Early Delivery Totally Ordered Multicast in Asynchronous Environments

Danny Dolev*, Shlomo Kramer, Dalia Malki†

The Hebrew University of Jerusalem, Israel

Abstract

This paper presents the construction of a multicast *service, called* agreed multicast, *that guarantees that messages arrive reliably and in the same total-order to all their destinations.* ToTo, *a novel protocol, implements the agreed multicast service of Transis, a communication sub-system for the High Availability project, currently developed at the Hebrew University of Jerusalem. This service is desired in distributed systems, and supports high level coordination among groups of processes in distributed applications.*

The ToTo protocol is genuinely symmetric and fairly simple for implementing. It provides early delivery *latency, and requires as little as* $\frac{n}{2}$ *messages for forming agreement on the order of delivery. Experimental results show up to* $O(\log(n))$ *speedup over previous protocols, which match our prediction of the expected speedup.*

Using the Transis membership service, ToTo can operate in a dynamic environment, and continue to form an agreed total order among the connected machines despite failures and recoveries.

1 Introduction

This paper presents ToTo, a novel protocol for total ordering of messages in asynchronous environments.

The topic of consistent dissemination of information in distributed systems has been the focus of many studies, both theoretical and practical. Work as early as the V system ([8]), deals with communication among groups of processes, via *multicast messages*. In V, multicast messages are not reliable, but provide "best effort" delivery semantics. In addition, if messages are sent concurrently from several sources, the order of their delivery at overlapping destinations is undefined. Later work in the ISIS system ([3]), deals with providing higher level services, that preserve the relative order of message delivery at all sites. Many distributed applications require such high degree of coordination among their processes. The main difficulty facing the designer of a distributed application is the consistency of information disseminated, and the control over the dissemination of that information. Thus, the designer of a distributed system would wish for a service that provides a guaranteed delivery-and-consistency of multicast messages. An *agreed multicast* service guarantees that messages arrive reliably and in the same total-order to all their destinations. Having such a service, most distributed applications become much easier to implement and to maintain.

This paper presents a novel protocol that implements the agreed service in a multicast environment. The protocol was implemented as part of the Transis communication sub-system for the High Availability project, currently developed at the Hebrew University of Jerusalem.

The agreement on a total order usually bears a cost in performance: all the communicating machines must agree on a single total order of delivery, despite the fact that messages may be transmitted concurrently from different sites, and may take arbitrarily long to arrive at different sites (including delays incurred by losses and *retransmissions*). Informally, the performance cost is in the *latency* of agreed messages, measured from the point the message is ready to be sent until it is ready to be delivered at the sender machine.

In our approach, an agreed multicast message can be sent spontaneously by any machine, without prior enforced order and without the intervention of any central coordinator. This message is received by other machines, and is kept for later delivery. It cannot be delivered right away. Meanwhile, the communication in the system can continue as usual. At some later point, when enough information has been gathered from other machines, the message can be delivered. It is perhaps easiest to understand this through the simple idea of an *all-ack* protocol (see [22, 4, 15, 20, 24]):

*also at IBM Almaden Research Center

†This work was supported in part by GIF I-207-199.6/91

- Wait for a message from each machine in the system.

- Deliver the set of messages that do not causally follow any other, in lexicographical order.

This kind of agreed service has a *post-transmission* delay, caused by the first step. The length of the delay is set by the *slowest* machine to respond with a message. The ToTo protocol essentially optimizes the first step. It forms the total order by waiting for messages only from a majority set of the machines in the system; thus, it provides *early delivery* of agreed messages. Due to the asynchrony of the system and the possibility of failure, a simple majority decision is not sufficient to guarantee consistency, and ToTo requires an adaptive majority rule.

Interestingly, the post-transmission delay is most apparent when the system is relatively idle, and waiting for responses from all (or some) of the machines incurs the worst-case delay. On the other hand, when the communication in the system is heavy, the regular messages continuously form the total order, and the cost of agreement on the total order is amortized across a pipeline of agreements.

The ToTo protocol has the following desirable properties:

- The ToTo protocol is genuinely symmetric and does not incur any message overhead.

- The order formed by the ToTo protocol extends the underlying *causal order* (defined in Section 2).

- In a configuration with n machines, as few as $\frac{n}{2}+1$ messages from different machines are required in order to decide on the next set of messages to be added to the total order. Experiments with the ToTo protocol in the Transis environment show a factor of $O(\log(n))$ speedup over the all-ack protocol. These results match the theoretical predictions developed in [18].

- The ToTo protocol is fairly simple and does not bear a significant processing burden. ToTo was implemented in the Transis system, over UDP/IP. Experimentis show a sustained throughput of 250 1K messages per second between 8 Sun-4 machines connected a 10 Mbit/sec Ethernet, when all 8 machines were receiving all messages[1].

- The ToTo protocol preserves a bounded latency worst case behavior, in the sense that it always delivers a message after all the machines acknowledge its reception.

The ToTo protocol is completely asynchronous. It can operate in a dynamic environment, using the Transis *membership service* (see [1]). The membership service provides information about changes in the system configuration, such as removing and adding machines to the configuration. The service employs *timeout* to decide that the communication with another machine is detached (due to a machine crash or a communication failure). Thus, the agreed multicast service forms the total order despite failures and network partitions. This does not violate the impossibility of making consensus decisions deterministically ([13]) since our membership protocol may extract machines from the configuration, even though they may have not failed. The membership protocol always terminates. However, there is a non-zero probability that a machine or a set of machines can unjustfully extract all the other machines from their membership and continue to form the total order on their own.

Related Work

The problem of total ordering has received considerable attention in the distributed processing community. Synchronous protocols for the total ordering problem have been studied for some time ([14, 12, 16, 11]). The weakness of that approach is that decisions should be handled "pessimistically," i.e., waiting for the worst possible sequence of events, and for the longest possible time. Furthermore, the need to maintain the underlying assumptions about synchrony may add to the complexity of the communication system. Probabilistic protocols introduce random steps to the protocol (see a survey in [9]). These steps are controlled by the random values of *global coins*. In order to produce global coins, probabilistic protocols usually exchange many additional messages and thus are impractical to use in high performance communication systems. Furthermore, these protocols are only partially correct in the sense that they terminate with a probability that increases asymptotically to one as the protocol advances. Asymmetric protocols ([6, 7, 17]) use a centralized coordinator for ordering the messages. The problems with this method are in the serial bottleneck it creates at the coordinating site, and with the costly handling of faults in case the coordinator crashes.

Most existing symmetric protocols ([22, 4, 15, 20, 24]) require all machines to relay their up-to-date view

[1] In a recent upgrade of the Transis system, we measured twice this throughput and more, but we did not repeat all the experiments yet.

on the latest delivered messages in order for a new message to be added to the total order. These methods are essentially *all-ack*, i.e. require all machines to send ack for a single message delivery. Unfortunately, all-ack protocols typically bear a considerable cost in message latency or message overhead.

Recent work by Melliar-Smith et al. ([21]) proposed a symmetric protocol called Total. In Total, the machines never exchange additional messages for creating the total order. The Total protocol continues to form the total order even in the presence of machine crashes and network partitions. The protocol uses a voting criterion to decide on the next set of messages to be added to the total order. In a configuration with n machines, a k-resilient Total protocol can reach a decision after collecting as few as $\frac{n+k+1}{2}$ votes from different machines. However, Total is too complicated, bearing a considerable processing burden. Thus, the Total protocol cannot be used in high performance communication systems. Furthermore, Total achieves only *partial correctness*. Even in *faultless runs*, Total may further delay delivery of a message after any number of acknowledgments had been received from all other machines.

2 Transis and System Model

The system comprises of a set of machines that communicate via asynchronous multicast messages. A multicast message leaves its source machine at once to all the destined machines in the system but may arrive at different times to them. Messages might be lost or delayed arbitrarily, but faults cannot alter messages' contents. Messages are uniquely identified through a pair < sender, counter > .

In this section we give a brief explanation of the multicast services supplied by Transis and discuss the assumptions made by the ToTo protocol about these services. A detailed description of the Transis environment and services can be found in [2]. ToTo can be used in similar environments, that provide similar multicast services.

2.1 Multicast Services

Transis contains the communication layer responsible for the reliable delivery of messages in the system ([2]). Transis guarantees the *causal* (see [19]) delivery order of messages, defined as the reflexive, transitive closure of:

(1) $m \xrightarrow{cause} m'$ if $\text{receive}_q(m) \rightarrow \text{send}_q(m')$ [2]

[2]Note that '$\rightarrow$' orders events occurring at q sequentially, and therefore the order between them is well defined.

(2) $m \xrightarrow{cause} m'$ if $\text{send}_q(m) \rightarrow \text{send}_q(m')$.

If m' follows m in the causal order, we say that m' *follows* m. Message m is *prior to* m' if m' follows m. For simplicity of the definitions that follow, we say that a message follows itself. Messages m, m' are *concurrent* if m does not follow m', and m' does not follow m. The set of all messages concurrent to m is called the *concurrency set* of m.

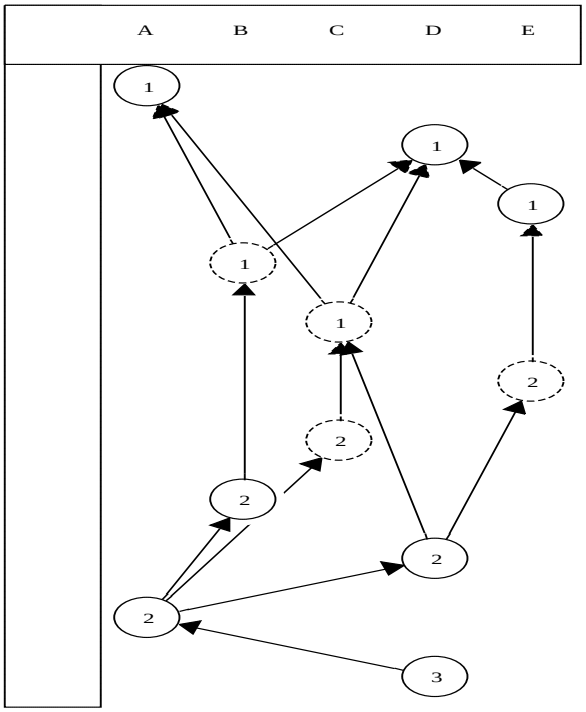

Figure 1: The causal order.

In Transis, each newly emitted message contains ACKs to previous messages. The ACKs form the $\xrightarrow{cause}$ relation directly, such that if m' contains an ACK to m, then $m \xrightarrow{cause} m'$. Implicit ACKs are readily available using the transitive closure of the $\xrightarrow{cause}$ relation. For example, in Figure 1, ACKs are denoted by arrows. Message $m_{d,2}$ (the second message from D) explicitly acknowledges messages $m_{c,1}$, $m_{e,2}$ and implicitly acknowledges messages $m_{a,1}$, $m_{d,1}$ and $m_{e,1}$. If a message arrives at a machine, and some of its causal predecessors are missing, Transis transparently handles message recovery and re-ordering. Other environments like [6, 22] are equally suitable for providing the causality requirement. Below, we sometimes refer to the environment and messages as the Transis

environment and Transis messages.

We think of the causal order as a directed acyclic graph (DAG): the nodes are the messages, the arcs connect two messages that are directly dependent in the causal order. All the machines eventually see the same DAG, although as they progress, it may be "revealed" to them gradually in different orders. We denote by DAG_p the current DAG at machine p.

The Transis communication sub-system receives the messages off the network. It performs recovery and message handling, and at some later time, it *delivers* the messages to the upper level. Transis provides a variety of reliable multicast services. The services use different delivery criteria on the messages in the DAG. To simplify our discussion we will restrict ourselves to an environment in which only agreed multicast and causal multicast messages are possible.

3 The Agreed Multicast Service

The agreed multicast service extends the underlying causal order to a total order on the agreed multicast messages. As long as partitions do not occur, all machines deliver the agreed multicast messages in the same total order. When a partition occurs, machines in every partition continue to form the same total order. However, this total order may differ across partitions. More formally, the agreed multicast service definition is defined in Figure 2.

The Agreed Multicast Service Definition

Correctness

Let m, m' be two agreed multicast messages:

- If m' follows m, all machines that deliver m and m' deliver m before m'.
- If m and m' are concurrent, let p be a machine s.t. p delivered m before delivering m'. For any machine q that delivers m' the following two cases are possible:
  1. q delivers m before delivering m'.
  2. If q delivers m' before m, q and p are not connected.

Liveness

Let m be an agreed multicast message. Let machine p receive m at time T. There is a time interval Δ s.t. by $T + \Delta$, message m had been delivered at p.

Figure 2: The Agreed Multicast Service Definition

The set of connected machines is maintained by the membership service. In Section 5 we give the exact definition of this notion. Intuitively, two machines are not connected when they belong to two different partitions. Note that the liveness property is defined in terms of the subjective time at machine p and does not assume the existence of any common global time.

4 The ToTo Protocol

This section assumes that the system comprises of a static configuration of n machines. In the next section (5) we show how to extend the algorithm to handle dynamicity. Therefore, in this section there is no guarantee of *liveness*, but in the extended protocol of Section 5 this guarantee is completed.

4.1 Notation and Definitions

The input to the ToTo protocol is a stream of causally ordered messages. We denote by $m_{p,i}$ the i'th message sent by machine p.

We define a *pending message* to be an agreed multicast message that was received by the protocol but has not yet been added to the total order, thus, not delivered for processing. A pending message that follows only delivered messages is called a *candidate message*. The set of current candidate messages is called the *candidate set*. This is the set of messages that are considered for the next slot in the total order. For example, in Figure 1, agreed multicast messages are denoted by the dashed circles, while causal multicast messages are denoted by full-line circles. Assume that messages $m_{a,1}$, $m_{d,1}$, and $m_{e,1}$ are delivered. Then the candidate set is $\{m_{b,1}, m_{c,1}, m_{e,2}\}$, while $m_{c,2}$ is a pending agreed message which, since it follows $m_{c,1}$, is not a candidate message.

Let $M_p = \{m_1, \ldots, m_k\}$ be the set of candidate messages at machine p. We associate with each message m_i the functions $Tail_p$, $NTail_p$, VT_p, and NVT_p:

1. $Tail_p(m_i) =$
   $\{q \mid \exists m_{q,j}\ s.t.\ m_{q,j} \in DAG_p,\ m_{q,j}\ follows\ m_i\}$.
   $Tail_p(m_i)$ is the set of all machines that sent a message causally following m_i in the current DAG_p. We also denote:
   $$Tail_p = \bigcup_{m_i \in M_p} Tail_p(m_i).$$
   $Tail_p$ is the set of all machines that sent a message following some candidate messages. Intuitively, $Tail_p$ contains all the machines we have information from.
2. $NTail_p(m_i) = \mid Tail_p(m_i) \mid$ and
   $NTail_p = \mid Tail_p \mid$.

3. Every candidate message m_i is associated with an *Extended Vector Time Stamp* (EVTS), $VT_p(m_i)$. The EVTS holds one entry per machine. Let q be a machine and let $m_{q,k}$ be the first message from q (if exists) causally following any candidate message:

$$VT_p(m_i)[q] = \begin{cases} * & \text{if } q \notin Tail_p \\ k & \text{if } m_{q,k} \ follows \ m_i \\ \infty & \text{otherwise.} \end{cases}$$

4. $NVT_p(m_i) = |\{q \mid VT_p(m_i)[q] \neq *, \ VT_p(m_i)[q] \neq \infty\}|$.

As a compact notation we sometimes denote $Tail_{pi} \equiv Tail_p(m_i)$, $NTail_{pi} \equiv NTail_p(m_i)$, $VT_{pi} \equiv VT_p(m_i)$, and $NVT_{pi} \equiv NVT_p(m_i)$.

The above functions are dynamically computed at each machine as messages arrive and are inserted to the local DAG. Note that the local DAG is an implicit parameter to all of these functions. In Table 1 we present these function values for the DAG depicted in Figure 1 (assuming that $m_{a,1}$, $m_{d,1}$ and $m_{e,1}$ were delivered already):

candidate	Tail_p	NTail_p	VT_p	NVT_p
$m_{b,1}$	$\{a, b, d\}$	3	$(2, 1, \infty, \infty, \infty)$	2
$m_{c,1}$	$\{a, c, d\}$	3	$(2, \infty, 1, 2, \infty)$	3
$m_{e,2}$	$\{a, d, e\}$	3	$(2, \infty, \infty, 2, 2)$	3

Table 1: Candidates' quantifying functions for the DAG of Figure 1.

Let $\Phi \geq \frac{n}{2}$ be a threshold parameter. Using VT_p, $Tail_p$ and $NTail_p$, we define two comparison functions, Win_p and $Future_p$:

$$Win_p(VT_{p1}, VT_{p2}) = \begin{cases} 1 & \text{if } |\{j \mid VT_{p1}[j] < VT_{p2}[j]\}| > \Phi \\ 0 & \text{if } |\{j \mid VT_{p2}[j] < VT_{p1}[j]\}| > \Phi \\ \chi & \text{otherwise.} \end{cases}$$

It is easy to verify that for $\Phi \geq \frac{n}{2}$:

$$Win_p(VT_{p1}, VT_{p2}) = 1 \implies \forall i \ Win_p(VT_{p2}, VT_{pi}) \neq 1. \quad (W.1)$$

As messages arrive and are inserted to DAG_p, Win_p changes. Win_p may change its value from χ to 1 or to 0. At a certain stage, Win_p fixes its value permanently. The latest this can be is when $NTail_p = n$.

We define $Future_p(VT_{p1}, VT_{p2})$ to be the set of all possible permanent values of $Win_p(VT_{p1}, VT_{p2})$. More formally: denote DAG_p^+ to be any possible extension to DAG_p (*i.e.* by any possible run) in which $NTail_p = n$. Denote $Tail_p^+$, VT_p^+, to be the corresponding functions defined on DAG_p^+. Then:

$$Future_p(VT_{p1}, VT_{p2}) = \{Win_p(VT_{p1}^+, VT_{p2}^+) \mid \ DAG_p^+ \ extending \ DAG_p\}.$$

$Future_p$ is a set valued function. $Future_p$'s value can intuitively be thought of as the set of all future possible comparison values. Given VT_{p1} and VT_{p2}, $Future_p$ is easily determined:

- $1 \in Future_p(VT_{p1}, VT_{p2}) \iff |\{j \mid VT_{p2}[j] \leq VT_{p1}[j]\}| < n - \Phi$.
- $\chi \in Future_p(VT_{p1}, VT_{p2}) \iff Win_p(VT_{p1}, VT_{p2}) = \chi$.
- $0 \in Future_p(VT_{p1}, VT_{p2}) \iff 1 \in Future_p(VT_{p2}, VT_{p1})$.

4.2 The $ToTo_\Phi$ Protocol

Let $\frac{n}{2} \leq \Phi < n$, we describe the $ToTo_\Phi$ protocol. Since the protocol is symmetric we describe it for a specific machine p. A candidate message m is a *source* if for every other candidate message m', $1 \notin Future_p(VT_p(m'), VT_p(m))$. Let S_p denote the set of sources at p. We define the following delivery criterion:

(DEL) Deliver S_p when :

1. Internal stability: $\forall m \in M_p - S_p \quad \exists m' \in M_p$ s.t. $Future(VT_p(m'), VT_p(m)) = \{1\}$. And,
2. External stability:
   (a) $\exists s \in S_p \quad NVT_p(s) > \Phi$ and $\forall s \in S_p \ NTail_p(s) \geq n - \Phi$.
   Or,
   (b) $NTail_p = n$.

The protocol is completely asynchronous and is described in an event-driven fashion. It specifies how to handle incoming messages, their delivery criterion and the effect taken when they are delivered. Figure 3 describes the protocol.

Using the ToTo protocol, the distribution of message latency directly depends on the parameter Φ. Increasing Φ can reduce message latency only when the waiting time for criterion (DEL) 2(a):

$\exists s \in S_p \ NVT_p(s) > \Phi \ and \ \forall s \in S_p \ NTail_p(s) \geq n - \Phi$

- When receiving a regular message $m_{q,k}$:
  1. If $m_{q,k}$ is a new candidate message then for every machine $r \in Tail_p$ set $VT_p(m_{q,k})[r] = \infty$. Add $m_{q,k}$ to M_p.
  2. If $q \notin Tail_p$, then for all $m_i \in M_p$ s.t. $m_{q,k}$ *follows* m_i set $VT_{pi}[q] = k$. For all $m_i \in M_p$ not followed by $m_{q,k}$, set $VT_{pi}[q] = \infty$.
- When criterion (DEL) holds: deliver all messages in S_p in a lexicographical order. Update M_p and recalculate the new values of $VT_p(m)$ for every m in M_p. Set $Nact$=$Nact$+1.

Figure 3: The $ToTo_\Phi$ protocol

is dominated by the condition:

$$\forall s \in S_p \quad NTail_p(s) \geq n - \Phi.$$

Intuitively, this will happen when $k = \mid S_p \mid$ is large. For a large k, the waiting time for all candidate messages to collect $n - \Phi$ acknowledgments ($NTail_p(s) \geq n - \Phi$ for every source message s) will dominate the waiting time for any candidate message to collect Φ first acknowledgments ($NVT_p(s) > \Phi$ for some source message s). As we demonstrate in Section 6, for communication systems involving a few machines on a single broadcast LAN, the DAG is narrow. Thus, for these type of communication systems, the optimal Φ is $\frac{n}{2}$. We expect that broadcasting among many machines, simulating broadcast on a segmented network or using point to point communication, will give rise to communication scenarios in which the DAG is wider. For these systems we expect the optimal Φ to be greater than $\frac{n}{2}$.

The ToTo protocol was implemented as part of the Transis communication system. Details about this implementation can be found in [18]. Denote by n_c the number of candidate messages. Denote by n_p the number of pending messages. In our implementation, the ToTo protocol may inflict a complexity of $O(n_p + n_c^2)$ operations on message reception, a complexity of $O(n_c^2)$ operations on the processing of a crash event and a complexity of $O(nn_c^2)$ operations on delivery of a message set and the initialization of a new activation. As we show in Section 6, using broadcast communication the DAG is narrow and thus n_c is small. In this case the complexity inflicted by the ToTo protocol is not high.

Proofs for the correctness and liveness of the ToTo protocol can be found in [18].

5 Dynamic Environment

The ToTo protocol operates on an asynchronous stream of causal messages. Should one or more machines fail, the ToTo protocol might hold the delivery of agreed messages indefinitely, waiting for messages from the failed machines. Furthermore, so far, the protocol does not account for the *joining* of new machines into the system. In this section, we show how ToTo can be extended to operate in a dynamic environment. We present the essentials of the *membership service* of Transis, and show how to incorporate membership information into ToTo, such that the consistency of the agreed multicast service is preserved among all the connected machines.

5.1 The Membership Service

The problem of maintaining membership consensus is formally described in [10, 23]. Transis contains a membership service, that maintains a consistent view of the *current configuration set* (CCS) among all the connected machines in a dynamic environment. The membership service interjects special membership messages among the stream of regular messages, that provide information about changes to the CCS. Thus, each regular message is sent and delivered within a certain *membership context*.

When machines crash or disconnect, the network partitions or re-merges, the connected machines must reconfigure and reach a new agreement on the CCS. Faults and detachments are detected based on timeout. The joining of new machines is spontaneous, and is trigerred when machines that were not connected (or up) before, become connected.

The internal membership algorithm operation can be schematically presented as consisting of three "stages":

- *Initiate*: initiate a configuration change procedure.
- *Agree*: all connected machines dynamically agree on the next configuration change to be performed. Furthermore, all connected machines agree on the set of regular messages that belong to the previous membership, and must be delivered before the next configuration change. This last property is termed by Birman et al. *virtual synchrony*, and its importance is discussed in [5, 6].
- *Flush*: after all the messages in the above agreed upon set have been delivered, compose a *configuration change* message and deliver it to the application.

The *Initiate* and *Agree* stages, when invoked, terminate within a finite delay of time. This is possible since the membership protocol never allows indefinite blocking but uses timeout to ensure progress and to extract machines from the configuration, possibly unjustfully. Note that this does not mean that we make synchronous assumptions. Time is not used as bearing information. In the algorithm, timeouts are measured using local clocks and there is no requirement of synchronization among clocks (see [1]).

The *Flush* stage may involve the delivery of agreed multicast messages of the previous membership. This occurs when pending messages wait for a detached machine q to acknowledge their reception. To solve this potential deadlock, the membership protocol must supply the multicast protocols with an intermediate event. This event is generated upon completion of the *Agree* stage and indicates that machine q had detached and no more messages from q will be received henceforth[3]. This event is called $Crash(q)$. Note that this event is different from the configuration change itself, which occurs upon delivery of a configuration change message. A $Crash(q)$ event is internal, it does not change the CCS, and is not associated with a message in the DAG.

To summarize, the ToTo protocol uses the following properties of the membership service:

- *Liveness:* the *Initiate* and *Agree* stages, when invoked, always complete within a finite delay of time. Upon completion of the *Agree* stage, internal *Crash(q)* events are generated, for each machine q that is identified as disconnected.
- *Virtual synchrony:* configuration change messages are delivered in the same order at all connected sites. Furthermore, all connected sites deliver the same set of messages between every pair of configuration changes.

5.2 The Extended ToTo protocol

These properties of the membership service of Transis guarantee that when a configuration-change message is delivered, the M_p set of ToTo is empty. Thus, the only extension to the ToTo protocol is to handle $Crash(q)$ events. We define $Defunc_p$ to be the set of all machines, q, for which p received a $Crash(q)$ event and q is still in CCS. We extend $Tail_p$ to contain $\bigcup_{m_i \in M_p} Tail_p(m_i) \cup Defunc_p$.

Finally, we need to specify how to handle $Crash(q)$ events. The algorithm in Figure 3 needs to be extended with the handler in Figure 4.

[3] More precisely, until q rejoins the configuration.

- When receiving a crash event, $Crash(q)$: if $q \notin Tail_p$ then set $VT_{pi}[q] = \infty$ for every $m_i \in M_p$. Add q to $Defunc_p$.

Figure 4: Handling of Crash(q) events

6 Performance Results

In this Section we present performance results obtained for the ToTo protocol in the Transis environment. First, a new measure of efficiency for distributed agreed multicast protocols is described. We proceed to present the results obtained. Finally, these results are compared with the predictions of our performance-model.

Index of Latency

Implementing the agreed multicast in a distributed way, such as ToTo, does not incur extraneous message passing. Messages that are received are kept for a while at each machine, until there is enough information to deliver them. The communication in the system continues meanwhile, and thus ordering is done as a pipeline of agreements. However, we are interested in the latency of each single message.

Let m be an agreed multicast message sent or received by machine p. We define the *index of latency* of m as the number of machines that p receives messages from, before it can deliver m. These messages may be concurrent to m, or follow m. The efficiency of an agreed multicast protocol may be measured in terms of the worst case and average case index of latency. Most existing symmetric protocols ([22, 4, 15, 20]) have index of latency n, the number of machines. Thus, they always need to wait for messages from all the machines, including the slowest one. It is easy to see that symmetric protocols, such as ToTo, must bear an index of latency greater then $\frac{n}{2}$ (think of two messages received and acknowledged by two disjoint sets of $\frac{n}{2}$ processors).

The motivation for developing the ToTo protocol was to reduce the average-case index of latency from n. The index of latency of $ToTo_\Phi$ ranges between $\Phi+1$ to n. A natural question is how does the reduced index of latency influence message latency. In [18] we develop a simple model of performance for agreed multicast protocols, to study this. The index of latency of the protocol is modeled by a random variable τ. The random variable ω models the number of machines having a concurrent message in DAG_p to a newly received message. We present the model predicition and the reality of τ, ω and the speedup of ToTo.

Performance Results

All performance results quoted in this Section have been obtained in the Transis environment, implemented over UDP/IP. The system included 8 Sun-4 machines connected by a 10 Mbit/Sec Ethernet. In every test the machines exchanged 4000 agreed multicast messages. In all our tests, the average size of the candidate set turned out to be between 1.0 and 2.0 (see the values of $\bar{\omega}$ below). For such narrow DAGs, one cannot hope to gain anything from increasing Φ. Thus the optimal Φ for this communication system is 4.

Results for two types of *message sources* are presented:

- The *periodic* source sends a message every fixed interval of time.
- The *Poisson* source sends a message every random interval of time. The length of this interval is the value of a random variable with an exponential distribution.

Figures 5 and 6 depict the sample distribution of ToTo's index of latency for various communication scenarios. These figures indicate that ToTo's index of latency is concentrated around $\Phi + 1$. For 8 machines and Φ set to 4, ToTo decreased the average index of latency to $5.1 - 5.9$, very close to the lower bound.

In [18] we obtain formulas for the expected message latency in these two communication scenarios:

$$Pois = \frac{1}{\lambda} \sum_{i=\frac{n}{2}+1}^{n} \sum_{j=1}^{i} P(\tau = i, \omega = j) \sum_{k=j}^{i-1} \frac{1}{n-k} \quad (1)$$

$$Per = c \sum_{i=\frac{n}{2}+1}^{n} \sum_{j=1}^{i} P(\tau = i, \omega = j) \frac{i-j}{n-j+1} \quad (2)$$

Here, $\frac{1}{\lambda}$ and c are the expected length of the time interval between two consecutive sends.

We have measured the time latency of agreed multicast messages sent by these type of sources for various communication loads. Tables 2 and 3 present the results of these tests.

For the periodic source, when communication loads range between 50 to 200 messages per second, ToTo's speedup is around 1.75. For the Poission source, the speedup is around 2.75. In all the tests we have performed, ToTo's average index of latency τ ranged between 5.1 to 5.9, very close to the optimal. One can also note that $\bar{\omega}$ ranges between 1.1 to 1.8, indicating that the *DAG* is narrow.

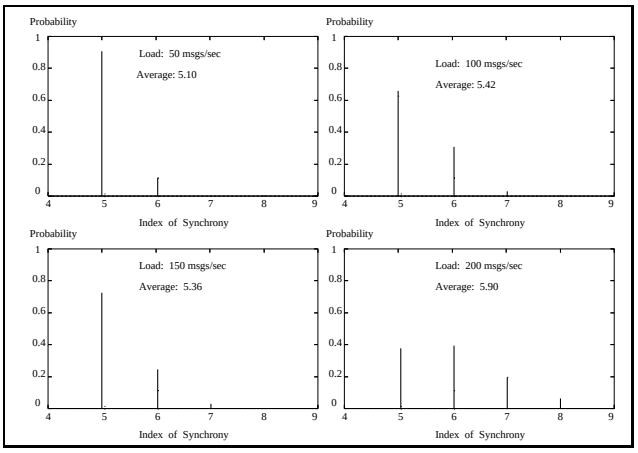

Figure 5: Index of latency, periodic source.

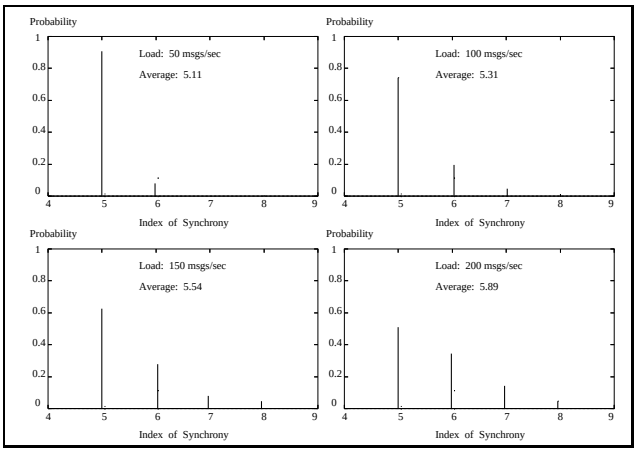

Figure 6: Index of latency, Poisson source.

Model and Reality

We have measured the joint sample distributions of (τ, ω) in the Transis environment for various communication scenarios. Using these sample distributions of (τ, ω) and Equations 1, 2, we computed the predicted average message latency and compared it with the observed average message latency in Tables 4 and 5. We denote by ρ the deviation of the predicted average latency from the observed average latency:

$$\rho = \frac{\mid l_{model} - l_{observed} \mid}{l_{observed}} .$$

For the periodic source (2), the model's predictions were very good: 0.4 to 6.0 percent deviation. For the Poisson source, the predictions were still good

but the deviation is slightly higher: 2.8 to 18.7 percent. This larger deviation is due to the fact that the *received* stream of messages is only approximately Poisson. Two causes contribute to this inaccuracy: the application sends a stream of messages which is only approximately Poisson. Furthermore, the emitted stream of messages is further delayed both at the sender's and at the receiver's kernel, causing a further dent to the distribution.

7 Conclusions

We presented an early-delivery total-ordering protocol called ToTo, that requires as little as $\frac{n}{2}+1$ machines to exchange information, in order to form the agreement on the order of messages. This information is piggybacked upon regular messages. Thus, when an *agreed* message is sent, its delivery is delayed until more messages (*e.g.* $\frac{n}{2}+1$) are received. This does not block the application or the communication system. Furthermore, as soon as one message is delivered, the following messages are advanced towards their delivery. In this way, the latency cost of total ordering is amortized across a pipeline of agreements.

Our performance results are encouraging: they indicate that ToTo is suitable for usage in broadcast environments, and gains the predicted speedup in latency. We anticipate ToTo to become valuable in constructing a *wide-area-network* agreed multicast service, since the variance in communication delays is much larger, and therefore waiting for communication with a specific machine may be too costly.

Load (msgs/sec)	Protocol	$\bar{\tau}$	$\bar{\omega}$	l (ms)	Speedup
50	$ToTo$	5.10	1.12	76.68	1.76
	$all-ack$	8.00	1.69	135.02	
100	$ToTo$	5.42	1.39	42.31	1.63
	$all-ack$	8.00	1.31	68.78	
150	$ToTo$	5.36	1.44	29.17	1.56
	$all-ack$	8.00	1.81	45.59	
200	$ToTo$	5.90	1.72	23.80	1.84
	$all-ack$	8.00	1.48	43.71	

Table 2: Performance results for a periodic source.

Load (msgs/sec)	Protocol	$\bar{\tau}$	$\bar{\omega}$	l (ms)	Speedup
50	$ToTo$	5.11	1.15	114.10	3.38
	$all-ack$	8.00	1.13	385.25	
100	$ToTo$	5.31	1.29	60.98	3.08
	$all-ack$	8.00	1.27	188.06	
150	$ToTo$	5.54	1.42	44.10	3.01
	$all-ack$	8.00	1.42	133.09	
200	$ToTo$	5.89	1.63	39.37	2.60
	$all-ack$	8.00	1.58	102.33	

Table 3: Performance results for a Poisson source.

Load (msgs/sec)	Protocol	l_{model} (ms)	$l_{observed}(ms)$	ρ
50	$ToTo$	75.67	76.68	0.013
	$all-ack$	137.51	135.02	0.018
100	$ToTo$	44.87	42.31	0.060
	$all-ack$	69.35	68.78	0.008
150	$ToTo$	29.29	29.17	0.004
	$all-ack$	47.34	45.59	0.038
200	$ToTo$	24.78	23.80	0.041
	$all-ack$	35.96	43.71	0.177

Table 4: Predicted average latency vs. observed average latency for the periodic source.

Load (msgs/sec)	Protocol	l_{model} (ms)	$l_{observed}(ms)$	ρ
50	$ToTo$	122.34	114.10	0.072
	$all-ack$	396.17	385.25	0.028
100	$ToTo$	66.64	60.98	0.093
	$all-ack$	194.58	188.06	0.035
150	$ToTo$	52.34	44.10	0.187
	$all-ack$	137.79	133.09	0.035
200	$ToTo$	46.12	39.37	0.171
	$all-ack$	111.45	102.33	0.089

Table 5: Model predicted average latency vs. observed average latency for the Poisson source.

References

[1] Y. Amir, D. Dolev, S. Kramer, and D. Malki. Membership Algorithms for Multicast Communication Groups. In *6th Intl. Workshop on Distributed Algorithms proceedings (WDAG-6), (LCNS, 647)*, pages 292–312, November 1992.

[2] Y. Amir, D. Dolev, S. Kramer, and D. Malki. Transis: A Communication Sub-System for High Availability. In *22nd Annual International Symposium on Fault-Tolerant Computing*, pages 76–84, July 1992.

[3] K. Birman, R. Cooper, T. A. Joseph, K. Marzullo, M. Makpangou, K. Kane, F. Schmuck, and M. Wood. *The ISIS System Manual*. Dept of Computer Science, Cornell University, Sep 90.

[4] K. Birman and T. Joseph. Realiable Communication in the Presence of Failures. *ACM Trans. Comput. Syst.*, 5(1):47–76, February 1987.

[5] K. Birman and T. Joseph. Exploiting Virtual Synchrony in Distributed Systems. In *11th Ann. Symp. Operating Systems Principles*, pages 123–138. ACM, Nov 87.

[6] K. Birman, A. Schiper, and P. Stephenson. Lightweight Causal and Atomic Group Multicast. *ACM Trans. Comput. Syst.*, 9(3):272–314, 1991.

[7] J. M. Chang and N. Maxemchuck. Realiable Broadcast Protocols. *ACM Trans. Comput. Syst.*, 2(3):251–273, August 1984.

[8] D. R. Cheriton and W. Zwaenepoel. Distributed Process Groups in the V Kernel. *ACM Trans. Comput. Syst.*, 2(3):77–107, May 1985.

[9] B. Chor and C. Dwork. Randomization in Byzantine Agreement. In S. Micali, editor, *Advances in Computing Research, Randomness in Computation*, volume 5, pages 443–497. JAI Press, 1989.

[10] F. Cristian. Reaching Agreement on Processor Group Membership in Synchronous Distributed Systems. *Distributed Computing*, 4(4):175–187, April 1991.

[11] 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.

[12] D. Dolev and H. R. Strong. Authenticated algorithms for byzantine agreement. *SIAM J. Comput.*, 12(3):656–666, Nov 1983.

[13] M. Fischer, N. Lynch, and M. Paterson. Impossibility of Distributed Consensus with One Faulty Process. *J. ACM*, 32:374–382, April 1985.

[14] M. J. Fischer. The consensus problem in unreliable distributed systems (a brief survey). In M. Karpinski, editor, *Foundations of Computation Theory*, volume 158 of *Lecture Notes in Computer Science*, pages 127–140. Springer-Verlag, 1983.

[15] K. J. Goldman. Highly Concurrent Logically Synchronous Multicast. *Distributed Computing*, 4(4):189–208, 1991.

[16] A. Griefer and R. Strong. DCF: Distributed Communication with Fault Tolerance. In *7th Ann. Symp. Principles of Distributed Computing*, pages 18–27, August 1988.

[17] M. F. Kaashoek, A. S. Tanenbaum, S. F. Hummel, and H. E. Bal. An Efficient Reliable Broadcast Protocol. *Operating Systems Review*, 23(4):5–19, October 1989.

[18] S. Kramer. Total ordering of messages in multicast communication systems. Master's thesis, Hebrew University, Jerusalem, Il, December 1992.

[19] L. Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. *Comm. ACM*, 21(7):558–565, July 78.

[20] S. W. Luan and V. D. Gligor. A Fault-Tolerant Protocol for Atomic Broadcast. *IEEE Trans. Parallel & Distributed Syst.*, 1(3):271–285, July 90.

[21] P. M. Melliar-Smith, L. E. Moser, and V. Agrawala. Broadcast Protocols for Distributed Systems. *IEEE Trans. Parallel & Distributed Syst.*, (1), Jan 1990.

[22] L. L. Peterson, N. C. Buchholz, and R. D. Schlichting. Preserving and Using Context Information in Interprocess Communication. *ACM Trans. Comput. Syst.*, 7(3):217–246, August 89.

[23] A. M. Ricciardi and K. P. Birman. Using Process Groups to Implement Failure Detection in Asynchronous Environments. In *proc. annual ACM Symposium on Principles of Distributed Computing*, pages 341–352, August 1991.

[24] P. Verissimo, L. Rogrigues, and J. Rufino. The Atomic Multicast Protocol (AMp). In D. Powell, editor, *Delta-4: A Generic Architecture for Dependable Distributed Computing*, pages 267–294. Springer-Verlag, 1991.