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

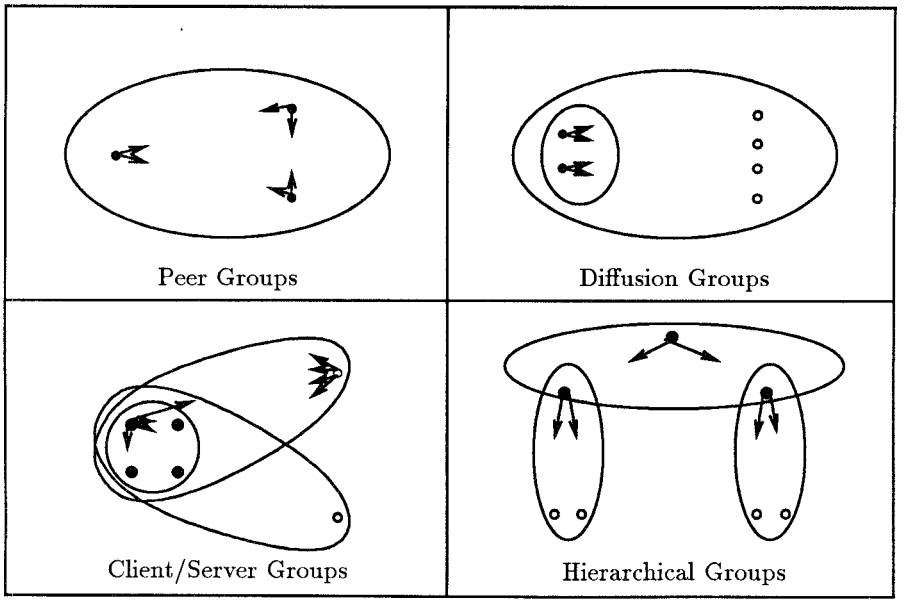

Fig. 1. Types of process groups.

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).

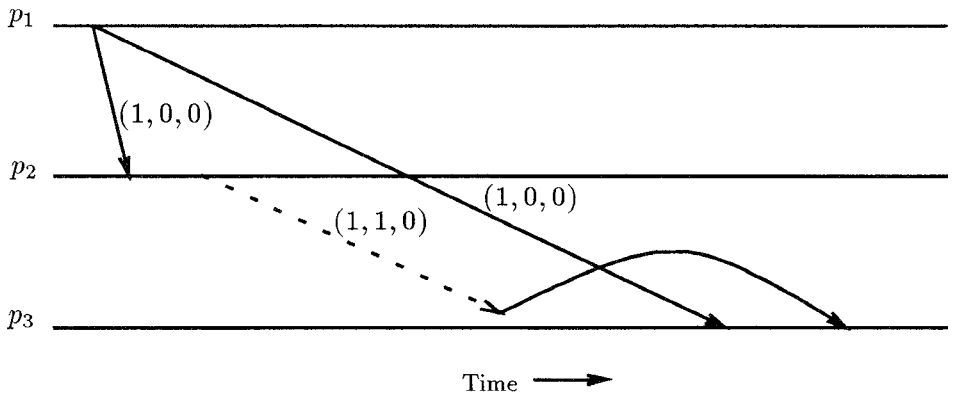

Fig. 2. Using the *VT* rule to delay message delivery.

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

done in stages:

(1) The sender CBCAST's *m* but marks it as undeliverable.[1] Processes other than the token holder (including the sender) that receive this message place *m* on the CBCAST delay queue in the usual manner, but do not remove *m* from the queue for delivery even after all messages that precede it causally have been delivered. It follows that a typical process may have some number of delayed ABCAST messages at the front of its CBCAST delay queue. This prevents the delivery of causally subsequent CBCAST messages, because the vector time is not updated until delivery occurs. On the other hand, a CBCAST that precedes or is concurrent with one of these undeliverable ABCAST messages will not be delayed.

(2) The token holder treats incoming ABCAST messages as it would treat incoming CBCAST messages, delivering them in the normal manner. However, it also makes note of the *uid* of each such ABCAST.

(3) After the process holding the token has delivered one or more ABCAST messages, it uses CBCAST to send a **sets-order** message giving a list of one or more messages, identified by *uid*, and ordered in the delivery order that arose in step 2. If desired, this CBCAST may be delayed so as to "batch" such transmissions, but it must be sent before (or piggybacked upon) any subsequent ABCAST or CBCAST by the token holder. If desired, a new token holder may be specified in this message.

(4) On receipt of a **sets-order** message, a process places it on the CBCAST delay queue in the normal manner. Eventually, all the ABCAST messages referred to in the **sets-order** message will be received, and all the CBCAST messages that precede the **sets-order** will have been delivered (liveness of CBCAST).
Recall that $\rightarrow$ places a partial order on the messages in the delay queue. Our protocol now reorders concurrent ABCAST messages by placing them in the order given by the **sets-order** message, and marks them as deliverable.

(5) Deliverable ABCAST messages may be delivered off the front of the queue.

Step 4 is the key one in the protocol. This step causes all participants to deliver ABCAST messages in the order that the token holder used. This order will be consistent with causality because the token holder itself treated these ABCAST messages as if they were CBCAST's.

The cost of doing an ABCAST depends on the locations where multicasts originate and the frequency with which the token is moved. If multicasts tend to originate at the same process repeatedly, then once the token is moved to that site, the cost is one CBCAST per ABCAST. If they originate

[1] It might appear cheaper to forward such a message directly to the token holder However, for a moderately large message such a solution would double the IO done by the token holder, creating a likely bottleneck, while reducing the IO load on other destinations only to a minor degree.

randomly and the token is not moved, the cost is $1 + 1/k$ CBCAST's per ABCAST, where we assume that one **sets-order** message is sent for ordering purposes after k ABCAST's.

5.3 Multicast Stability

The knowledge that a multicast has reached all its destinations will be useful below. Accordingly, we will say that a multicast m is *k-stable* if it is known that the multicast has been received at k destinations. When $k = |dests(m)|$ we will say that m is (*fully*) *stable.*

Recall that our model assumes a reliable transport layer. Since messages can be lost in transport for a variety of reasons (buffer overflows, noise on communication lines, etc.), such a layer normally uses some sort of positive or negative acknowledgement scheme to confirm delivery. Our liveness assumption is strong enough to imply that any multicast m will eventually become stable, if the sender does not fail. It is thus reasonable to assume that this information is available to the process that sent a CBCAST message.

To propagate stability information among the members of a group, we will introduce the following convention. Each process p_i maintains a *stability sequence number*, *stable* (p_i). This number will be the largest integer n such that all multicasts by sender p_i having $VT(p_i)[i] \leq n$ are stable.

When sending a multicast m, process p_i piggybacks its current value of *stable*(p_i); recipients make note of these incoming values. If *stable*(p_i) changes and p_i has no outgoing messages then, when necessary (see below), p_i can send a **stability** message containing only *stable*(p_i).

5.4 VT Compression

It is not always necessary to transmit the full vector timestamp on each message.

LEMMA 1. *Say that process p_i sends a multicast m. Then $VT(m)$ need only carry vector timestamp fields that have changed since the last multicast by p_i.*

PROOF. Consider two multicasts m and m' such that $m \rightarrow m'$. If p_i is the sender of m and p_j is the sender of m', then there are two cases. If $i = j$ then $VT(m)[i] < VT(m')[i]$, hence (step 2, Section 5.1) m' cannot be delivered until after m is delivered. Now, if $i \neq j$ but m' carries the field for p_i, then $VT(m)[i] \leq VT(m')[i]$, and again, m' will be delayed under step 2 until after m is delivered. But, if this field is omitted, there must be some earlier message m'', also multicast by p_j, that did carry the field. Then m will be delivered before m'' and, under the first case, m'' will be delivered before m'. □

Compression may not always be advantageous: the data needed to indicate which fields have been transmitted may actually increase the size of the VT representation. However, in applications characterized by relatively localized, bursty communication, compression could substantially reduce the size of the timestamp. In fact, if a single process sends a series of messages, and receives no messages between the sends, then the VT timestamp on all

messages but the first will just contain one field. Moreover, in this case, the value of the field can be inferred from the FIFO property of the channels, so such messages need not contain *any* timestamp. We will make further use of this idea below.

5.5 Delivery Atomicity and Group Membership Changes

We now consider the implementation of atomicity and how group membership changes impact the above protocols. Such events raise several issues that are addressed in turn:

(1) Virtually synchronous addressing.
(2) Reinitializing VT timestamps.
(3) Delivery atomicity when failures occur.
(4) Handling delayed ABCAST messages when the token holder fails without sending a **sets-order** message.

Virtually synchronous addressing. To achieve virtually synchronous addressing when group membership changes while multicasts are active, we introduce the notion of *flushing* the communication in a process group. Initially, we will assume that processes do not fail or leave the group (we treat these cases in a subsequent subsection). Consider a process group in $view_i$. Say that $view_{i+1}$ now becomes defined. We can flush communication by having all the processes in $view_{i+1}$ send a message **"flush i + 1"**, to all other processes in this view. After sending such messages and before receiving such a flush message from all members of $view_{i+1}$ a process will accept and deliver messages but will not initiate new multicasts. Because communication is FIFO, if process p has received a flush message from all processes in $view_{i+1}$, it will first have received all messages that were sent by members of $view_i$. In the absence of failures, this establishes that multicasts will be virtually synchronous in the sense of Section 4.

A disadvantage of this flush protocol is that it sends n^2 messages. Fortunately, the solution is readily modified into one that uses a linear number of messages. Still deferring the issue of failures, say that we designate one member of $view_{i+1}$, p_c, as the *flush coordinator*. Any deterministic rule can be used for this purpose. A process p_i other than the coordinator flushes by first waiting until all multicasts that it sent have stabilized, and then sending a $view_{i+1}$ flush message to the coordinator.[2] The coordinator, p_c, waits until flush messages have been received from all other members of $view_{i+1}$. It then multicasts its own flush message to the members of $view_{i+1}$ (it need not wait for its own multicasts to stabilize). Reception of the flush multicast from p_c provides the same guarantees as did the original solution. If stability is achieved rapidly, as would normally be the case, the cost of the new protocol is much lower: $2n$ messages.

[2] Actually, it need not wait until the coordinator has received its active multicasts, since channels are FIFO.

Reinitializing VT fields. After executing the flush protocol, all processes can reset the fields of VT to zero. This is clearly useful. Accordingly, we will now assume that the vector timestamp sent in a message, $VT(m)$, includes the index i of $view_i$ in which m was transmitted. A vector timestamp carrying an expired index may be ignored, since the flush protocol used to switch views will have forced the delivery of all multicasts that could have been pending in the prior view.

Delivery atomicity and virtual synchrony when failures occur. We now consider the case where some process *fails* during an execution. We discuss the issues raised when processes voluntarily leave a group at the end of this subsection.

Failures introduce two basic problems:

(1) A failure could disrupt the transmission of a multicast. Thus, if p_j has received a multicast m message from p_i, and has not learned of the stability of that multicast, some of the other destinations of m may not have received a copy.

(2) We can no longer assume that all processes will respect the flush protocol, since the failure of a process p_i could prevent it from sending flush messages for some view, even if that view reports p_i as still operational. On the other hand, we also know that a view showing the failure of p_i will eventually be received.

To solve the first problem, we will have all processes retain copies of the messages they receive. If p_j detects the failure of p_i, it will forward a copy of any unstable multicasts it has received from p_i to other members of the group. All processes identify and reject duplicates. However, the second problem could now prevent the protocol from being respected, leaving the first problem unsolved, as illustrated in Figure 3. This shows that the two problems are closely related and have to be addressed in a coordinated way.

The solution to this atomicity and virtual synchrony problem is most readily understood in terms of the original n^2 message flush protocol. If we are running that protocol, it suffices to delay the installation of $view_{i+1}$ until, for some $k \geq 1$, flush messages for $view_{i+k}$ have been received from all processes in $view_{i+k} \cap view_{i+1}$. Notice that a process may be running the flush protocol for $view_{i+k}$ despite not yet having installed $view_{i+1}$.

More formally, the algorithm executed by a process p is as follows.

(1) On receiving $view_{i+k}$, p increments a local variable *inhibit_sends*; while the counter remains greater than 0, new messages will not be initiated. p forwards a copy of any unstable message m that was sent in $view_j (j < i + k)$ to all processes in $view_j \cap view_{i+k}$[3], and then marks m as stable. Lastly, p sends **"flush i + k"** to each member of $view_{i+k}$.

(2) On receiving a copy of a message m, p examines the index of the view in which m was sent. If p most recently installed $view_i$ and m was sent in

[3] In practice, it may be easier and faster to multicast all the unstable messages for $view_j$ to all processes in $view_{i+k}$. Processes *not* in $view_j \cap view_{i+k}$ will discard this multicast.

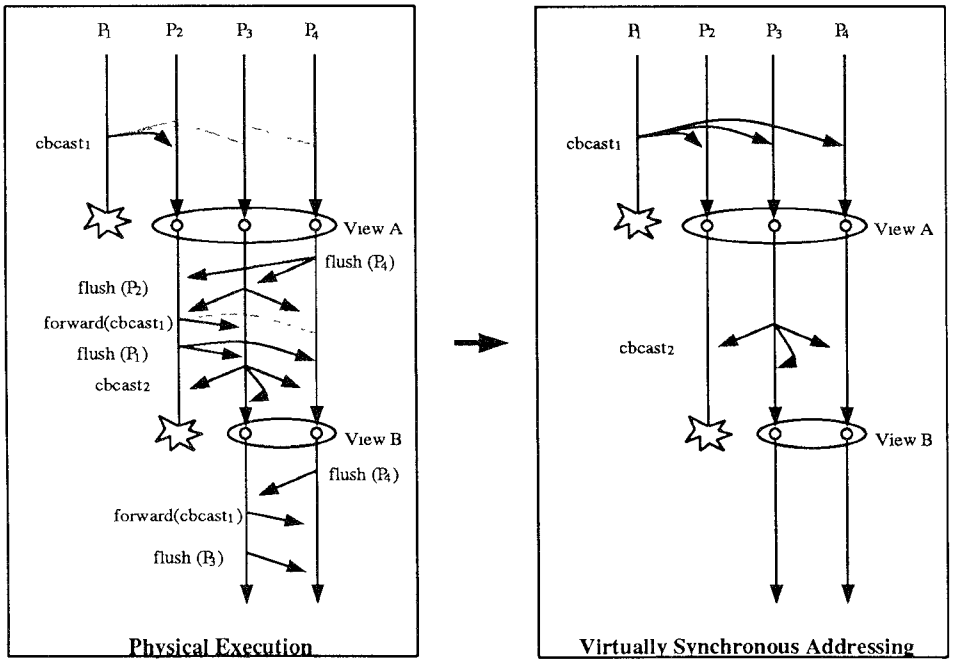

Fig. 3. Group flush algorithm.

$view(m) < i$, p ignores m as a duplicate. If m was sent in $view_i$, p applies the normal delivery rule, identifying and discarding duplicates in the obvious manner. If m was sent in $view(m) > i$, p saves m until $view(m)$ has been installed.

(3) On receiving **"flush i + k"** messages from all processes in $view_{i+1} \cap view_{i+k}$ (for any $k \geq 1$), p installs $view_{i+1}$ by delivering it to the application layer. It decrements the *inhibit_sends* counter, and, if the counter is now zero, permits new messages to be initiated.
Any message m that was sent in $view_i$ and has not yet been delivered may be discarded. This can only occur if the sender of m has failed, and has previously received and delivered a message m' ($m' \rightarrow m$) that has now been lost. In such a situation, m is an orphan of a system execution that was "erased" by multiple failures.[4]

(4) A message can be discarded as soon as it has been delivered locally and has become stable. Notice that a message becomes stable after having been forwarded at most once (in step 1).

[4] Notice that in our model, even if process p accepts and delivers a message m under this protocol, the failure of p could lead to a situation in which m is not delivered to its other destinations. Our definition of delivery atomicity could be changed to exclude such executions, but atomicity could then only be achieved using a much more costly 2-phase delivery protocol.

(5) A process p that sent a message m in $view_i$ will now consider m to have stabilized if delivery confirmation has been obtained from all processes in $view_i \cap view_{i+k}$, for any value of k, or if $view_{i+1}$ has been installed.

LEMMA 2. *The flush algorithm is safe and live.*

PROOF. A participant in the protocol, p, delays installation of $view_{i+1}$ until it has received flush messages sent in $view_{i+k}$ ($k \geq 1$) from all members of $view_{i+1}$ that are still operational in $view_{i+k}$. These are the only processes that could have a copy of an unstable message sent in $view_i$. This follows because messages from failed processes are discarded, and because messages are only forwarded to processes that were alive in the view in which they were originally sent. Because participants forward unstable messages before sending the flush, and channels are FIFO, p will have received all messages sent in $view_i$ before installing $view_{i+1}$. It follows that the protocol is safe. Liveness follows because the protocol to install $view_{i+1}$ can only be delayed by the failure of a member of $view_{i+1}$. Since $view_{i+1}$ has a finite number of members, any delay will be finite. □

This protocol can be modified to send a linear number of messages in each view, using the same coordinator-based scheme as was proposed earlier. Instead of sending **"flush i + k"** messages to all processes in $view_{i+1} \cap view_{i+k}$, a process forwards copies of unstable messages to the coordinator, followed by a **"flush i + k"** message. (A message m originally sent in $view_i$ by a process p, which failed before m became stable, is regarded as unstable by a surviving process $q \in VIEW_i$ until q installs $view_{i+1}$.) The coordinator, in turn, forwards these messages back to the other processes in the view, and then, after receiving **"flush i + k"** messages from all the members in $view_{i+k}$, multicasts its own flush message. $view_{i+k}$ can be installed when the flush message from the coordinator is received.

We now consider the case where failures occur during the execution of the above protocol. The coordinator should abort its protocol if it is waiting for a flush message from some process p, and $view_{i+k+l}$ becomes defined, showing that p has failed. In this case, the protocol for $view_{i+k+l}$ will subsume the one for $view_{i+k}$. Similarly, if the coordinator itself fails, a future flush run by some other coordinator will subsume the interrupted protocol. A successfully completed flush permits installation of all prior views.

For liveness, it now becomes necessary to avoid infinitely delaying the installation of a view in the event of an extended process join/failure sequence. We therefore modify the protocol to have participants inform the coordinator of their current view. A coordinator that has gathered flush messages for $view_{i+k}$ from all the processes in $view_{i+1} \cap view_{i+k}$ can send a $view_{i+1}$ flush message to any process still in $view_i$, even if it has not yet received all the flush messages needed to terminate the protocol for $view_{i+k}$. With this change, the protocol is live.

As illustrated in Figure 3, this protocol converts an execution with nonatomic multicasts into one in which all multicasts are atomically delivered, at linear cost.

A slight change to the protocol is needed in the case where a process leaves a group for reasons other than failure. ISIS supports this possibility, and it triggers a view change similar to the one for a failure, but in which the departing process is reported as having left voluntarily. In this case, the flush algorithm must include the departing process, which is treated as a member of the group until it either fails or the algorithm terminates. This ensures that even if the departing member is the only process to have received some multicast interrupted by a failure, delivery atomicity will still be preserved.

ABCAST ordering when the token holder fails. The atomicity mechanism of the preceding subsection requires a small modification of the ABCAST protocol. Consider an ABCAST that is sent in $view_i$ and for which the token holder fails before sending the **sets-order** message.

After completion of the flush protocol for $view_i$, the ABCAST message will be on every delay queue, but not delivered anywhere. Moreover, any **sets-order** messages that were initiated before the failure will have been delivered everywhere, hence the set of undelivered ABCAST messages is the same at all processes. These messages must be delivered before $view_{i+1}$ can be installed.

Notice that the delay queue is partially ordered by $\rightarrow$. We can solve our problem by ordering any concurrent ABCAST messages within this set using any well-known, deterministic rule. For example, they can be sorted by *uid*. The resulting total order on ABCAST messages will be the same at all processes and consistent with causality.

6. EXTENSIONS TO THE BASIC PROTOCOL

Neither of the protocols in Section 5 is suitable for use in a setting with multiple process groups. We first introduce the modifications needed to extend CBCAST to a multigroup setting. We then briefly examine the problem of ABCAST in this setting.

The CBCAST solution we arrive at initially could be costly in systems with very large numbers of process groups or groups that change dynamically. This has *not* been a problem in the current ISIS system because current applications use comparatively few process groups, and processes tend to multicast for extended periods in the same group. However, these characteristics will not necessarily hold in future ISIS applications. Accordingly, the second part of the section explores additional extensions of the protocol that would permit its use in settings with very large numbers of very dynamic process groups. The resulting protocol is interesting because it exploits properties of what we call the *communication structure* of the system.

6.1 Extension of CBCAST to Multiple Groups

The first extension to the protocol is concerned with systems composed of multiple process groups. We will continue to assume that a given multicast is sent to a single group destination, but it may now be the case that a process belongs to several groups and is free to multicast in any of them.

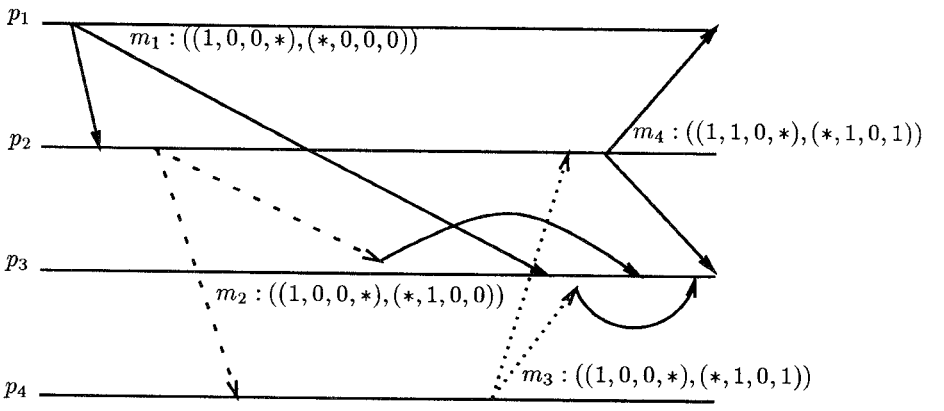

Fig. 4. Messages sent within process groups. $G_1 = \{p_1, p_2, p_3\}$ and $G_2 = \{p_2, p_3, p_4\}$.

Suppose that process p_i belongs to groups g_a and g_b, and multicasts within both groups. Multicasts sent by p_i to g_a must be distinguished from those to g_b. If not, a process belonging to g_b and not to g_a that receives a message with $VT(m)[i] = k$ will have no way to determine how many of these k messages were actually sent to g_b and should, therefore, precede m causally.

This leads us to extend the single VT clock to multiple VT clocks. We will use the notation VT_a to denote the logical clock associated with group g_a; $VT_a[i]$ thus counts[5] multicasts by process p_i to group g_a. The stability sequence number, $stable(p_i)$ should be similarly qualified: $stable_a(p_i)$. Processes maintain VT clocks for each group in the system, and attach all the VT clocks to every message that they multicast.

The next change is to step 2 of the protocol (Section 5.1). Suppose that process p_j receives a message m sent in group g_a with sender p_i, and that p_j also belongs to groups $\{g_1, \ldots, g_n\} \equiv G_j$. Step 2 can be replaced by the following rule:

2′. On reception of message m from $p_i \neq p_j$, sent in g_a, process p_j delays m until

2.1′ $VT_a(m)[i] = VT_a(p_j)[i] + 1$, and

2.2′ $\forall k: (p_k \in g_a \wedge k \neq i)$: $VT_a(m)[k] \leq VT(p_j)[k]$, and

2.3′ $\forall g: (g \in G_j)$: $VT_g(m) \leq VT_g(p_j)$.

This is just the original protocol modified to iterate over the set of groups to which a process belongs. As in the original protocol, p_j does not delay messages received from itself.

Figure 4 illustrates the application of this rule in an example with four processes identified as $p_1 \ldots p_4$. Processes p_1, p_2 and p_3 belong to group G_1,

[5] Clearly, if p_i is not a member of g_a, then $VT_a[i] = 0$, allowing a sparse representation of the timestamp. For clarity, our figures will continue to depict each timestamp VT_g as a vector of length n, with a special entry $*$ for each process that is not a member of g_a.

and processes p_2, p_3 and p_4 to group G_2. Notice that m_2 and m_3 are delayed at p_3, because it is a member of G_1 and must receive m_1 first. However, m_2 is not delayed at p_4, because p_4 is not a member of G_1. And m_3 is not delayed at p_2, because p_2 has already received m_1 and it was the sender of m_2.

The proof of Section 5 adapts without difficulty to this new situation; we omit the nearly identical argument. One can understand the modified protocol in intuitive terms. By ignoring the vector timestamps for certain groups in step 2.3′, we are asserting that there is no need to be concerned that any undelivered message from these groups could causally precede m. But, the ignored entries correspond to groups to which p_j does not belong. Since all communication is done within groups, these entries are irrelevant to p_j.

6.2 Multiple-Group ABCAST

When run over this extended CBCAST protocol, our ABCAST solution will continue to provide a total, causal delivery ordering within any single process group. However, it will not order multicasts to different groups *even if those groups have members in common.* In a previous article [4] we examine the need for a global ABCAST ordering property and suggest that it may be of little practical importance, since the single-group protocol satisfies the requirements of all existing ISIS applications of which we know. We have extended our ABCAST solution to a solution that provides a global total ordering; the resulting protocol, in any single group g, has a cost proportional to the number of other groups that overlap with g. Details are presented by Stephenson [28]. This extended, global-ordering ABCAST protocol could be implemented if the need arises.

6.3 Extended VT Compression

In Section 5.4 we introduced a rule for compressing a vector timestamp before transmission. One might question the utility of such a technique within a single process group, especially if the group is likely to be small. In a multiple-group setting, the same technique might permit a process to omit entire vector timestamps from some of its multicasts. Additionally, because our group flush algorithm resets the group timestamp to all zeros, the multiple-group algorithm will frequently obtain a vector timestamp for a group in which there has been no communication since the last flush, and hence is all zeros. Obviously, such a timestamp can be omitted.

More generally, the "latest" nonzero timestamp for a group g_a need only be included on the first of any series of messages sent in some other group g_b. This is the timestamp obtained by taking the element-by-element maximum for fields in all received timestamps for the group. Further communication within g_b need not include this timestamp, since all this communication will be causally after the message that contained the updated value of VT_a. To be precise, any process in g_b that updates its value of VT_a as a result of a message received from some process *not* in g_b will include the updated value of VT_a in the next message m that it multicasts to g_b. The updated value

need not be included in any subsequent messages multicast to g_b. Further details of this optimization, and a proof of correctness, can be found presented by Stephenson [28].

We can do even better. Recall the definition of multicast stability (Section 5.3). Say that a group is *active for process p*, if

(1) p is the initiator of a multicast to g that is not stable, or
(2) p has received an unstable multicast to g.

Activity is a local property; i.e., process p can compute whether or not some group g is active for it by examining local state. Moreover, g may be active for p at a time it is inactive for q (in fact, by delaying the delivery of messages, a process may force a group to become inactive just by waiting until any active multicasts stabilize).

Now, we modify step 2′ of the extended protocol as follows. A process p which receives a multicast m in group g_a must delay m until any multicasts m', apparently concurrent with m, and previously received in other groups g_b ($b \neq a$) have been delivered locally. For example, say that process p receives m_1 in group g_1, then m_2 in group g_2, and then m_3 in group g_1. Under the rule, m_2 must not be delivered until after m_1. Similarly, m_3 must not be delivered until after m_2. Since the basic protocol is live in any single group, no message will be delayed indefinitely under this modified rule.

Then, when sending messages (in any group), timestamps corresponding to inactive groups can be omitted from a message. The intuition here is that it is possible for a stable message to have reached its destinations, but still be blocked on some delivery queues. Our change ensures that such a message will be delivered before any subsequent messages received in other groups. Knowing that this will be the case, the vector timestamp can be omitted.

It is appealing to ask how effective timestamp compression will be in typical ISIS applications. In particular, if the compression dramatically reduces the number of timestamps sent on a typical message, we will have arrived at the desired, low overhead, protocol. On the other hand, if compression is ineffective, measures may be needed to further reduce the number of vector timestamps transmitted. Unfortunately, we lack sufficient experience to answer this question experimentally. At this stage, any discussion must necessarily be speculative.

Recall from Section 2 that ISIS applications are believed to exhibit *communication locality*. In our setting, locality would mean that a process that most recently received a message in group g_i will probably send and receive several times in g_i before sending or receiving in some other group. It would be surprising if distributed systems did not exhibit communication locality, since analogous properties are observed in almost all settings, reflecting the fact that most computations involve some form of looping [12]. Process-group-based systems would also be expected to exhibit locality for a second reason: when a distributed algorithm is executed in a group g there will often be a flurry of message exchange by participants. For example, if a process group were used to manage transactionally replicated data, an update transaction

might multicast to request a lock, to issue each update, and to initiate the commit protocol. Such sequences of multicasts arise in many ISIS algorithms.

The extended compression rule benefits from communication locality, since few vector timestamps would be transmitted during a burst of activity. Most groups are small, hence those timestamps that *do* need to be piggybacked on a message will be small. Moreover, in a system with a high degree of locality, each process group through which a vector timestamp passes will "delay" the timestamp briefly.

For example, suppose that a process that sends one multicast in group g_i will, on the average, send and receive a total of n multicasts in g_i before sending or receiving in some other group. Under our extended rule, only the first of these multicasts will carry vector timestamps for groups other than g_i. Subsequent multicasts need carry no vector timestamps at all, since the sender's timestamp can be deduced using the method of Section 5.4. Moreover, if the vector timestamp for a group g_1 changes k times per second, members of an adjacent group g_2 (that are not also members of g_1) will see a rate of change of k/n. A group at distance d would see every n^dth value, giving a rate of k/n^d per second. Thus, the combination of compression and communication locality can substantially reduce the vector timestamp overhead on messages. In fact, if most messages are sent in bursts, the "average" multicast may not carry any timestamps at all!

6.4 Garbage Collection of Vector Timestamps

Our scheme is such that the accumulation of "old" vector timestamps can occur in processes not belonging to the groups for which those timestamps were generated. For example, say that process p, not a member of group g, receives a message dependent on some event in group g. Then p will obtain a copy of $VT(g)$, and will retain it indefinitely (although transmitting it only once to each process in the groups to which it belongs). This could introduce significant overhead when a process joins a group, since it will need to transmit a large number of timestamps on its first multicast in the group, and group members will reciprocally need to send it any vector timestamps that they hold.

In fact, there are several ways that old timestamps could be garbage collected. An especially simple solution is to flush any active group periodically (say, at least once every n seconds). Then if the time at which a vector timestamp was received is known, the timestamp can always be discarded after n seconds.

6.5 Atomicity and Group Membership Changes

The protocols for group flush and multicast atomicity need to be reconsidered in light of this multiple group extension.

Virtually synchronous addressing. Recall that virtually synchronous addressing is implemented using a group flush protocol. In the absence of failures, the protocol of Section 5.5 continues to provide this property.

Although it may now be the case that some messages arrive carrying "stale" vector timestamps corresponding to group views that are no longer installed, the convention of tagging the vector timestamp with the view index number for which is was generated permits the identification of these timestamps, which may be safely ignored: any messages to which they refer were delivered during an earlier view flush operation.

Atomicity and virtual synchrony when failures occur. When the flush algorithm of Section 5.5 is applied in a setting with multiple, overlapping groups, failures introduce problems not seen in single-group settings.

Consider two processes p_1, p_2 and two groups g_1, g_2, such that p_1 belongs to g_1 and p_2 to both g_1 and g_2, and suppose the following event sequence occurs:

(1) p_1 multicasts m_1 to g_1 in $view(g_1)$.

(2) p_2 receives and delivers m_1, while in $(view(g_1), view(g_2))$.

(3) p_2 multicasts m_2 to g_2, still in $(view(g_1), view(g_2))$.

(4) Both p_1 and p_2 fail, causing the installation of $view(g_1)'$ and $view(g_2)'$.

Now, consider a process q belonging to both g_1 and g_2. This process will be a destination of both m_1 and m_2. If p_2 was the only process to have received m_1 before p_1 failed, m_1 will be lost due to the failure; q would then install $view(g_1)'$ without delivering m_1. Further, suppose that m_2 has been received by another process q', belonging to g_2 but not g_1. If q' remains operational, q will receive m_2 during the flush protocol for $view(g_2)'$. This creates a problem:

(1) If q delivers m_2 before installing $view(g_2)'$, causality will be violated, because m_1 was not delivered first.

(2) If m_2 is not delivered by q, atomicity will be violated, because m_2 was delivered at a process q' that remained operational.

Even worse, q may not be able to detect any problem in the first case. Here, although m_2 will carry a vector timestamp reflecting the transmission of m_1, the timestamp will be ignored as "stale." In general, q will only recognize a problem if m_2 is received before the flush of g_1 has completed.

There are several ways to solve this problem. Since the problem does not occur when a process communicates only within a single group, our basic approach will be to intervene when a process begins communicating with another group, delaying communication in group g_2 until causally prior messages to group g_1 are no longer at risk of loss. Any of the following rules would have this effect:

—One can build a k-resilient protocol that operates by delaying communication outside a group until all causally previous messages are k-stable; as k approaches n, this becomes a conservative but completely safe approach. Here, the sender of a message may have to wait before being permitted to transmit it.

—Rely on the underlying message transport protocol to deliver the message to all destinations despite failures. For example, a token ring or ethernet

might implement reliable multicast transport directly at the link level, so that messages are never lost at this stage.

—Construct a special message-logging server which is always guaranteed to have a copy of any messages that have not yet been delivered to all their destinations. Such a facility would resemble the *associative message store* in the original ISIS system [7] and also recalls work by Powell and Presotto [23].

Our implementation uses the first of these schemes, and as initially structured, operates completely safely (i.e., with $k = n$). That is, a process p_i that has sent or received multicasts in group g_1 will delay initiation of a multicast in group g_2 until the g_1 multicasts are all fully stable. Our future system will be somewhat more flexible, allowing the process that creates a group to specify a value of k for that group; such an approach would have a performance benefit. We note that standard practice in industry is to consider a system fault-tolerant if it can tolerate a single failure, i.e., $k = 1$.

Barry Gleeson[6] has made the following observation. Delays associated with multicast stability for reasons of atomicity represent the most likely source of delay in our system. However, with the following sort of specialized multicast transport protocol, this delay can be completely eliminated. Consider an application in which the sender of a message is often coresident with one of the recipients, that is, on the same physical computer, and in which the network interface is reliable (lossless) and sequenced (sends messages in order, even when the destination sets differ). This implies, for example, that if a message m is received by a process p, any message m' transmitted from the same source prior to m will have already been received at all its destinations.

On the resulting system, it would never be necessary to delay messages to a local destination: any action taken by a local recipient and causally related to the received message would only reach external nodes after stability of the prior multicast. For k-stability in the case $k = 1$, a remote destination would never need to delay a message because k stability has (trivially) been achieved in this case. With reliable multicast hardware (or a software device driver that implements a very simple form of reliable, FIFO-ordered multicast), reception might even imply total stability. In such settings, no message need ever be delayed due to atomicity considerations! Our protocols could then perform particularly well, since they would tend to "pipeline" multicasts between nodes, while never delaying intranode communication at all.

6.6 Use of Communication Structure

Until the present, we have associated with each message a vector time or vector times having a total size that could be linear in the number of processes and groups comprising the application. On the one hand, we have argued that in many systems compression could drastically reduce size.

[6] Barry Gleeson is with the UNISYS Corporation, San Jose, California.

Moreover, similar constraints arise in many published CBCAST protocols. However, one can still imagine executions in which vector sizes would grow to dominate message sizes. A substantial reduction in the number of vector timestamps that each process must maintain and transmit is possible in the case of certain communication patterns, which are defined precisely below. Even if communication does not always follow these patterns, our new solution can form the basis of other slightly more costly solutions which are also described below.

Our approach will be to construct an abstract description of the group overlap structure for the system. This structure will not be physically maintained in any implementation, but will be used to reason about communication properties of the system as it executes. Initially, we will assume that group membership is "frozen" and that the communication structure of the system is static. Later, in Section 6.7, we will extend these to systems with dynamically changing communication structure. For clarity, we will present our algorithms and proofs in a setting where timestamp compression rules are *not* in use; the algorithms can be shown to work when timestamp compression is in use, but the proofs are more complicated.

Define the *communication structure* of a system to be an undirected graph $CG = (G, E)$ where the nodes, G, correspond to process groups and edge (g_1, g_2) belongs to E iff there exists a process p belonging to both g_1 and g_2. If the graph so obtained has no biconnected component[7] containing more than k nodes, we will say that the communication structure of the system is k-bounded. In a k-bounded communication structure, the length of the largest simple cycle is k.[8] A 0-bounded communication structure is a tree (we neglect the uninteresting case of a forest). Clearly, such a communication structure is acyclic.

Notice that causal communication cycles can arise even if CG is acyclic. For example, in Figure 4, messages m_1, m_2, m_3 and m_4 form a causal cycle spanning both g_1 and g_2. However, the acyclic structure *restricts* such communication cycles in a useful way. Below, we demonstrate that it is unnecessary to transport all vector timestamps on each message in the k-bounded case. If a given group is in a biconnected component of size k, processes in this group need only to maintain and transmit timestamps for other groups in this biconnected component. We can also show that they need to maintain *at least* these timestamps. As a consequence, if the communication structure is acyclic, processes need only maintain the timestamps for the groups to which they belong.

We proceed to the proof of our main result in stages. First we address the special case of an acyclic communication structure, and show that if a system has an acyclic communication structure, each process in the system only maintains and multicasts the VT timestamps of groups to which it belongs.

[7] Two vertices are in the same biconnected component of a graph if there is still a path between them after any other vertex has been removed.

[8] The nodes of a simple cycle (other than the starting node) are distinct; a complex cycle may contain arbitrary repeated nodes.

Notice that this bounds the overhead on a message in proportion to the size and number of groups to which a process belongs.

We will wish to show that if message m_1 is sent (causally) before message m_k, then m_1 will be delivered before m_k at all overlapping sites. Consider the chain of messages below.

$$p_1 \overset{m_1}{\underset{g_1}{\Rightarrow}} p_2 \overset{m_2}{\underset{g_2}{\Rightarrow}} p_3 \overset{m_3}{\underset{g_3}{\Rightarrow}} \cdots p_{k-1} \overset{m_{k-1}}{\underset{g_{k-1}}{\Rightarrow}} p_k \overset{m_k}{\underset{g_k}{\Rightarrow}} p_{k+1}$$

This schema signifies that process p_1 multicasts message m_1 to group g_1, that process p_2 first receives message m_1 as a member of group g_1 and then multicasts m_2 to g_2, and so forth. In general, g_i may be the same as g_j for $i \neq j$ and p_i and p_j may be the same even for $i \neq j$ (in other words, the processes p_i and the groups g_i are not necessarily all different). Let the term *message chain* denote such a sequence of messages, and let the notation $m_i \overset{p_j}{\rightsquigarrow} m_j$ mean that p_j transmits m_j using a timestamp $VT(m_j)$ that directly reflects the transmission of m_i. For example, say that m_i was the kth message transmitted by process p_i in group g_a. So $m_i \overset{p_i}{\rightsquigarrow} m_j$ iff $VT_a(p_j)[i] \geq k$ and consequently $VT_a(m_j)[i] \geq k$. Our proof will show that if $m_i \rightarrow m_j$ and the destinations of m_i and m_j overlap, then $m_i \overset{p_j}{\rightsquigarrow} m_j$, where p_j is the sender of m_j. Consequently, m_i will be delivered before m_j at any overlapping destinations.

We now note some simple facts about this message chain that we will use in the proof. Recall that a multicast to a group g_a can only be performed by a process p_i belonging to g_a. Also, since the communication structure is acyclic, processes can be members of at most two groups. Since m_k and m_1 have overlapping destinations, and p_2, the destination of m_1, is a member of g_1 and of g_2, then g_k, the destination of the final broadcast, is either g_1 or g_2. Since CG is acyclic, the message chain $m_1 \ldots m_k$ simply traverses part of a tree reversing itself at one or more distinguished groups. We will denote such a group g_r. Although causality information is lost as a message chain traverses the tree, we will show that when the chain reverses itself at some group g_r, the relevant information will be "recovered" on the way back.

LEMMA 3. *If a system has an acyclic communication structure, each process in the system only maintains and multicasts the VT timestamps of groups to which it belongs.*

PROOF. The proof is by induction on l, the length of the message chain $m_1 \ldots m_k$. Recall that we must show that if m_1 and m_k have overlapping destinations, they will be delivered in causal order at all such destinations, i.e., m_1 will be delivered before m_k.

Base case. $l = 2$. Here, causal delivery is trivially achieved, since $p_k \equiv p_2$ must be a member of g_1 and m_k will be transmitted with g_1's timestamp. It will therefore be delivered correctly at any overlapping destinations.

Inductive step. Suppose that our algorithm delivers all pairs of causally related messages correctly if there is a message chain between them of length $l < k$. We show that causality is not violated for message chains where $l = k$. Consider a point in the causal chain where it reverses itself. We represent this by $m_{r-1} \rightarrow m_r \rightarrow m_{r'} \rightarrow m_{r+1}$, where m_{r-1} and m_{r+1} are sent in $g_{r-1} \equiv g_{r+1}$ by p_r and p_{r+1} respectively, and m_r and $m_{r'}$ are sent in g_r by p_r and $p_{r'}$. Note that p_r and p_{r+1} are members of both groups. This is illustrated in Figure 5. Now, $m_{r'}$ will not be delivered at p_{r+1} until m_r has been delivered there, since they are both broadcast in g_r. We now have $m_{r-1} \overset{p_r}{\rightarrow} m_r \overset{p_{r+1}}{\rightarrow} m_{r+1}$. We have now established a message chain between m_1 and m_k where $l < k$. So, by the induction hypothesis, m_1 will be delivered before m_k at any overlapping destinations, which is what we set out to prove. □

THEOREM 1. *Each process p_i in a system needs only to maintain and multicast the VT timestamps of groups in the biconnected components of CG to which p_i belongs.*

PROOF. As with Lemma 3, our proof will focus on the message chain that established a causal link between the sending of two messages with overlapping destinations. This sequence may contain simple cycles of length up to k, where k is the size of the largest biconnected component of CG. Consider the simple cycle illustrated below, contained in some arbitrary message chain.

$$p_1 \overset{m_1}{\underset{g_1}{\Rightarrow}} \cdots p_2 \overset{m_l}{\underset{g_l}{\Rightarrow}} p_3 \overset{m_{l+1}}{\underset{g_1}{\Rightarrow}}$$

Now, since p_1, p_2 and p_3 are all in groups in a simple cycle of CG, all the groups are in the same biconnected component of CG, and all processes on the message chain will maintain and transmit the timestamps of all the groups. In particular, when m_l arrives at p_3, it will carry a copy of VT_{g1} indicating that m_1 was sent. This means that m_l will not be delivered at p_3 until m_1 has been delivered there. So m_{l+1} will not be transmitted by p_3 until m_1 has been delivered there. Thus $m_1 \overset{p_3}{\rightarrow} m_{l+1}$. We may repeat this process for each simple cycle of length greater than 2 in the causal chain, reducing it to a chain within one group. We now apply Lemma 3, completing the proof. □

Theorem 1 shows us what timestamps are sufficient to assure correct delivery of messages. Are all these timestamps in fact necessary? It turns out that the answer is yes. It is easy to show that if a process that is a member of a group within a biconnected component of CG does not maintain a VT timestamp for some other group in CG, causality may be violated. We therefore state without formal proof:

THEOREM 2. *If a system uses the VT protocol to maintain causality, it is both necessary and sufficient for a process p_i to maintain and transmit those*

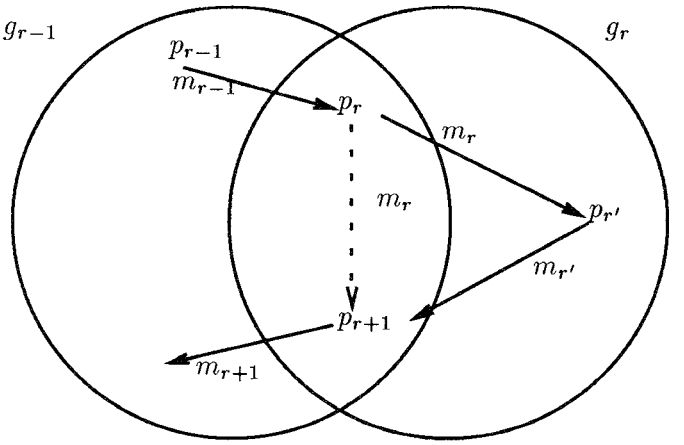

Fig. 5. Causal reversal

VT timestamps corresponding to groups in the biconnected component of CG to which p_i belongs.

6.7 Extensions to Arbitrary, Dynamic Communication Structures

The previous section assumed that the communication graph was known statically. Operationally, this would correspond to a system in which, once established, process group membership never changed. Any realistic application is likely to be more dynamic, making it hard to manage information concerning the biconnected components of *CG*. Moreover, any real distributed system will probably contain a mixture of subsystems, some having a regular communication structure, and some not.

Consider the multiple-group examples raised in the introduction. A scientific computation using groups for nearest neighbor communication will have a regular communication structure. The structure is known in advance and is a property of the algorithm, and it would be desirable to exploit this to reduce overhead on messages. Lemma 3 and Theorem 1 are ideally suited for this purpose. (We describe the system call interface used to communicate this information in Section 7.3).

This might or might not be true for the network information service. The designer of such a service has a choice between controlling the location of replication domains, or permitting data to migrate in an uncontrolled manner, creating fairly random domains. However, such a service will want to balance processor loads and storage utilization, which might be hard in the latter approach. Thus, the designer might well prefer to "tile" the network with process groups in a regular way, which could then be exploited using our above results—again, presuming a suitable interface for communicating this information.

On the other hand, a system built of abstract objects will almost certainly have an arbitrary communication structure that changes continuously as applications are started and terminate. Here, the communication structure

would be impossible to represent; indeed, it may well change faster than information about it can be propagated within the system. The best a process could possibly do is to reason about local properties of the structure.

We now develop some simple results that enable processes to maintain only timestamps for groups to which they actually belong, and yet to operate safely in dynamically changing communication graphs that may contain cycles. Below, we will assume that processes may join and leave groups dynamically, and may leave a group for reasons other than failure (in existing ISIS this is possible, but uncommon). This results in a highly dynamic environment. Nonetheless, a process might be able to infer that a group to which it belongs is not present in any cycle. This would follow, for example, if the group is adjacent to at most one other group. Such information can be obtained by an exchange of adjacency information when a process joins a group, and subsequently multicasting updated information in g_1 each time a current member joins some other group g_2. Further, say that group g_2 is adjacent to groups g_1 and g_3, but that g_1 is adjacent to no other group. Then g_2 and eventually g_3 may come to learn that there is no cycle present.

Conservative solution. Our first solution is called the *conservative protocol* and uses multicast stability (Section 5.3). The idea will be to restrict the initiation of new multicasts so that a message m can only be sent in a group g when it is known that any causally prior messages m' will be delivered first, if m and m' share destinations.

The *conservative multicast rule* states that a process p may multicast to group g_1 iff g_1 is the only active group for process p or p has no active groups (the notion of an active group was defined in Section 6.3). If p attempts to multicast when this rule is not satisfied, it is simply delayed. During this delay, incoming messages are not delivered. This means that all groups will eventually become inactive and the rule above will eventually be satisfied. At this point, the message is sent. It is now immediate from the extended compression rule of Section 6.3 that when a message m is multicast in group g, only the sender's timestamp for group g need be transmitted.

The conservative rule imposes a delay only when two causally successive messages are sent to *different* groups. Thus, the rule would be inexpensive in systems with a high degree of locality. On the other hand, the overhead imposed would be substantial if processes multicast to several different groups in quick succession.

Multicast epochs. We now introduce an approach capable of overcoming the delays associated with the conservative rule but at the cost of additional group flushes. We will develop this approach by first assuming that a process leaves a group only because of a failure and then extending the result to cover the case of a process that leaves a group but remains active (as will be seen below, this can create a form of phantom cycle).

Assume that CG contains cycles but that some mechanism has been used to select a subset of edges X such that $CG' = (G, E - X)$ is known to be acyclic. We extend our solution to use the acyclic protocol proved by Lemma 2 for *most* communication within groups. If there is some edge $(g, g') \in X$, we

will say that one of the two groups, say g, must be designated as an *excluded group*. In this case, all multicasts to or from g will be done using the protocol described below.

Keeping track of excluded groups could be difficult; however, it is easy to make pessimistic estimates (and we will derive a protocol that works correctly with such pessimistic estimates). For example, in ISIS, a process p might assume that it is in an excluded group if there is more than one other neighboring group. This is a safe assumption; any group in a cycle in CG will certainly have two neighboring groups. This subsection develops solutions for arbitrary communication structures, assuming that some method such as the previous is used to safely identify excluded groups.

We will define a notion of multicast *epoch*, to be associated with messages such that if for two messages m_1 and m_2, $epoch(m_1) < epoch(m_2)$, then m_1 will always be delivered before m_2. In situations where a causal ordering problem could arise, our solution will increment the *epoch* counter.

Specifically, each process p maintains a local variable, $epoch_p$. When process p initiates a multicast, it increments its *epoch* variable if the condition given below holds. It then piggybacks $epoch_p$ on the outgoing message. On reception of a message m, if $epoch_p < epoch(m)$, then p will initiate the flush protocol for all groups to which it belongs, by sending a message **"start flush"** to the other group members. Reception of this message triggers execution of the flush protocol of Section 5.5, just as for a new group view (because our implementation clears vector timestamps as part of the flush, ISIS numbers views using a major and minor view number: the major number is incremented for each new view, and the minor one for each flush done within the same view). On completing the flush protocol, all group members set the value of their *epoch* variables to the maximum of the ones held by group members.

When will a process increment its epoch variable? Say that m is about to be multicast by p to g. We say that p *is not safe in* g if:

—The last message p received was from some other group g', and
—either g or g' is an excluded group.

Our protocol rule is simple; on sending, if process p is not safe in group g, p will increment $epoch_p$ before multicasting a message m to g. In this case, m will carry the *epoch* value but need not carry *any* vector timestamps; reception of m will cause a flush in g (and any other groups to which recipients belong). Otherwise, p will just increment its VT timestamp in the usual manner, and then piggyback onto m the *epoch* variable and timestamps for any (active) groups to which it belongs. A message is delivered when it is deliverable according to both the above flush rule and the VT delivery rule.

Notice that the flushing overhead of the modified protocol is incurred only when *epoch* values actually change, which is to say only on communication within two different groups in immediate succession, where one of the groups is excluded. That is, if process p executes for a period of time using the VT protocol and receives only messages that leave $epoch_p$ unchanged, p will not

initiate a flush. However, when an *epoch* variable is incremented[9] the result could be a cascade of group flushes. Epoch variables will stabilize at the maximum existing value and flushing will then cease.

THEOREM 3. *The VT protocol extended to implement multicast epochs will deliver messages causally within arbitrary communication structures.*

PROOF. Consider an arbitrary message chain where the first and last messages have overlapping destinations. For example, in the chain shown below, p_{k+1} might be a member of both g_1 and g_k and hence a destination of both m_1 and m_k. Without loss of generality, we will assume that $g_1 \ldots g_k$ are distinct. We wish to show that the last message will be delivered after the first at all such destinations.

$$p_1 \underset{g_1}{\overset{m_1}{\Rightarrow}} p_2 \underset{g_2}{\overset{m_2}{\Rightarrow}} \ldots p_k \underset{g_k}{\overset{m_k}{\Rightarrow}} p_{k+1}$$

If none of $g_1 \ldots g_k$ is an excluded group, then, by Lemma 3, m_1 will be delivered before m_k at p_{k+1}. Now, let g_i be the first excluded group in the above message chain. If g_i is excluded, then p_i will increment its *epoch* variable before sending m_i. As *epoch* numbers can never decrease along a causal chain, we will have $epoch(m_1) < epoch(m_k)$, and a flush protocol will have run in any groups to which a destination of m_k belongs, before m_k can be delivered. m_1 was sent before the flush, and hence will be delivered by p_{k+1} before it delivers m_k. □

We have been assuming that a process only leaves a group because of failure. Now, without changing the definition of the communication graph, say that processes can also leave groups for other reasons, remaining active and possibly joining other groups. Earlier, it was suggested that a process might observe that the (single) group to which it belongs is adjacent to just one other group and conclude that it cannot be part of a cycle. In this class of applications, this rule may fail. The implication is that a process that should have incremented its *epoch* variable may neglect to do so, thereby leading to a violation of the causal delivery ordering.

To see how a problem could arise, suppose that a process p belongs to group g_1, then leaves g_1 and joins g_2. If there was no period during which p belonged to both g_1 and g_2, p would use the acyclic *VT* protocol for all communication in both g_1 and g_2. Yet, it is clear that p represents a path by which messages sent in g_2 could be causally dependent upon messages p received in g_1, leading to a cyclic message chain that traverses g_1 and g_2. This creates a race condition under which violations of the causal delivery ordering could result.

[9] An interesting variation on this scheme would involve substituting synchronized real-time clocks for the epoch variable; a message would then be delayed until the epoch variable for a recipient advanced beyond some minimum value. This would also facilitate implementation of the real-time *VT* garbage collection method of Section 6.4. Readers familiar with the △-T real-time protocols of Cristian et al. [11] will note the similarity between that protocol and such a modification of ours. In fact, clock synchronization (on which the △-T scheme is based) is normally done using periodic multicasts [17, 27].

This problem can be overcome in the following manner. Each process p_i simply increments its epoch counter after it leaves a group g_a and before it sends any subsequent messages. This will ensure that any message sent by p_i after it leaves g_a will be delivered subsequent to any message whose effect p_i had observed, directly or indirectly, before it left g_a.

7. APPLYING THE PROTOCOLS TO ISIS

This section briefly discusses some pragmatic considerations that arise when implementing the protocols for use in ISIS.

7.1 Optimization for Client / Server Groups

Up to now, our discussion has focused on communication in *peer groups*. In ISIS client/server settings, a set of servers forms one group, and each client of the service they are providing forms an additional group containing that client and the server set (see Figure 1). Theorem 2 appears to state that in this case, each group containing one of the clients needs to maintain the timestamps of every other such client group—a total timestamp size of $O(s*c)$, where s is the number of servers and c is the number of clients. Fortunately, since clients will not be communicating with each other except through the servers, and the servers form a peer subgroup of size s that receives all of these multicasts, an optimization can be applied to reduce the entire timestamp size to $O(s + c)$. This optimization essentially collapses all the client groups into one large group; it is fully described by Stephenson [28]. A modified version of the conservative delay rule can be used to reduce any timestamps transmitted outside of the group to size s. Finally, since timestamps associated with inactive groups can be omitted, and most clients of a large group will presumably be inactive, even the internal timestamps can be reduced in size, to length $O(s + k)$ for some small k.[10]

Our protocols also work well with the other styles of process group usage, as summarized in Table I. In diffusion groups, one vector timestamp is needed for the whole group. The number of entries in the timestamp can be optimized: entries are only needed for the server processes, since these are the only ones that initiate multicasts. Hierarchical groups fall naturally into our general approach: since a process normally interacts with a single representative subgroup, the length of the vector timestamp seen is normally determined by the size of that subgroup. Further, a hierarchical group manager might have the freedom to create subgroups to be explicitly acyclic.

Some ISIS applications form large process groups but would benefit from the ability to multicast to *subsets* of the total membership. For example, a stock quote service might send an IBM quote to only those brokers actually trading IBM stock. Such an ability can be emulated by forming groups approximating the multicast destination sets and discarding unwanted messages. Alternatively, our protocol can be extended into one directly support-

[10] For example, by implementing the compression scheme or by simply having a client of a group drop out of it after some period of inactivity.

Table I. Overhead Resulting from Different Process Group Styles

Fixed Overhead	$sender\_id, uid, epoch_p, dest\_group, stable_{dest}(p)$ 28 bytes total
Variable Overhead	VT timestamps as needed (Section 6.3)
VT timestamp sizes: *Peer Group*	$k, k \leq n$ n is the number of group members.
Client/Server	$k, k \leq s + c$ s is the number of servers and c is the number of clients.
Diffusion	$k, k \leq b$ b is the number of members broadcasting into the group.
Hierarchical	$k, k \leq n$ n is the size of a subgroup.

ing subset multicast. The basic idea is to move to a large timestamp, representing the number of times each possible sender has sent to each possible recipient. The resulting array would be sparse and repetitious, and hence could be substantially compressed. At present, we favor the former scheme, as it requires no changes to our basic protocol.

7.2 Point-to-Point Messages

Early in the the paper, we asserted that asynchronous CBCAST is the dominant protocol used in ISIS. Point-to-point messages, arising from replies to multicast requests and RPC interactions, are also common. In both cases, causal delivery is desired. Our implementation supports the transmission of point-to-point messages with causal delivery guarantees. This is done using an RPC scheme, in which the sender is inhibited from starting new multicasts until reception of the point-to-point message is acknowledged. The sender transmits the vector timestamps that would be needed for a CBCAST, but does not increment its own vector timestamp prior to transmission. Point-to-point messages are thus treated using the basic causality algorithm but are events internal to the processes involved.

The argument in favor of this method is that a single point-to-point RPC is fast and the cost is unaffected by the size of the system. Although one can devise more complex methods that eliminate the period of inhibited multicasting, problems of fault-tolerance render them less desirable.

7.3 System Interface Issues

One question raised by our protocols concerns the mechanism by which the system would actually be informed about special application structure, such as an acyclic communication structure. This is not an issue in the current ISIS implementation, which uses the conservative rule, excluding groups adjacent to more than one neighboring group. In the current system, the only

problem is to detect clients of a group, and as noted earlier, these declare themselves through the **pg_client** interface.

In the future, we expect to address these issues through a new system construct, the *causality domain* [4]. A causality domain is a set of groups within which causality is enforced. Each group is created in a domain and subsequently remains in it. Causality is *not* enforced between domains, but a **flush** primitive can be provided, which will block a process until all causally prior messages have been delivered.[11] We are designing an interface by which parameters such as the message stability constant k or assertions about the communication structure can be asserted on a per-domain basis. In the case of a domain declared to have an *acyclic* communication structure, a routine **pg_exclude** may be used to designate excluded groups.

An application such as the physics simulation described earlier could set up its group structure as a separate causality domain, declaring it to be acyclic and excluding enough groups to ensure that cycles will be broken. By invoking **flush** before switching from domain to domain, causal safety would be achieved on both intra- and interdomain operations.

7.4 Group View Management

The current ISIS implementation retains the prior ISIS group view management server. Looking to the future, however, it will be possible to use our new protocol in a stand-alone fashion. Membership changes (adding or dropping a member from a group) can be implemented using the CBCAST protocol, including the new member as a destination in the former case. Clearly, this requires a form of mutual exclusion, which is obtained by having a distinguished group member initiate any such membership changes.[12] Reception of this CBCAST message would trigger a view flush protocol: approximately $3n$ messages are thus needed to add one member to a group of size n. The addition or deletion of a client in a client-server or diffusion group is cheaper: a multicast and flush are still needed, but since clients don't need to know about other clients, it can be confined to the server subgroup.

A source of failure information is needed in this scheme. In a previous article [24], we discuss the asynchronous failure detection problem and present an optimal algorithm.

Such a redesigned system would realize a long-term goal of our effort. The current ISIS architecture is difficult to scale to very large LAN settings because of its reliance on a central protocol server. Although this server need not reside directly on every node, it introduces a bottleneck that limits the scalability of the architecture to networks with at most a few hundred

[11] There are a number of possible implementations of **flush**, but for brevity we defer discussion of this issue to a future paper.

[12] As in conventional distributed computing environments, this approach assumes that groups would be registered with some sort of group location service Initial connection to a group would be via a forwarded request, causing the caller to be added as a new member or client. Subsequent to this, group operations could be performed directly.

workstations. By reimplementing the group view mechanism in terms of our new multicast protocol and separating failure detection into a free-standing module, this limit to scalability can be eliminated.

8. PERFORMANCE

We implemented the new CBCAST and ABCAST protocols within the ISIS Toolkit, Versions 2.1 and 3.0 (respectively available since October 1990 and March 1991); the figures below are from the V3.0 implementation evaluated using SUN 4/60 workstations on a lightly loaded 10Mbit Ethernet. The implementation is less ambitious than what we plan for the future ISIS system, which will support the new protocols directly in the operating system. Nonetheless, the measured performance was encouraging.

Table II shows that the cost of transmitting a message using CBCAST grows roughly linearly with the size of the destination group. The first line of the table shows the cost of sending a CBCAST to another thread in the same process; subsequent lines show the cost to one or more remote destinations. Figures for ABCAST appear in Table III; in these, the multicasting process did not hold the ordering token.

ISIS performance can be compared to what can be achieved using the vendor-supplied remote procedure mechanism for the machines with which we worked. For example, a null 2-process remote procedure call using our facility had a round-trip latency of 6.51ms; with a 1k message and a null reply we measured 7.86ms. The figures for SUN RPC (over TCP) are 3.5ms and 4.5ms respectively.[13] These SUN figures do not include creation of any sort of lightweight task on the remote side, nor is the wall-clock time when a message was sent or received recorded, although ISIS does both. Thus, of the 3.36ms additional cost seen with the 1k ISIS RPC, a substantial part is attributable to aspects of the ISIS environment unrelated to the multicast protocol. We note that a 7-destination null multicast in which all destinations reply costs 22.9ms using our protocol; 7 successive null RPC's would cost 24.5ms using the SUN protocol.

Figure 6 shows the relative cost of CBCAST RPC's as a function of packet size and number of sites; Figure 7 compares the cost of a 1k CBCAST to a 1k ABCAST as the number of sites grows. From these graphs we see that packet size is a dominant factor in determining cost, and that ABCAST runs at roughly half the speed of CBCAST in tests with small packets. This last point reflects an initial ABCAST implementation in which the ordering token does not move, and also the higher scheduling and IO costs incurred by applications running the more synchronous protocol.

ISIS throughput figures compare well with traditional streaming protocols such as TCP: for 7k packets sent to a single remote destination, CBCAST

[13] It should be noted that although the SUN RPC figure is about as good as one can find in commercial UNIX systems, the state of the art in experimental operting systems is quite a bit faster. We are now working on an an ISIS implementation in the x-Kernel under Mach [1, 21], and plan to compare performance of the resulting ISIS RPC with that of the Mach/x-Kernel RPC.

Table II. Multicast Performance Figures (CBCAST) S: Null Packets; M: 1K Packets; L: 7K Packets. All Figures Measured on SUN 4/60's Running SUNOS 4.1.1

Destinations	Causal multi-rpc				
	All dests reply			Asynchronous	
	(ms)			msg/sec	kb/sec
	S	M	L	S	L
Self	1.51	1.52	1.51	4240	16706
1	6.51	7.86	17.0	1019	550
2	8.67	10.1	23.4	572	361
3	10.7	12.2	34.9	567	249
4	12.9	15.2	43.4	447	180
5	16.3	18.8	54.1	352	143
6	19.6	21.9	64.6	305	118
7	22.9	25.7	75.7	253	101

Table III. Multicast Performance Figures (ABCAST). S: Null Packets; M: 1K Packets; L: 7K Packets

Destinations	Ordered multi-rpc				
	All dests reply			Asynchronous	
	(ms)			msg/sec	kb/sec
	S	M	L	S	L
Self	1.52	1.56	1.55	3300	20000
1	12.7	14.0	25.5	341	560
2	18.5	19.7	30.7	273	317
3	24.6	25.6	42.6	207	175
4	31.4	32.5	53.5	197	152
5	38.1	39.2	70.3	352	115
6	44.3	42.1	77.5	133	98.9
7	55.2	48.3	97.0	97	79.7

achieved exactly the same performance (550k bytes per second) as the TCP implementation supplied by SUN, and the total rate of data sent through UNIX remains between 550kb/sec and 750kb/sec as the number of destinations increases (recall that we currently make no use of hardware multicast so that an n-destination multicast involves sending essentially identical messages n times).

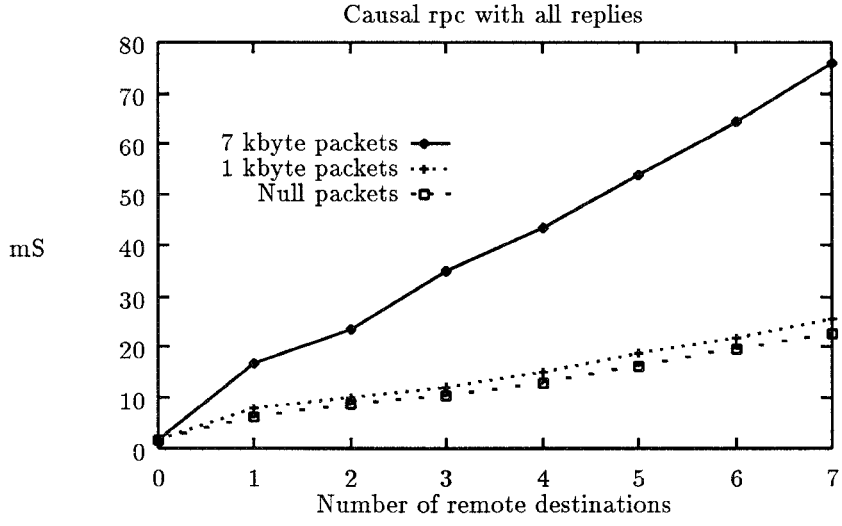

Fig. 6. CBCAST RPC timing as a function of message and group size.

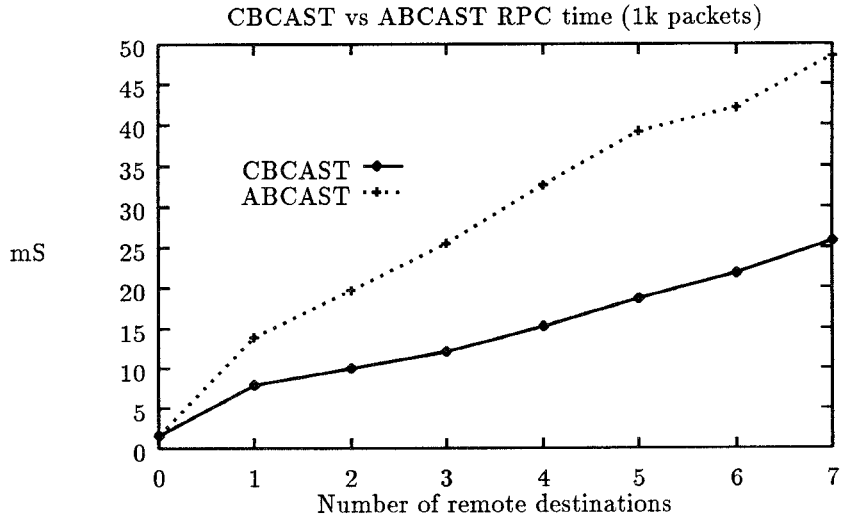

Fig. 7. CBCAST vs. ABCAST RPC timing as a function of group size.

The engineering of UNIX UDP leads to flow control problems when large volumes of data are transmitted. When ISIS streams to, say, 7 destinations, the sender generates large numbers of UDP packets (these are typically 4000-8000 bytes long and may contain as many as 25 ISIS messages piggybacked into a single data packet). For 7 destinations, ISIS may have as many as 100 or more of these messages in transit at one time and when a single process generates such a high volume of UDP traffic, UNIX begins to lose

Table IV. Null Causal Multicast to Another Local Thread with Reply

Operation	Total Cost	Causal Cost
send	406 uS	86 uS
receive	394 uS	31 uS
reply	406 uS	86 uS
receive	224 uS	31 uS
Total	1.43 mS	234 uS
(Measured)	1.51 mS	

both outgoing and incoming UDP packets silently. To avoid this, we resort to heuristic flow control, choking down the sender just before we expect to overload UNIX. Within the x-Kernel, where detailed memory statistics will be directly available, far better flow control should be possible.

One might wonder why we didn't implement ISIS over TCP. Most UNIX implementations limit application programs to have only a small number of open TCP connections at any time, due to memory buffering (mbuf) limitations within the kernel. This is unrelated to limits on the number of *file descriptors* a program may have open.

It is difficult to directly compare the performance of the ISIS protocols with that of other published multicast protocols. We know of no other reliable multicast protocols implemented over UNIX. Perhaps the best known protocol, the Amoeba reliable multicast [15], is constructed as a special device driver implemented at the lowest possible layer of the kernel. This system achieves a performance of more than 1000 ordered multicasts per second. Although we achieve comparable performance for our streaming CBCAST protocol (to one remote destination), the Amoeba performance is unaffected by scale, while ours drops rapidly with scale. Amoeba achieves this impressive result using special hardware, an option that would also accelerate the ISIS protocols.

In order to understand how CPU time was expended by our protocol, we profiled the protocol within a single processor and between a pair of processes using RPC over a 10Mbit ethernet. Both request and reply were sent as single-destination CBCAST's. (The focus on RPC may seem odd, in light of our emphasis on asynchronous multicast, but this is the obvious way to measure round-trip delays). We then computed the costs attributable to different parts of the system. Table IV shows a profile for a null RPC sent by a thread to another entry point within its own address space. This involves creating a new thread to handle each delivered message but no communication outside of the address space of the test program. The first column shows total (measured) costs, the second shows costs attributed to causality, which we obtained by comparing the costs of sending causally ordered and FIFO messages. The send and receive figures were obtained by hardwiring our

Table V. 1k Causal Multicast to a Remote Process with Reply

Causal multicast	.23 mS
Rest of Isis	1.21 mS
Transport	3.6 mS
System Calls	1.59 mS
Wire Time	1.5 mS
Total	8.1 mS
(Measured)	7.86 mS

system to loop through the corresponding sections of code ten thousand times. Cache hit ratios may explain the slightly low estimate.

Table V shows the costs on a layer-by-layer basis for the 7.86ms RPC to a remote process. The costs are broken down into the time spent in the protocol implementation (taken from the null multicast table, above), the transport costs, costs spent in system calls, and the time on the wire for a 1k message with its ISIS-supplied header and a null reply. The header size used was approximately 220 bytes[14] in each case. Our breakdown compares causal multicast with a FIFO transport, as opposed to a completely unordered multicast, because our protocols make such heavy use of a FIFO transport assumption. The reader should note, however, that remote-procedure call protocols would generally not include such a requirement, offering an avenue for improved RPC times (i.e., using a non-FIFO transport) that may be closed to the ISIS system. For example, if the destination is a process running on a parallel processor, ISIS applications will not obtain as much benefit from physical parallelism as could an RPC protocol. The ISIS application would see a serialized stream of incoming messages (which it could accept using multiple threads), while a good RPC implementation might multithread the operating system itself.

The major conclusion we draw from these performance studies is that nearly all the time spent in our new protocol is in the layers concerned with physically transporting messages in FIFO order to a remote machine. Our protocol imposes little cost in comparison to this number. This result has convinced us that our new system should be built over some sort of extensible

[14] Most of this header reflects the ISIS system structure, not our protocols. Each ISIS packet consists of an enclosing message containing one or more data messages, and each of these messages has a 64-byte header, listing information such as the destinations of the message, the sender, the size of the message, the byte format of the sender, etc. An intersite packet carries additional sequencing and acknowledgment data structure, adding 24 bytes. Further, ISIS messages are transmitted as self-describing symbol tables, imposing an overhead of both space and time. The vector timestamp data, in the present implementation, adds $(40 + 4n)$ bytes to a CBCAST in an n-member group. In developing ISIS, this scheme gave us valuable flexibility. However, in reimplementing the system, we will be using a simpler and more compact representation, and this overhead should be much reduced.

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