Lightweight Causal and Atomic Group Multicast

KENNETH BIRMAN
Cornell University

ANDRÉ SCHIPER
Ecole Polytechnique Fédéral de Lausanne, Switzerland
and
PAT STEPHENSON
Cornell University

The ISIS toolkit is a distributed programming environment based on virtually synchronous process groups and group communication We present a new family of protocols in support of this model Our approach revolves around a multicast primitive, called CBCAST, which implements fault-tolerant, causally ordered message delivery. CBCAST can be used directly, or extended into a totally ordered multicast primitive, called ABCAST. It normally delivers messages immediately upon reception, and imposes a space overhead proportional to the size of the groups to which the sender belongs, usually a small number. Both protocols have been implemented as part of a recent version of ISIS and we discuss some of the pragmatic issues that arose and the performance achieved. Our work leads us to conclude that process groups and group communication can achieve performance and scaling comparable to that of a raw message transport layer—a finding contradicting the widespread concern that this style of distributed computing may be unacceptably costly.

Categories and Subject Descriptors: C.2.1 [**Computer-Communication Networks**]: Network Architecture and Design—*network communications*; C.2.2 [**Computer Communication Networks**]: Network Protocols—*protocol architecture*; C.2.4 [**Computer-Communication Networks**]: Distributed Systems—*distributed applications, network operating systems*; D.4.1 [**Operating Systems**]: Process Management—*concurrency, synchronization*; D.4.4 [**Operating Systems**]: Communications Management—*message sending, network communication*; D.4.7 [**Operating Systems**]: Organization and Design—*distributed systems*

General Terms: Algorithms, Reliability

Additional Key Words and Phrases: Fault-tolerant process groups, message ordering, multicast communication

---

This work was supported by the Defense Advanced Research Projects Agency (DoD) under DARPA/NASA subcontract NAG2-593 administered by the NASA Ames Research Center, and by grants from GTE, IBM, and Siemens, Inc. The views, opinions, and findings contained in this report are those of the authors and should not be construed as an official Department of Defense position, policy, or decision

Authors' Addresses: K. Birman and P. Stephenson, Cornell University, Department of Computer Science, 4130 Upson Hall, Ithaca, NY 14853-7501; A. Schiper, Ecole Polytechnique Fédérale de Lausanne, Switzerland.

1. INTRODUCTION

1.1 The ISIS Toolkit

The ISIS toolkit [8] provides a variety of tools for building software in loosely coupled distributed environments. The system has been successful in addressing problems of distributed consistency, cooperative distributed algorithms and fault-tolerance. At the time of this writing, Version 2.1 of the Toolkit was in use at several hundred locations worldwide.

Two aspects of ISIS are key to its overall approach:

—An implementation of *virtually synchronous process groups*. Such a group consists of a set of processes cooperating to execute a distributed algorithm, replicate data, provide a service fault-tolerantly or otherwise exploit distribution.

—A collection of reliable multicast protocols with which processes and group members interact with groups. Reliability in ISIS encompasses *failure atomicity*, *delivery ordering guarantees* and a form of *group addressing atomicity*, under which membership changes are synchronized with group communication.

Although ISIS supports a wide range of multicast protocols, a protocol called CBCAST accounts for the majority of communication in the system. In fact, many of the ISIS tools are little more than invocations of this communication primitive. For example, the ISIS replicated data tool uses a single (asynchronous) CBCAST to perform each update and locking operation; reads require no communication at all. A consequence is that the cost of CBCAST represents the dominant performance bottleneck in the ISIS system.

The original ISIS CBCAST protocol was costly in part for structural reasons and in part because of the protocol used [6]. The implementation was within a protocol server, hence all CBCAST communication was via an indirect path. Independent of the cost of the protocol itself, this indirection was expensive. Furthermore, the protocol server proved difficult to scale, limiting the initial versions of ISIS to networks of a few hundred nodes. With respect to the protocol used, our initial implementation favored generality over specialization thereby permitting extremely flexible destination addressing. It used a *piggybacking* algorithm that achieved the CBCAST ordering property but required periodic garbage collection.

The case for flexibility in addressing seems weaker today. Experience with ISIS has left us with substantial insight into how the system is used, permitting us to focus on core functionality. The protocols described in this paper support highly concurrent applications, scale to systems with large numbers of potentially overlapping process groups and bound the overhead associated with piggybacked information in proportion to the size of the process groups to which the sender of a message belongs. Although slightly less general than the earlier solution, the new protocols are able to support the ISIS toolkit and all ISIS applications with which we are familiar. The benefit of this reduction in generality has been a substantial increase in the

performance and scalability of our system. In fact, the new protocol suite has no evident limits to the scale of system it could support. In the common case of an application with localized, bursty communication, most multicasts will carry only a small overhead regardless of the size or number of groups used, and a message will be delayed only if it actually arrives out of order.

The paper is structured as follows. Section 2 discusses the types of process groups supported by ISIS and the patterns of group usage and communication that have been observed among current ISIS applications. Section 3 surveys prior work on multicast. Section 4 formalizes the virtually synchronous multicasting problem and the properties that a CBCAST or ABCAST protocol must satisfy. Section 5 introduces our new technique in a single process group; multiple groups are considered in Section 6. Section 7 considers a number of ISIS-specific implementation issues. The paper concludes with a discussion of the performance of our initial implementation, in Section 8.

2. EXPERIENCE WITH ISIS USERS

We begin by reviewing the types of groups and patterns of group usage seen in existing ISIS applications. This material is discussed in more detail by Birman and Cooper [3].

ISIS supports four types of groups, illustrated in Figure 1. The simplest of these is denoted the *peer group*. In a peer group, processes cooperate as equals in order to get a task done. They may manage replicated data, subdivide tasks, monitor one another's status, or otherwise engage in a closely coordinated distributed action. Another common structure is the *client/server group*. Here, a peer group of processes act as servers on behalf of a potentially large set of clients. Clients interact with the servers in a request/reply style, either by picking a favorite server and issuing RPC calls to it, or by multicasting to the whole server group. In the later case, servers will often multicast their replies both to the appropriate client and to one another. A *diffusion group* is a type of client-server group in which the servers multicast messages to the full set of servers and clients. Clients are passive and simply receive messages. Diffusion groups arise in any application that broadcasts information to large a number of sites, for example on a brokerage trading floor. Finally, *hierarchical* group structures arise when larger server groups are needed in a distributed system [10, 14]. Hierarchical groups are tree-structured sets of groups. A *root* group maps the initial connection request to an appropriate subgroup, and the application subsequently interacts only with this subgroup. Data is partitioned among the subgroups, and although a large-group communication mechanism is available, it is rarely needed.

Many ISIS applications use more than one of these structures, employing overlapping groups when mixed functionality is desired. For example, a diffusion group used to disseminate stock quotes would almost always be overlaid by a client/server group through which brokerage programs register their interest in specific stocks. Nonetheless, existing ISIS applications rarely use large numbers of groups. Groups change membership infrequently, and

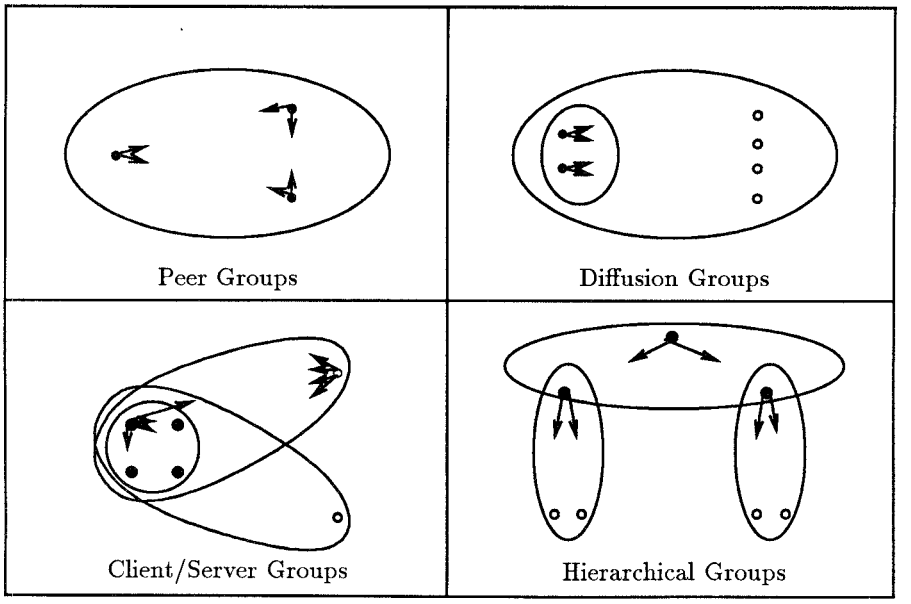

Fig. 1. Types of process groups.

generally contain just enough members for fault-tolerance or load-sharing (e.g., 3–5 processes). On the other hand, the number of *clients* of a client/server or diffusion group may be large (hundreds).

Through studies of ISIS users [3, 4] we have concluded that these patterns are in part artifacts of the way ISIS evolved. In versions of ISIS prior to the one discussed here, groups were fairly heavy-weight entities. Applications obtained acceptable performance only by ensuring that communication to a group was much more frequent than membership changes. Looking to the future, we expect our system to continue supporting these four types of groups. We also expect that groups will remain small, (except for the client set of a client-server or diffusion group). However, as we rebuild ISIS around the protocols described here and move the key modules into lower layers of the operating system, groups and group communication can be expected to get much cheaper. These costs seem to be a dominant factor preventing ISIS users from employing very large numbers of groups, especially in cases where process groups naturally model some sort of application-level data type or object. As a result, we expect that for some applications, groups will substantially outnumber processes. Furthermore, groups may become much more dynamic, because the cost of joining or leaving a group can be substantially reduced using the protocols developed in this paper.

To illustrate these points, we consider some applications that would have these characteristics. A scientific simulation employing an n-dimensional grid might use a process group to represent the neighbors of each grid element. A network information service running on hundreds of sites might

replicate individual data items using small process groups; the result would be a large group containing many smaller data replication domains, perhaps moving data in response to access patterns. Similarly, a process group could be used to implement replicated objects in a modular application that imports many such objects. In each case, the number of process groups would be huge and the overlap between groups extensive.

The desire to support applications like these represents a primary motivation for the research reported here. The earlier ISIS protocols have proven inflexible and difficult to scale; it seems unlikely that they could be used to support the highly dynamic, large-scale applications that now interest us. The protocols reported here respond to these new needs, enabling the exploration of such issues as support for parallel processing, the use of multicast communication hardware, and mechanisms to enforce realtime deadlines and message priorities.

3. PRIOR WORK ON GROUP COMMUNICATION PROTOCOLS

Our communication protocols evolved from a causal message delivery protocol developed by Schiper [25], and are based on work by Fidge [13] and Mattern [19]. In the case of a single process group, the algorithm was influenced by protocols developed by Ladin [16] and Peterson [20]. However, our work generalizes these protocols in the following respects:

—Both of the other multicast protocols address causality only in the context of a single process group. Our solution transparently addresses the case of multiple, overlapping groups. Elsewhere, we argue [4] that a multicast protocol must respect causality to be used asynchronously (without blocking the sender until remote delivery occurs). Asynchronous communication is the key to high performance in group-structured distributed applications and is a central feature of ISIS.

—The ISIS architecture treats client/server groups and diffusion groups as sets of overlayed groups, and optimizes the management of causality information for this case. Both the clients and servers can multicast directly and fault-tolerantly within the subgroups of a client/server group. Peterson's protocols do not support these styles of group use and communication. Ladin's protocol supports client/server interactions, but not diffusion groups, and does not permit clients to multicast directly to server groups.

—Ladin's protocol uses stable storage as part of the fault-tolerance method. Our protocol uses a notion of message *stability* that requires no external storage.

Our CBCAST protocol can be extended to provide a total message delivery ordering, inviting comparison with atomic broadcast (ABCAST) protocols [6, 9, 14, 22, 29]. Again, the extensions supporting multiple groups represent our primary contribution. However, our ABCAST protocol also uses a delivery order consistent with causality thereby permitting it to be used

asynchronously. A delivery ordering might be total without being causal, and indeed, several of the protocols cited would not provide this guarantee.

4. EXECUTION MODEL

We now formalize the model and the problem to be solved.

4.1 Basic System Model

The system is composed of processes $P = \{p_1, p_2, \ldots, p_n\}$ with disjoint memory spaces. Initially, we assume that this set is static and known in advance; later we relax this assumption. Processes fail by crashing detectably (a *fail-stop* assumption); notification is provided by a failure detection mechanism, described below. When multiple processes need to cooperate, e.g., to manage replicated data, subdivide a computation, monitor one another's state, and so forth, they can be structured into *process groups*. The set of such groups is denoted by $G = \{g_1, g_2 \ldots\}$.

Each process group has a name and a set of member processes. Members join and leave dynamically; a failure causes a departure from all groups to which a process belongs. The members of a process group need not be identical, nor is there any limit on the number of groups to which a process may belong. The protocols presented below all assume that processes only multicast to groups that they are members of, and that all multicasts are directed to the full membership of a single group. (We discuss client/server groups in Section 7.)

Our system model is unusual in assuming an external service that implements the process group abstraction. This accurately reflects our current implementation, which obtains group membership management from a preexisting ISIS process-group server. In fact, however, this requirement can be eliminated, as discussed in Section 7.4.

The interface by which a process joins and leaves a process group will not concern us here, but the manner in which the group service communicates membership information to a process is relevant. A *view* of a process group is a list of its members. A *view sequence* for g is a list $view_0(g), view_1(g), \ldots, view_n(g)$, where

(1) $view_0(g) = 0$,

(2) $\forall i$: $view_i(g) \subseteq P$, where P is the set of all processes in the system, and

(3) $view_i(g)$ and $view_{i+1}(g)$ differ by the addition or subtraction of exactly one process.

Processes learn of the failure of other group members only through this view mechanism, never through any sort of direct observation.

We assume that direct communication between processes is always possible; the software implementing this is called the *message transport* layer. Within our protocols, processes always communicate using point-to-point and multicast messages; the latter may be transmitted using multiple point-to-point messages if no more efficient alternative is available. The transport communication primitives must provide lossless, uncorrupted, sequenced

message delivery. The message transport layer is also assumed to intercept and discard messages from a failed process once the failure detection has been made. This guards against the possibility that a process might hang for an extended period (e.g., waiting for a paging store to respond), but then attempt to resume communication with the system. Obviously, transient problems of this sort cannot be distinguished from permanent failures, hence there is little choice but to treat both the same way by forcing the faulty process to run a recovery protocol.

Our protocol architecture permits application builders to define new transport protocols, perhaps to take advantage of special hardware. The implementation described in this paper uses a transport that we built over an unreliable datagram layer.

The execution of a process is a partially ordered sequence of *events*, each corresponding to the execution of an indivisible action. An acyclic event order, denoted $\xrightarrow{p}$, reflects the dependence of events occurring at process p upon one another. The event $send_p(m)$ denotes the transmission of m by process p to a set of one or more destinations, $dests(m)$; the reception of message m by process p is denoted $rcv_p(m)$. We omit the subscript when the process is clear from the context. If $|\, dests(m)\,| > 1$ we will assume that *send* puts messages into all communication channels in a single action that might be interrupted by failure, but not by other *send* or *rcv* actions.

We denote by $rcv_p(view_i(g))$ the event by which a process p belonging to g "learns" of $view_i(g)$.

We distinguish the event of *receiving* a message from the event of *delivery*, since this allows us to model protocols that delay message delivery until some condition is satisfied. The delivery event is denoted $deliver_p(m)$ where $rcv_p(m) \xrightarrow{p} deliver_p(m)$.

When a process belongs to multiple groups, we may need to indicate the group in which a message was sent, received, or delivered. We will do this by extending our notation with a second argument; for example, $deliver_p(m, g)$, will indicate that message m was delivered at process p, and was sent by some other process in group g.

As Lamport [17], we define the potential causality relation for the system, $\rightarrow$, as the transitive closure of the relation defined as follows:

(1) If $\exists p$: $e \xrightarrow{p} e'$, then $e \rightarrow e'$

(2) $\forall m$: $send(m) \rightarrow rcv(m)$

For messages m and m', the notation $m \rightarrow m'$ will be used as a shorthand for $send(m) \rightarrow send(m')$.

Finally, for demonstrating liveness, we assume that any message sent by a process is eventually received unless the sender or destination fails, and that failures are detected and eventually reflected in new group views omitting the failed process.

4.2 Virtual Synchrony Properties Required of Multicast Protocols

Earlier, we stated that ISIS is a *virtually synchronous* programming environment. Intuitively, this means that users can program as if the system

scheduled one distributed event at a time (i.e., group membership changes, multicast, and failures). Were a system to actually behave this way, we would call it *synchronous*; such an environment would greatly simplify the development of distributed algorithms but offers little opportunity to exploit concurrency. The "schedule" used by ISIS is, however, synchronous in appearance only. The ordering requirements of the tools in the ISIS toolkit have been analyzed, and the system actually enforces only the degree of synchronization needed in each case [6]. This results in what we call a *virtually* synchronous execution, in which operations are often performed concurrently and multicasts are often issued asynchronously (without blocking), but algorithms can still be developed and reasoned about using a simple, synchronous model.

Virtual synchrony has two major aspects.

(1) *Address expansion.* It should be possible to use group identifiers as the destination of a multicast. The protocol must expand a group identifier into a destination list and deliver the message such that

   (a) All the recipients are in identical group views when the message arrives.

   (b) The destination list consists of precisely the members of that view.

   The effect of these rules is that the expansion of the destination list and message delivery appear as a single, instantaneous event.

(2) *Delivery atomicity and order.* This involves delivery of messages fault-tolerantly (either all operational destinations eventually receive a message, or, and only if the sender fails, none do). Furthermore, when multiple destinations receive the same message, they observe consistent delivery orders, in one of the two senses detailed below.

Two types of delivery ordering will be of interest here. We define the *causal delivery* ordering for multicast messages m and m' as follows:

$$m \rightarrow m' \Rightarrow \forall p \in dests(m) \cap dests(m'): deliver(m) \xrightarrow{p} deliver(m').$$

CBCAST provides only the causal delivery ordering. If two CBCAST's are concurrent, the protocol places no constraints on their relative delivery ordering at overlapping destinations. ABCAST extends the causal ordering into a total one, by ordering concurrent messages m and m' such that

$$\exists m, m', p \in g: deliver_p(m, g) \xrightarrow{p} deliver_p(m', g) \Rightarrow$$
$$\forall q \in g: deliver_q(m, g) \xrightarrow{q} deliver_q(m', g).$$

Note that this definition of ABCAST only orders messages sent to the *same* group; other definitions are possible. We discuss this further in Section 6.2. Because the ABCAST protocol orders concurrent events, it is more costly than CBCAST, thereby requiring synchronous solutions where the CBCAST protocol admits efficient, asynchronous solutions.

Although one can define other sorts of delivery orderings, our work on ISIS suggests that this is not necessary. The higher levels of the ISIS toolkit are

themselves implemented almost entirely using asynchronous CBCAST [5, 26]. In fact, Schmuck shows [26] that many algorithms specified in terms of ABCAST can be modified to use CBCAST without compromising correctness. Further, he demonstrates that both protocols are complete for a class of delivery orderings. For example, CBCAST can emulate any ordering property that permits message delivery on the first round of communication.

Fault tolerance and message delivery ordering are not independent in our model. A process will not receive further multicasts from a faulty sender after observing it to fail; this requires that multicasts in progress at the time of the failure be *flushed* from the system before the view corresponding to the failure can be delivered to group members. Furthermore, failures will not leave gaps in a causally related sequence of multicasts. That is, if $m \rightarrow m'$ and a process p_i has received m', it need not be concerned that a failure could somehow prevent m from being delivered to any of its destinations (even if the destination of m and m' don't overlap). Failure atomicity alone would not yield either guarantee.

4.3 Vector Time

Our delivery protocol is based on a type of logical clock called a *vector clock*. The vector time protocol maintains sufficient information to represent $\rightarrow$ precisely.

A vector time for a process p_i, denoted $VT(p_i)$, is a vector of length n (where $n = |P|$), indexed by process-id.

(1) When p_i starts execution, $VT(p_i)$ is initialized to zeros.
(2) For each event *send*(m) at p_i, $VT(p_i)[i]$ is incremented by 1.
(3) Each message multicast by process p_i is timestamped with the incremented value of $VT(p_i)$.
(4) When process p_j delivers a message m from p_i containing $VT(m)$, p_j modifies its vector clock in the following manner:

$$\forall k \in 1 \cdots n: VT(p_j)[k] = \max(VT(p_j)[k], VT(m)[k]).$$

That is, the vector timestamp assigned to a message m counts the number of messages, on a per-sender basis, that causally precede m.

Rules for comparing vector timestamps are

(1) $VT_1 \leq VT_2$ iff $\forall i: VT_1[i] \leq VT_2[i]$
(2) $VT_1 < VT_2$ if $VT_1 \leq VT_2$ and $\exists i: VT_1[i] < VT_2[i]$

It can be shown that given messages m and m', $m \rightarrow m'$ iff $VT(m) < VT(m')$: vector timestamps represent causality precisely.

Vector times were proposed in this form by Fidge [13] and Mattern [19]; the latter includes a good survey. Other researchers have also used vector times or similar mechanisms [16, 18, 26, 30]. As noted earlier, our work is an outgrowth of the protocol presented in [25], which uses vector times as the

basis for a protocol that delivers point-to-point messages in an order consistent with causality.

5. THE CBCAST AND ABCAST PROTOCOL

This section presents our new CBCAST and ABCAST protocols. We initially consider the case of a single process group with fixed membership; multiple group issues are addressed in the next section. This section first introduces the causal delivery protocol, then extends it to a totally ordered ABCAST protocol, and finally considers view changes.

5.1 CBCAST Protocol

Suppose that a set of processes P communicate using only broadcasts to the full set of processes in the system; that is, $\forall m$: $dests(m) = P$. We now develop a *delivery protocol* by which each process p receives messages sent to it, delays them if necessary, and then delivers them in an order consistent with causality:

$$m \rightarrow m' \Rightarrow \forall p: deliver_p(m) \xrightarrow{p} deliver_p(m').$$

Our solution is derived using vector timestamps. The basic idea is to label each message with a timestamp, $VT(m)[k]$, indicating precisely how many multicasts by process p_k precede m. A recipient of m will delay m until $VT(m)[k]$ messages have been delivered from p_k. Since $\rightarrow$ is an acyclic order accurately represented by the vector time, the resulting delivery order is causal and deadlock free.

The protocol is as follows:

(1) Before sending m, process p_i increments $VT(p_i)[i]$ and timestamps m.

(2) On reception of message m sent by p_i and timestamped with $VT(m)$, process $p_j \neq p_i$ delays delivery of m until:

$$\forall k: 1 \cdots n \begin{cases} VT(m)[k] = VT(p_j)[k] + 1 & \text{if } k = i \\ VT(m)[k] \leq VT(p_j)[k] & \text{otherwise} \end{cases}$$

Process p_j need not delay messages received from itself. Delayed messages are maintained on a queue, the CBCAST *delay queue*. This queue is sorted by vector time, with concurrent messages ordered by time of receipt (however, the queue order will not be used until later in the paper).

(3) When a message m is delivered, $VT(p_j)$ is updated in accordance with the vector time protocol from Section 4.3.

Step 2 is the key to the protocol. This guarantees that any message m' transmitted causally before m (and hence with $VT(m') < VT(m)$) will be delivered at p_j before m is delivered. An example in which this rule is used to delay delivery of a message appears in Figure 2.

We prove the correctness of the protocol in two stages. We first show that causality is never violated (safety) and then we demonstrate that the protocol never delays a message indefinitely (liveness).

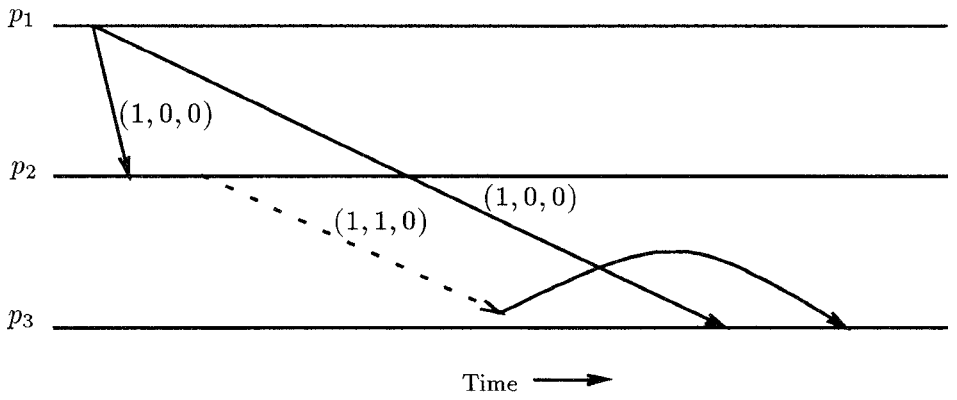

Fig. 2. Using the *VT* rule to delay message delivery.

Safety. Consider the actions of a process p_j that receives two messages m_1 and m_2 such that $m_1 \rightarrow m_2$.

Case 1. m_1 and m_2 are both transmitted by the same process p_i. Recall that we assumed a lossless, live communication system, hence p_j eventually receives both m_1 and m_2. By construction, $VT(m_1) < VT(m_2)$, hence under step 2, m_2 can only be delivered after m_1 has been delivered.

Case 2. m_1 and m_2 are transmitted by two distinct processes p_i and $p_{i'}$. We will show by induction on the messages received by process p_j that m_2 cannot be delivered before m_1. Assume that m_1 has not been delivered and that p_j has received k messages.

Observe first that $m_1 \rightarrow m_2$, hence $VT(m_1) < VT(m_2)$ (basic property of vector times). In particular, if we consider the field corresponding to process p_i, the sender of m_1, we have

$$VT(m_1)[i] \leq VT(m_2)[i]. \tag{1}$$

Base case. The first message delivered by p_j cannot be m_2. Recall that if no messages have been delivered to p_j, then $VT(p_j)[i] = 0$. However, $VT(m_1)[i] > 0$ (because m_1 is sent by p_i), hence $VT(m_2)[i] > 0$. By application of step 2 of the protocol, m_2 cannot be delivered by p_j.

Inductive step. Suppose p_j has received k messages, none of which is a message m such that $m_1 \rightarrow m$. If m_1 has not yet been delivered, then

$$VT(p_j)[i] < VT(m_1)[i]. \tag{2}$$

This follows because the only way to assign a value to $VT(p_j)[i]$ greater than $VT(m_1)[i]$ is to deliver a message from p_i that was sent subsequent to m_1, and such a message would be causally dependent on m_1. From relations 1 and 2 it follows that

$$VT(p_j)[i] < VT(m_2)[i].$$

By application of step 2 of the protocol, the $k + 1$st message delivered by p_j cannot be m_2.

Liveness. Suppose there exists a broadcast message m sent by process p_i that can never be delivered to process p_j. Step 2 implies that either:

$$\exists k: 1 \cdots n \begin{cases} VT(m)[k] \neq VT(p_j)[k] + 1 & \text{for } k = i, \text{ or} \\ VT(m)[k] > VT(p_j)[k] & k \neq i \end{cases}$$

and that m was not transmitted by process p_j. We consider these cases in turn.

- $VT(m)[i] \neq VT(p_j)[i] + 1$; that is, m is not the *next message* to be delivered from p_i to p_j. Notice that only a finite number of messages can precede m. Since all messages are multicast to all processes and channels are lossless and sequenced, it follows that there must be some message m' sent by p_i that p_j received previously, has not yet delivered, and that *is* the next message from p_i, i.e., $VT(m')[i] = VT(p_j)[i] + 1$. If m' is also delayed, it must be under the other case.
- $\exists k \neq i: VT(m)[k] > VT(p_j)[k]$. Let $n = VT(m)[k]$. The nth transmission of process p_k, must be some message $m' \rightarrow m$ that has either not been received at p_j, or was received and is delayed. Under the hypothesis that all messages are sent to all processes, m' was already multicast to p_j. Since the communication system eventually delivers all messages, we may assume that m' has been received by p_j. The same reasoning that was applied to m can now be applied to m'. The number of messages that must be delivered before m is finite and $>$ is acyclic, hence this leads to a contradiction.

5.2 Causal ABCAST Protocol

The CBCAST protocol is readily extended into a causal, totally ordered, ABCAST protocol. We should note that it is unusual for an ABCAST protocol to guarantee that the total order used conforms with causality. For example, say that a process p asynchronously transmits message m using ABCAST, then sends message m' using CBCAST, and that some recipient of m' now sends m'' using ABCAST. Here we have $m \rightarrow m' \rightarrow m''$, but m and m'' are transmitted by different processes. Many ABCAST protocols would use an arbitrary ordering in this case; our solution will always deliver m before m''. This property is actually quite important: without it, few algorithms could safely use ABCAST asynchronously, and the delays introduced by blocking until the protocol has committed its delivery ordering could be significant. This issue is discussed further by Birman et al. [4].

Our solution is based on the ISIS replicated data update protocol described by Birman and Joseph [7] and the ABCAST protocol developed by Birman and Joseph [7] and Schmuck [26]. Associated with each view $view_i(g)$ of a process group g will be a *token holder* process, $token(g) \in view_i(g)$. We also assume that each message m is uniquely identified by $uid(m)$.

To ABCAST m, a process holding the token uses CBCAST to transmit m in the normal manner. If the sender is not holding the token, the ABCAST is

kernel, so that the protocol can be moved closer to the hardware communication device. For example, both Mach and Chorus permit application developers to move modules of code into the network communication component of the kernel. In our case, this would yield a significant speedup. The other obvious speedup would result from the use of hardware multicast, an idea that we are now exploring experimentally.

9. CONCLUSIONS

We have presented a protocol efficiently implementing a reliable, causally ordered multicast primitive. The protocol is easily extended into a totally ordered "atomic" multicast primitive and has been implemented as part of Versions 2.1 and 3.0 of the ISIS Toolkit. Our protocol offers an inexpensive way to achieve the benefits of *virtual synchrony*. It is fast and scales well; in fact, there is no evident limit to the size of network in which it could be used. Even in applications with large numbers of overlapping groups, the overhead on a multicast is typically small and in systems with bursty communication, most multicasts can be sent with no overhead other than that needed to implement reliable, FIFO interprocess channels.With appropriate device drivers or multicast communication hardware, the basic protocol will operate safely in a completely asynchronous, streaming fashion, never blocking a message or its sender unless out-of-order reception genuinely occurs. Our conclusion is that systems such as ISIS can achieve performance competitive with the best existing multicast facilities—a finding contradicting the widespread concern that fault-tolerance may be unacceptably costly.

ACKNOWLEDGMENTS

The authors are grateful to Maureen Robinson for the preparation of many of the figures. We also thank Keith Marzullo, who suggested we search for the necessary and sufficient conditions of Section 6.6 and made many other useful comments. Gil Neiger, Tushar Chandra, Robert Cooper, Barry Gleeson, Shivakant Misra, Aleta Ricciardi, Mark Wood, and Guerney Hunt made numerous suggestions concerning the protocols and presentation, which we greatly appreciate.

REFERENCES

1. ACCETTA, M., BARON, R , GOLUB, D., RASHID, R., TEVANIAN, A., AND YOUNG, M. Mach: A new kernel foundation for UNIX development. Tech. Rep., School of Computer Science, Carnegie Mellon Univ., Pittsburgh, PA, Aug. 1986. Also in *Proceedings of the Summer 1986 USENIX Conference* (July 1986), pp. 93–112.
2. ACM SIGOPS. *Proceedings of the Ninth ACM Symposium on Operating Systems Principles* (Bretton Woods, N.H., Oct. 1983).
3. BIRMAN, K., AND COOPER, R. The ISIS project: Real experience with a fault tolerant programming system. *European SIGOPS Workshop*, Sept. 1990. To appear in *Operating Syst. Rev.* April 1991; also available as Tech. Rep. TR90-1138, Cornell Univ , Computer Science Dept.
4. BIRMAN, K. A., COOPER, R., AND GLEESON, B. Programming with process groups: Group

and multicast semantics. Tech. Rep. TR91-1185. Cornell Univ., Computer Science Dept., Feb. 1991.

5. BIRMAN, K., AND JOSEPH, T. Exploiting replication in distributed systems. In *Distributed Systems*, Sape Mullender, Ed., ACM Press, New York, 1989, pp. 319–368,
6. BIRMAN, K. P., AND JOSEPH, T. A. Exploiting virtual synchrony in distributed systems. In *Proceeding of the Eleventh ACM Symposium on Operating Systems Principles* (Austin, Tex., Nov. 1987). ACM SIGOPS, New York, 1987, pp. 123–138.
7. BIRMAN, K. P., AND JOSEPH, T. A. Reliable communication in the presence of failures. *ACM Trans. Comput. Syst. 5*, 1 (Feb. 1987), 47–76.
8. BIRMAN, K. P., JOSEPH, T. A., KANE, K., AND SCHMUCK, F. *ISIS – A Distributed Programming Environment User's Guide and Reference Manual, first edition.* Dept. of Computer Science, Cornell Univ., March 1988.
9. CHANG, J., AND MAXEMCHUK, N. Reliable broadcast protocols. *ACM Trans. Comput. Syst. 2*, 3 (Aug. 1984), 251–273.
10. COOPER, R., AND BIRMAN, K. Supporting large scale applications on networks of workstations. In *Proceedings of 2nd Workshop on Workstation Operating Systems* (Washington D.C., Sept. 1989), IEEE, Computer Society Press. Order 2003, pp. 25–28.
11. CRISTIAN, F., AGHILI, H., STRONG, H. R., AND DOLEV, D. Atomic broadcast: From simple message diffusion to Byzantine agreement. Tech. Rep. RJ5244, IBM Research Laboratory, San Jose, Calif., July 1986. An earlier version appeared in *Proceedings of the International Symposium on Fault-Tolerant Computing*, 1985.
12. DENNING, P. Working sets past and present. *IEEE Trans. Softw. Eng. SE-6*, 1 (Jan. 1980), 64–84.
13. FIDGE, C. Timestamps in message-passing systems that preserve the partial ordering. In *Proceedings of the 11th Australian Computer Science Conference*, 1988, pp. 56-66.
14. GARCIA-MOLINA, H., AND SPAUSTER, A. Message ordering in a multicast environment. In *Proceedings 9th International Conference on Distributed Computing Systems* (June 1989), IEEE, New York, 1989, pp. 354–361.
15. KAASHOEK, M F., TANENBAUM, A. S., HUMMEL, S. F., AND BAL, H. E. An efficient reliable broadcast protocol. *Operating Syst. Rev. 23*, 4 (Oct. 1989), 5–19.
16. LADIN, R., LISKOV, B., AND SHRIRA, L. Lazy replication: Exploiting the semantics of distributed services. In *Proceedings of the Tenth ACM Symposium on Principles of Distributed Computing* (Quebec City, Quebec, Aug. 1990). ACM, New York, 1990, pp. 43–58.
17. LAMPORT, L. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM, 21*, 7 (July 1978), 558–565.
18. MARZULLO, K. Maintaining the time in a distributed system. PhD thesis, Dept. of Electrical Engineering, Stanford Univ., June 1984.
19. MATTERN, F. Time and global states in distributed systems. In *Proceedings of the International Workshop on Parallel and Distributed Algorithms*. North-Holland, Amsterdam, 1989.
20. PETERSON, L. L., BUCHOLZ, N. C., AND SCHLICHTING, R. Preserving and using context information in interprocess communication. *ACM Trans. Comput. Syst. 7*, 3 (Aug. 1989), 217–246.
21. PETERSON, L. L., HUTCHINSON, N., O'MALLEY, S., AND ABBOTT, M. Rpc in the x-Kernel: Evaluating new design techniques. In *Proceedings of the Twelfth ACM Symposium on Operating Systems Principles* (Litchfield Park, Ariz., Nov. 1989). ACM, New York, 1989, pp. 91–101.
22. PITELLI, F., AND GARCIA-MOLENA, H. Data processing with triple modular redundancy. Tech. Rep. TR-002-85, Princeton Univ., June 1985.
23. POWELL, M. AND PRESOTTO, D. L. Publishing: A reliable broadcast communication mechanism. In *Proceedings of the Ninth ACM Symposium on Operating System Principles* pp. 100–109. Proceedings published as *Operating Systems Review 17*, 5.
24. RICCIARDI, A., AND BIRMAN, K. P. Using process groups to implement failure detection in asynchronous environments. Tech. Rep. TR91-1188, Computer Science Dept., Cornell Univ., Feb. 1991.
25. SCHIPER, A., EGGLI, J., AND SANDOZ, A. A new algorithm to implement causal ordering. In

Proceedings of the 3rd International Workshop on Distributed Algorithms, Lecture Notes on Computer Science 392, Springer-Verlag, New York, 1989, pp. 219–232.

26 SCHMUCK, F. The use of efficient broadcast primitives in asynchronous distributed systems. PhD thesis, Cornell Univ., 1988.

27. SRIKANTH, T. K., AND TOUEG, S. Optimal clock synchronization. *J. ACM 34*, 3 (July 1987), 626–645.

28 STEPHENSON, P. Fast causal multicast. PhD thesis, Cornell Univ., Feb. 1991.

29. VERÍSSIMO, P., RODRIGUES, L., AND BAPTISTA, M. Amp: A highly parallel atomic multicast protocol. In *Proceedings of the Symposium on Communications Architectures & Protocols* (Austin, Tex., Sept 1989). ACM, New York, 1989, 83–93.

30. WALKER, B., POPEK, G., ENGLISH, R., KLINE, C., AND THIEL, G. The LOCUS distributed operating system. In *Proceedings of the Ninth ACM Symposium on Operating Systems Principles* (Bretton Woods, N H , Oct 1983), pp. 49–70

Received April 1990; revised April 1991; accepted May 1991