$$

Here, because messages are not processed instantaneously, each message is acknowledged by two subsequent messages. Note that the negative acknowledgment mechanism is still effective in provoking the necessary retransmission.

C. The Partial Order

Trans is a very robust protocol that is resilient to most forms of failure, with the exception of Byzantine failures and complete failure of the communication medium. It is easy to prove an Eventual Delivery Property, which states that for any message, eventually, if any working processor has broadcast or received the message, then all working processors have received it [20].

The positive and negative acknowledgments contained in messages permit processors to determine whether a processor has received a message even though the processor did not acknowledge the message directly. We define an Observable Predicate for Delivery, denoted by OPD(P,A,C), which states that processor P can be certain that the processor that broadcast message C has received and acknowledged, directly

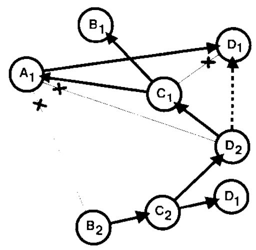


Fig. 1. The graphical representation of the positive acknowledgments represented by arrows and negative acknowledgments represented by lines marked with X's in a sequence of broadcasts by four processors.

or indirectly, message A at the time of broadcasting message C. The predicate is true if and only if processor P receives a sequence of messages, not necessarily consecutive broadcast messages, such that

- The sequence commences with message A and ends with message C.
- Every message of the sequence, other than A, positively acknowledges its predecessor in the sequence or is broadcast by the processor that broadcast its predecessor in the sequence.
- No message in the sequence is negatively acknowledged by message C.

By enumeration, the Observable Predicate for Delivery can be used to determine that a message has been received by all processors in the configuration and can therefore be deleted from the data structures maintained by the Trans protocol. The Observable Predicate for Delivery enables the processors to construct a partial order relation on the broadcast messages.

> *The Partial Order:* In the partial order constructed by processor P, message C *follows* message B if and only if OPD(P, B, C) and, for all messages A, OPD(P, A, B) implies OPD(P, A, C).

The partial order satisfies an important property, the Prior Reception Property, which states that if message C is included in the partial order, then at the time processor P_C broadcast message C, it had received and acknowledged, directly or indirectly, all messages that precede C in the partial order. Note that OPD(P, B, C) may remain undefined indefinitely if processor P_C fails. In such a case, message C can never be included in the partial order.

As an example of the construction of the partial order, consider the following sequence of messages broadcast by four processors where A_1 is the first message broadcast by processor P_A, etc.

$$B_1 \quad D_1 \quad A_1 d_1 \quad C_1 \bar{d}_1 b_1 a_1 \quad D_2 \bar{a}_1 c_1 \quad D_1 \quad C_2 d_2 d_1 \quad B_2 \bar{a}_1 c_2$$

Fig. 1 graphically represents the positive and negative acknowledgments resulting from this sequence of messages. The heavy arrows represent acknowledgments while the lighter lines marked with X's represent negative acknowledgments. The acknowledgment by message D_2 for its predecessor D_1 is implicit rather than explicit.

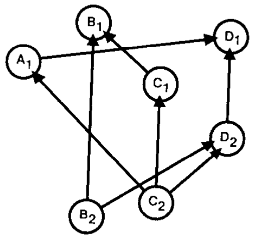


Fig. 2. The partial order derived from the acknowledgments shown in Fig. 1. For example, message C_1 does not follow message A_1 because A_1 follows D_1 and processor P_C had not received D_1 when broadcasting C_1.

Fig. 2 shows the partial order constructed from these acknowledgments. Message C_1 does not follow message A_1 because A_1 follows D_1 and processor P_C had not received D_1 when broadcasting C_1. Rather, C_2 follows A_1 because P_C had received the retransmission of D_1 by the time that it broadcast C_2. Similarly, B_2 does not follow C_2 because C_2 follows A_1 which processor P_B had not received at the time it broadcast B_2. Instead, B_2 follows D_2 since P_B received D_2 and all the messages P_D had received at that time.

It is relatively easy to show using the Eventual Delivery Property that all working processors construct the same partial order and that the failure of a processor may result in some of its messages being excluded from the partial order [20]. In case that the network partitions, the working processors in the same component of the partition construct the same partial order. This partial order, constructed from the acknowledgments of the Trans protocol and satisfying the Eventual Delivery Property and the Prior Reception Property, is the base upon which the Total protocol is built.

III. The Total Protocol

The objective of the Total protocol is to reach distributed fault-tolerant agreement by placing a total order on messages and by ensuring that all working processors determine the same total order. The Total protocol is based on the partial order relation derived from the acknowledgments of messages by the Trans protocol. There is only one partial order, which must be the same for all processors, but some processors may be aware of only part of the partial order because they have not yet received all of the messages that have been broadcast. Typically, the partial orders of the various processors differ only in the more recent messages.

If the system were completely reliable, the partial order would be a total order. Unfortunately, it is possible for one message to be received by a subset of processors and another message to be received by a disjoint subset, thus providing no information by which to order them. There is also a risk that a processor has failed and will never be heard from again; thus, it must be possible to make decisions in the absence of messages from some of the processsors. Moreover, one or more processors may be unable to broadcast a message for some period of time, even though they have not failed, because of contention for the bus or other internal activities.

Even with broadcast communication, the acknowledgments of messages can yield an arbitrary partial order. The importance of broadcast communication is that such bad cases are rare; broadcast communication almost always yields a partial order that can quickly be converted into a total order.

The Total protocol is a fault-tolerant algorithm for converting a partial order into a total order, whose probability of determining an extension to the total order asymptotically approaches unity as more messages are broadcast. The algorithm is resilient to fewer than $n/3$ faults where n is the number of processors in the system. We have also developed a more complex but slightly slower algorithm for determining a total order that is resilient to fewer than $n/2$ faults [22].

A. The Protocol

The Total protocol needs no additional broadcast messages beyond those required by the Trans protocol. However, determination of the total order does not occur immediately after a message is broadcast but must wait for reception of broadcasts by other processors. The protocol incrementally extends the total order by selecting messages from those in the partial order but not yet in the total order.

A message that does not follow in the partial order any other message aside from those already in the total order is a *candidate* message. Clearly, there must be at least one candidate message, and there can be at most a single candidate message from each source. The total order is extended by a decision to include a set of candidate messages in the total order. Each such candidate set is voted on separately. The vote of a message is determined by the votes of the messages that precede it in the partial order, and the decision of a processor is determined by the votes of the messages in its partial order, not on the decisions of other processors.

Voting on a candidate set takes place in a sequence of stages; different candidate sets have different sequences of stages. A message votes on a candidate set in a stage only if no previous message from its source has already voted on the candidate set in that stage. In stage 0, the vote of a message on a candidate set depends on whether or not that message follows in the partial order other candidate messages. In stage i, where $i > 0$, a message votes on a candidate set if it follows in the partial order enough messages that voted in stage $i-1$. The number of votes required for a decision and for a further vote must be at least N_d and N_v, the parameters to the algorithm.

Resilience	N_d	N_v
1	$\frac{n+2}{2}$	$\frac{n-1}{2}$
2	$\frac{n+3}{2}$	$\frac{n-2}{2}$
$k<\frac{n}{3}$	$\frac{n+k+1}{2}$	$\frac{n-k}{2}$.

The Total protocol is defined, completely rigorously, by the following voting and decision criteria; these criteria determine which candidate set is chosen for inclusion in the total order.

The Voting Criteria

In stage 0

• A message *votes for* a candidate set if that message follows in the partial order every message in the candidate set and it follow no other candidate message. (A candidate message votes for the set containing only itself.)

• A message *votes against* a candidate set if that message follows in the partial order any candidate message other than those in the candidate set. (A candidate message votes against all sets of which it is not a member.)

In stage i where $i > 0$

• A message *votes for* a candidate set if

—the number of messages that if follows in the partial order and that voted for the candidate set in stage $i - 1$ is at least N_v, and

—it follows in the partial order fewer message that voted against the candidate set than voted for the set in stage $i - 1$.

• A message *votes against* a candidate set if

—the number of messages that it follows in the partial order and that voted against the candidate set in stage $i - 1$ is at least N_v, and

—it does not vote for the candidate set in stage i.

The Decision Criteria

In stage i where $i > 0$

• A processor decides for a candidate set if

—the number of messages in its partial order that voted for the candidate set in stage i is at least N_d, and

—for each proper subset of the candidate set, the processor had decided against that proper subset.

• A processor decides against a candidate set if

—the number of messages in its partial order that voted against the candidate set in stage i is at least N_d.

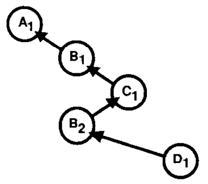


Fig. 3. A simple example for six processors in which every broadcast message is received by every processor. There is only one candidate set $\{A_1\}$, and the decision to extend the total order to include message A_1 can be made as soon as message D_1 is received.

Once a decision has been made in favor of a candidate set, the messages of that set are included in the total order in any arbitrary but deterministic order, and the whole process is repeated. Since each message follows itself in the partial order, a message can include its own vote in stage $i-1$ towards the totals required to vote in stage i. The votes and decisions need not be included in the messages themselves but can be deduced from the acknowledgments in the messages.

A processor can always determine the vote of a message in its partial order since the message would not have been placed in the partial order if any message that precedes it had not been received. The Trans protocol guarantees that if any working processor places a message in the partial order then eventually every working processor does.

B. Examples

First consider a one-resilient system of six processors that requires at least four votes for a decision and three votes for a further vote. Fig. 3 shows a very simple situation that might result when every broadcast message is received by every

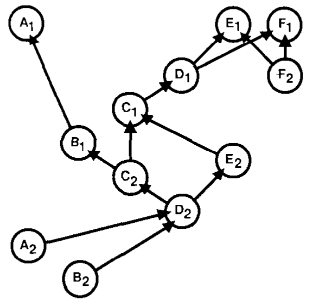


Fig. 4. A more complex example in which messages are not received by all processors. Here the candidate sets $\{A_1\}$, $\{E_1\}$ and $\{F_1\}$ obtain too many negative votes in stage 0 and, thus, are decided against, but the set $\{E_1, F_1\}$ obtains four favorable votes in stage 0 from D_1, C_1, E_2, and F_2, enough for a favorable decision. Even if message F_2 is lost, there remain three favorable votes in stage 0, but there are four favorable votes in stage 1 from E_2, D_2, A_2, and B_2, again enough for a favorable decision.

processor. There is only one candidate message A_1, and the messages A_1, B_1, C_1, and D_1 are sufficient for a decision. Thus, every processor on receiving message D_1 will decide to extend the total order to include message A_1. To make this decision there is no need to know what the other two processors in the system are doing.

A more difficult situation is shown in Fig. 4, where messages are not received by all processors. There are three candidate messages A_1, E_1, and F_1. The candidate sets $\{E_1\}$ and $\{F_1\}$ are voted for only by the messages themselves. Messages A_1 and B_1 vote for the candidate set $\{A_1\}$, but messages C_2 and D_2 do not because they follow other candidate messages in the partial order. Messages D_1, C_1, E_2, and F_2 vote for the candidate set $\{E_1, F_1\}$, a sufficient number of votes for a decision.

Note that the candidates in the set $\{A_1, E_1, F_1\}$ precede the four messages C_2, D_2, A_2, and B_2. Thus, no processor can decide for the set $\{A_1, E_1, F_1\}$ without first deciding against the set $\{E_1, F_1\}$.

We must also consider the possibility that processors may fail at inconvenient moments. Suppose that processor P_F fails some time after broadcasting message F_1 and before broadcasting F_2. The other processors do not know whether P_F had received messages E_1, D_1, C_1, and E_2 and, thus, had decided for the set $\{E_1, F_1\}$. Nor can they be confident that P_F had indeed failed; P_F may be trying to broadcast but may be blocked by contention for the bus, or it may be working on an urgent task, or it may be taking a short siesta from which it will awake to announce that is has indeed received those messages and decided for $\{E_1, F_1\}$, or against, as the case may be.

However, even without knowledge of processor P_F's vote, three messages D_1, C_1, and E_2 vote for the set $\{E_1, F_1\}$ in stage 0, and four messages E_2, D_2, A_2, and B_2 follow those three messages and, therefore, vote for $\{E_1, F_1\}$ in stage 1. Consequently, messages E_2, D_2, A_2, and B_2 suffice for the decision to include the set $\{E_1, F_1\}$ in the total order.

C. Validity

The validity of the algorithm depends on showing that for each extension of the total order

- If a processor decides for (against) a candidate set, then no processor decides against (for) that set.
- If a processor decides in favor of a candidate set, then no processor decides in favor of a different set.
- If a processor decides in favor of a candidate set in stage i, then each working processor decides in favor of that set in stage h where $h \leq i + 1$.
- If a processor includes a particular candidate set at its jth extension of the total order, then every working processor includes that set at its jth extension of the total order. Consequently, the total orders determined by all working processors are identical.
- If a working processor broadcasts a message that follows each message in a candidate set S, then in each stage each working processor broadcasts a message that votes on S.
- A processor cannot decide against the candidate set consisting of all candidate messages in its partial order.
- If a message m' follows a message m in the total order, then m does not follow m' in the partial order. Thus, the total order is consistent with the partial order.

Each of these properties has been proved for the $n/3$ protocol and also for the $n/2$ protocol [22]. We can also demonstrate that, given reasonable behavior by the broadcast communication system, the probability of all processors remaining undecided diminishes quite quickly to zero.

D. Performance Model

At first sight the protocols may appear to be somewhat complex and, thus, likely to be slow and expensive. However, if the local area network has reasonable reliability, then almost every broadcast message is received by every processor. Under these very probable conditions, the broadcast protocols excel.

To simplify our performance model, we assume optimistically that all processors are equally likely to broadcast at every time, that every message broadcast is received by every processors, and that every message acknowledges the message broadcast immediately before it. Thus, there are no negative acknowledgments and no retransmissions. Consequently, for each extension of the total order, there is only one candidate message and, once sufficient messages have been broadcast by distinct processors, every processor will decide to include that message in the total order.

For example, in a one-resilient system with $n = 10$ processors, the minimum number of messages required is $\lceil (n+2)/2 \rceil = 6$. A message can be included in the total order once five further messages from five different processors have been received. Of course, we cannot assume that the next five messages will be from different processors, but we can compute the probability of receiving messages from five different processors. This is related to a well-understood problem, the "urn problem" [16]. The derivation of the performance models is too complex to present here; consequently, we display only a few samples of our performance results.

Fig. 5 shows the probability of incurring delays between receipt of a message and its inclusion in the total order for various configurations. Such delays are often referred to as the

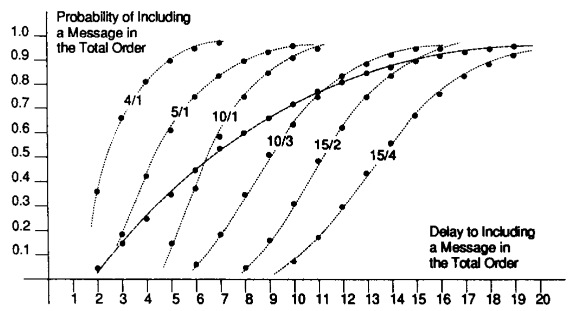


Fig. 5. Probability of incurring delays between receipt of a message and its inclusion in the total order. The horizontal axis represents the delay in message transmissions. The curves are labeled with the number of processors in the system and the resiliency of the system. The unlabeled dashed curve represents a four-processor one-resilient subsystem within a ten-processor system.

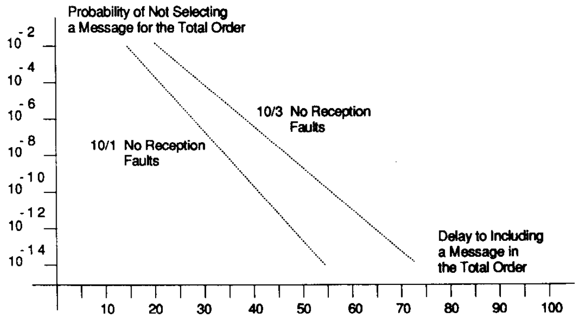


Fig. 6. The probability of not deciding on a candidate set to include in the total order diminishes rapidly as more messages are broadcast.

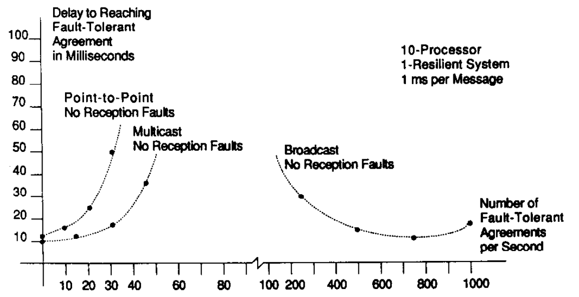


Fig. 7. The delay to reaching fault-tolerant agreement as a function of the load on the system. A ten-processor one-resilient system is assumed. The point-to-point and multicast algorithms use a three-processor subsystem to reach agreement.

latency of the protocol. Examining the graph for the ten-processor one-resilient case, we note that there is a 0.15 probability that the message can be placed in the total order after five additional messages (i.e., the next five messages all came from different processors), a 0.38 probability that six messages suffice (two messages came from the same processor), and a 0.59 probability that seven messages suffice. The expected number of additional messages required before a message can be placed in the total order is 7.5.

Smaller systems are able to place messages in the total order after less delay than larger systems; for a four-processor one-resilient system the expected number of additional messages is only 3.3. Since the four-processor one-resilient case performs so well, it might be thought that, even when more processors are available, processors should be grouped in fours with the algorithm applied only to messages within a group, ignoring other messages. Fig. 5 shows the probability of delay for a four-processor one-resilient subgroup of a ten-processor system. Although the smaller subgroup can sometimes decide on the total order very quickly, more often it is delayed while messages from other processors are broadcast. Overall, the four-processor subgroup performs worse then the full ten-processor one-resilient system, the expected delay being increased from 7.5 to 8.3 messages with a large increase in the variance. Thus, broadcasting is more effective than multicasting in establishing the total order.

Even if the mean delay is acceptable, we must also consider the possibility of occasionally incurring a very long delay before a message can be placed in the total order. Fig. 6 shows the probability of not deciding on a candidate set as the number of broadcast messages increases. It can be seen that for a ten-processor one-resilient system the probability of remaining undecided diminishes by a factor of 10^{-3} with every ten messages and that by the time 50 messages are broadcast the probability is indeed truly negligible.

We now compare the performance of Trans and Total, again for a ten-processor one-resilient system and for a message transmission time of 1 ms, against the best existing algorithms for point-to-point and multicast communication [23], which run on a three-processor subsystem. Fig. 7 shows the expected delay from the moment at which a processor seeks the use of the bus to broadcast a request for a fault-tolerant agreement until it receives the resulting agreement. Note the change of scale on the horizontal axis of this graph. As the load increases, the broadcast protocols show improved performance because the required six messages from distinct sources can be obtained sooner with higher traffic. The small increase in delay at very high traffic rates is caused by waiting to obtain access to the bus. Optimum use of these protocols requires that processors without messages to broadcast should periodically broadcast null messages.

With no reception faults, the Trans and Total protocols are capable of more than 700 fault-tolerant agreements per second with very low delay. In contrast, the point-to-point and multicast protocols exhibit acceptable performance only at low agreement rates, deteriorating rapidly at more than 30 agreements per second. The performance advantages of Trans and Total are evident. Agreement rates of 100 or more per second are typical in current high-performance transaction processing systems. While it is possible to reduce the number of fault-tolerant agreements required in distributed systems, a price is paid in design complexity and in risk of rollback.

The computational costs of the Trans and Total protocols must also be considered. In the worst case the computational costs are infinite, but the overall mean computational cost is very close to the best case cost in which all messages have been received by every processor, there is only one candidate message, and the decision can be made in stage 0. We are currently investigating the computational costs and devising efficient implementation algorithms for the protocols. Certain modifications to the protocols, such as acknowledging messages from a source only in sequence number order, permit substantially simpler and more efficient implementations.

IV. Conclusion

The Trans and Total protocols are in the early stages of their development, but already it is clear that broadcast communication can provide large performance improvements for distributed fault-tolerant systems when appropriate protocols are used. The use of broadcast communication will make it feasible to develop high-performance transaction processing systems using fault-tolerant distributed architectures rather than the centralized architectures that are currently used.

Imposing a consensus total order on broadcast messages eliminates one of the traditional problems in the design of distributed systems, the lack of a global system state. Without a global system state, complex reasoning is necessary about what information is known to each processor. The agreed total order on broadcast messages imposes a common system history and, thus, a common system state with each processor's maintaining as much of the system state as is necessary for its functioning. Consequently, distributed systems need be no more difficult to design than asynchronous centralized systems.

The protocols also demonstrate that agreement in a distributed fault-tolerant system is not inherently expensive using existing local area networks. In an n-processor one-resilient system, the Trans and Total protocols require, under favorable and quite probable conditions, only one broadcast message per agreement, and they reach that agreement after only $\lceil (n+2)/2 \rceil$ broadcast messages from distinct processors. These numbers of broadcast messages approximate the minimum possible.

References

[1] P. Bernstein and N. Goodman, "The failure and recovery problems for replicated databases," in *Proc. ACM Symp. Prin. Distribut. Computing,* 1983, pp. 114-122.

[2] K. P. Birman and T. A. Joseph, "Reliable communication in the presence of failures," *ACM Trans. Comput. Syst.,* vol. 5, no. 1, pp. 47-76, Feb. 1987.

[3] ——, "Exploiting virtual synchrony in distributed systems," in *Proc. ACM Symp. Operat. Syst. Prin.,* 1987, pp. 123-138.

[4] A. Birrell and B. Nelson, "Implementing remote procedure calls," *ACM Trans. Comput. Syst.,* vol. 2, no. 1, pp. 39-59, Feb. 1984.

[5] G. Bracha, "Asynchronous Byzantine agreement protocols," *Inform. Computat.,* vol. 75, pp. 130-143, Nov. 1987.

[6] G. Bracha and S. Toueg, "Asynchronous consensus and broadcast protocols," *JACM,* vol. 32, no. 4, pp. 824-840, Oct. 1985.

[7] J. Chang, "Simplifying distributed data base systems design by using a broadcast network," in *Proc. ACM SIGMOD '84,* vol. 14, no. 2, 1984, pp. 223-233.

[8] J. Chang and N. F. Maxemchuk, "Reliable broadcast protocols," *ACM Trans. Comput. Syst.,* vol. 2, no. 3, pp. 251-273, Aug. 1984.

[9] D. R. Cheriton and W. Zwaenepoel, "Distributed process groups in the V kernel," *ACM Trans. Comput. Syst.,* vol. 3, no. 2, pp. 77-107, May 1985.

[10] D. R. Cheriton, "VMTP: A transport protocol for the next generation of communication systems," in *Proc. ACM Sigcomm Symp. Commun. Architect. Protocols,* 1986, pp. 406-415.

[11] F. Cristian, H. Aghili, and R. Strong, "Atomic broadcast: From simple message diffusion to Byzantine agreement," in *Proc. IEEE Symp. Fault Tolerant Computing Syst.,* 1985, pp. 200-206.

[12] *Data Communications Networks, Services and Facilities,* Red Book VIII.2, Geneva: CCITT, 1984.

[13] D. Dolev, C. Dwork, and L. Stockmeyer, "On the minimal synchronism needed for distributed consensus," *JACM,* vol. 34, no. 1, pp. 77-97, Jan. 1987.

[14] M. J. Fischer, N. A. Lynch, and M. S. Paterson, "Impossibility of distributed consensus with one faulty process," *JACM,* vol. 32, no. 2, pp. 374-382, Apr. 1985.

[15] H. Garcia-Molina and A. Spauster, "Message ordering in a multicast environment," in *Proc. IEEE 9th Int. Conf. Distrib. Computing Syst.,* 1989, pp. 354-361.

[16] N. L. Johnson and S. Kotz, *Urn Models and Their Application.* New York: Wiley, 1977.

[17] H. Kopetz *et al.,* "Distributed fault-tolerant real-time systems: The Mars approach," *IEEE Micro,* vol. 9, no. 1, pp. 25-40, Feb. 1989.

[18] H. Kopetz, G. Grüsteidl, and J. Reisinger, "Fault-tolerant membership service in a synchronous distributed real-time system," in *Proc. IFIP Int. Working Conf. Dependable Computing for Crit. Appl.,* 1989, pp. 167-174.

[19] S. W. Luan and V. D. Gligor, "A fault-tolerant protocol for atomic broadcast," in *Proc. IEEE 7th Symp. Reliable Distrib. Syst.,* 1988, pp. 112-126.

[20] P. M. Melliar-Smith and L. E. Moser, "Trans: A broadcast protocol for distributed systems," to be published.

[21] L. E. Moser, P. M. Melliar-Smith, and V. Agrawala, "On the impossibility of broadcast agreement protocols," to be published.

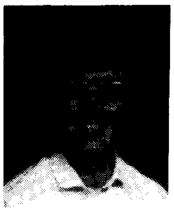
[22] ——, "Asymptotic broadcast agreement protocols," to be published.

[23] K. J. Perry and S. Toueg, "Distributed agreement in the presence of processor and communication faults," *IEEE Trans. Software Eng.,* vol. SE-12, no. 3, pp. 477-482, Mar. 1986.

[24] L. L. Peterson, N. C. Buchholz, and R. D. Schlichting, "Preserving and using context information in interprocess communication," *ACM Trans. Comput. Syst.,* vol. 7, no. 3, pp. 217-246, Aug. 1989.

P. M. Melliar-Smith (M'89) received the Ph.D. degree in computer science from the University of Cambridge, Cambridge, England, in 1987.

He was a senior research scientist and program director at SRI International in Menlo Park (1976-1987), senior research associate at the University of Newcastle Upon Tyne (1973-1976), and principal designer for GEC Computers Ltd. in England (1964-1973). He is currently a member of the faculty of the Department of Electrical and Computer Engineering, University of California, Santa Barbara. His research interests include fault-tolerant distributed systems, parallel processing, and temporal logic.



Louise E. Moser (M'87) received the Ph.D. degree in mathematics from the University of Wisconsin, Madison, in 1970.

From 1970 to 1987 she was a Professor of Mathematics and Computer Science, California State University, Hayward. In 1987 she moved to the University of California, Santa Barbara, where she has recently been appointed to a faculty position in the Department of Electrical and Computer Engineering. Her current research interests include parallel and distributed systems, fault tolerance, and formal specification and verification.

Vivek Agrawala was born in Bikaner, India, on August 28, 1963. He received the B.Tech. degree in chemical engineering in 1984 and the M.Tech. degree in computer technology in 1986 from the Indian Institute of Technology, Delhi.

Since September 1986, he has been working toward the Ph.D. degree in computer science at the University of California, Santa Barbara. His research interests include fault-tolerant communication protocols, distributed databases, algorithms, and complexity.