32P.

Fast Causal Multicast*

Kenneth Birman
Department of Computer Science, Cornell University

Andre Schiper
Ecole Polytechnique Federale de Lausanne, Switzerland

Pat Stephenson
Department of Computer Science, Cornell University

April 10, 1990

Abstract

A new protocol is presented that efficiently implements a reliable, causally ordered multicast primitive and is easily extended into a totally ordered one. Intended for use in the ISIS toolkit, it offers a way to *bypass* the most costly aspects of ISIS while benefiting from *virtual synchrony.* The facility scales with bounded overhead. Measured speedups of more than an order of magnitude were obtained when the protocol was implemented within ISIS. One 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.

Keywords and phrases: Distributed computing, fault-tolerance, process groups, reliable multicast, ABCAST, CBCAST, ISIS.

*This work was supported by the Defense Advanced Research Projects Agency (DoD) under ARPA order 6037, Contract N00140-87-C-8904 and under DARPA/NASA subcontract NAG2-593 administered by the NASA Ames Research Center. 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.

1 Introduction

The Isis Toolkit [BJKS88] 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, ISIS was in use at more than 250 locations worldwide.

Two aspects of Isis are key to its overall approach:

- An implementation of ***virtually synchronous process groups.***
- A collection of atomic multicast protocols with which processes and group members interact with groups.

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 initial Isis **CBCAST** protocol was costly in part for structural reasons, and in part because of the protocol used. 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 tremendously expensive. With respect to the protocol used, our initial implementation favored generality over specialization, permitting extremely flexible destination addressing, and using a piggybacking mechanism that achieved a desired ordering property but required a garbage collection mechanism. On the other hand, this structure seemed to be the only one capable of supporting a powerful, general set of programming tools like the ones in our toolkit: simpler protocols often simply overlook critical forms of functionality, which may explain why so few have entered widespread use. Particularly valuable to us has been the ability to to support multiple, possibly overlapping process groups, and virtual synchrony [BJKS88].

The protocol we present here is based on a causal ordering protocol originally developed by Schiper [SES89]. Unlike our previous work, it assumes a preexisting virtually synchronous programming environment like the one that Isis provides, although using few of its features. Further, it supports a relatively restricted form of multicast addressing. Were our work done outside of the context of Isis, this would seriously limit its generality. In our implementation, however, messages that do not conform to these restrictions are simply routed via the old, more costly algorithm. A highly optimized multicast protocol results that *bypasses* the old Isis system and imposes very little overhead beyond that of the message transport layer. The majority of Isis communication satisfies the requirements of the bypass protocols and hence benefits from our work.

Our protocol uses a timestamping scheme, and in this respect resembles prior work by

Ladkin [LL86] and Peterson [PBS89]. However, our results are substantially more general. The most important differences are these:

- Peterson's Psync-based protocol can be used only in systems composed of a single process group, ours supports multiple, possibly overlapping process groups.
- Both Peterson's and Ladkin's protocols have overhead linear in the number of processes that ever participated in the application, which could be large; our overhead is bounded and small.

Like Peterson's and Ladkin's protocols, our basic protocol provides for message delivery ordering that respects causality in the sender (CBCAST), but is readily extended into a more costly protocol that provides a total delivery ordering even for concurrent invocations (ABCAST).

The bypass protocol suite lets users select the multicast properties desired for an application. Choices include a "raw" delivery service achieving extremely high performance but with minimal reliability guarantees, multicast with atomicity and FIFO delivery, and causal or total ordering. This approach permits the user to pay for just those reliability and ordering properties needed by the application.

The paper is structured as follows. Section 2 reviews the multicasting problem and defines our terminology. Sections 3 and 4 introduce our new technique. Section 5 discussions extensions of the **CBCAST** protocol, including the bypass **ABCAST** protocol. The costs of our various primitives are measured in Section 6.

2 Execution model

2.1 Basic system model

The system is composed of processes $P = \{p_1, p_2, ..., p_n\}$ 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 ISIS in a manner described below. In many situations, processes will need to cooperate. For this purpose, they form *process groups*. Each such 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 set of groups is denoted by $G = \{g_1, g_2 ...\}$. In typical settings, the number of groups will be large and processes will belong to several groups.

Our system model is unusual in assuming an external service that implements the process group abstraction. The interface from a process to this service will not concern us here, but the manner in which the service communicates to a process is highly relevant.

A *view* of a process group is a list of its members. A *view sequence* for g is a list $view_0(g), view_1(g), ..., view_n(g)$, where

1. $view_0(g) = \emptyset$.
2. $\forall i : view_i(g) \subseteq P$, where P is the set of all processes in the system.
3. $view_i(g)$ and $view_{i+1}(g)$ differ by the addition or subtraction of exactly one process.

We assume that some sort of process group service computes new views and communicates them to the members of the groups involved. 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. Our approach permits application builders to define new transport protocols, perhaps to take advantage of special hardware. Our initial implementation uses unreliable datagrams, but has an experimental protocol that exploits ethernet hardware multicast.

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 1 or more destinations $dests(m)$; the receive event is denoted $rcv_p(m)$. We omit the subscript when the context is unambiguous. 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 $deliver(m)$ where $rcv(m) \xrightarrow{p} deliver(m)$.

2.2 Properties required of multicast protocols

Although ISIS makes heavy use of virtual synchrony, it will not be necessary to formalize this property for our present discussion. However, the support of virtual synchrony places several obligations on the processes in our system. First, when a process multicasts a message m to group g, $dests(m)$ must be the current membership of g. Secondly, when the group view changes, all messages sent in the prior view must be "flushed" out of the system (delivered) before the new view may be used. Finally, messages must satisfy a failure atomicity property: if a message m is delivered to any member of a group, and it stay operational, m must be delivered to all members of the group even if the sender fails before completing the transmission.

The multicast protocols that interest us here also provide delivery ordering guarantees. As in [Lam78], 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. $\forall m: \; send(m) \rightarrow rcv(m)$

CBCAST satisfies a causal delivery property:

> If m and m' are **CBCAST**'s and $send(m) \rightarrow send(m')$ then
>
> $\forall p \in dests(m) \cap dests(m'): \; deliver(m) \xrightarrow{p} deliver(m')$.

If two **CBCAST** messages are concurrent, the protocol places no constraints on their delivery ordering at overlapping destinations.

ABCAST extends the **CBCAST** ordering into a total one:

> If m and m' are **ABCAST**'s then either
>
> 1. $\forall p \in dests(m) \cap dests(m'): \; deliver(m) \xrightarrow{p} deliver(m')$, or
> 2. $\forall p \in dests(m) \cap dests(m'): \; deliver(m') \xrightarrow{p} deliver(m)$.

Because the **ABCAST** protocol orders concurrent events, it is more costly than **CBCAST**; requiring synchronous solutions where the **CBCAST** protocol admits efficient asynchronous solutions. Birman and Joseph [BJ89] and Schmuck [Sch88] have exhibited a large class of algorithms that can be implemented using asynchronous **CBCAST**. Moreover, Schmuck has shown that in many settings algorithms specified in terms of **ABCAST** can be modified to use **CBCAST** without compromising correctness.

The protocols presented here all assume that processes only multicast to groups that they are members of, and that all multicasts are to the full membership of a single group.

For demonstrating liveness, we will assume that any message sent by a process is eventually received unless the sender or destination fails, and that failures are eventually reported by ISIS.

3 The CBCAST bypass protocol

This section presents two basic **CBCAST** protocols for use within a single process group with fixed membership. Both use timestamps to delay messages that arrive out of causal order. The section that follows extends these schemes and then merges them to obtain a single solution for use with multiple, dynamic process groups.

3.1 Timestamping protocols

We begin by describing two protocols for assigning timestamps to messages and for comparing timestamps. The protocols are standard except in one respect: whereas most timestamping protocols count arbitrary "events", the ones defined here count only *send* events.

3.2 Logical time

The first timestamping protocol is based on one introduced by [Lam78], called the *logical clock* protocol. Each process p maintains an unbounded local counter, $LT(p)$, which it initializes to zero. For each event $send(m)$ at p, p sets $LT(p) = LT(p) + 1$. Messages are timestamped with the sender's incremented counter. A process p receiving a message with timestamp $LT(m)$ sets $LT(p) = max(LT(p), LT(m))$. As in [Lam78], one can show that if $send(m) \rightarrow send(m')$ then $LT(m) < LT(m')$. The converse, however, does not hold: the protocol may order messages that were sent concurrently.

Note that the LT counter for a process is updated at the *rcv* event, as opposed to the *deliver* event, for an incoming message. We make use of this property in the development below.

3.3 Vector time

A second timestamping protocol is based on the substitution of *vector times* for the local counters in the logical time protocol. Vector times were proposed originally in [Mar84]; other researchers have also used them [Fid88,Mat89,LL86,Sch88]; our use of them is motivated by an protocol presented in [SES89]. In comparison with logical times, this protocol has the advantage of representing → 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 sent 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 time in the following manner:

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

Rules for comparing vector times 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 $\exists i: \; VT_1[i] < VT_2[i]$

Notice that in contrast to the rule for $LT(p)$, $VT(p)$ is updated at the *deliver* event for an incoming message. We will make use of this distinction below.

It can be shown that given messages m and m', $send(m) \rightarrow send(m')$ iff $VT(m) < VT(m')$ [Mat89,Fid88]: vector timestamps represent causality precisely. This constitutes the fundamental property of vector times, and the primary reason for our interest in such times as opposed to logical ones.

3.4 Causal message delivery

Recall that if processes communicate using **CBCAST**, all messages must be delivered in an order consistent with causality. Suppose that a set of processes P communicate using only broadcasts to the full set of processes in the system; that is, $\forall m : dests(m) = P$. This hypothesis is unrealistic, but Section 4 will adapt the resulting protocol to a settings with multiple process groups.[1] We now develop two *delivery protocols* by which each process p receives messages sent to it, delays them if necessary, and then delivers them such that:

If $send(m) \rightarrow send(m')$ then $deliver(m) \rightarrow deliver(m')$.

3.4.1 LT protocol

Our first solution to the problem is based on logical clocks; and is referred to as the LT protocol from here on. It is related to other solutions that have appeared in the literature [Lam78,CASD86] and will be used as a building block later on. The basic technique will be to delay a message until messages with at least as large a timestamp has been received from every other process in the system. However, since this would only work if every process sends an infinite stream of multicasts, a channel flushing mechanism is introduced to avoid potentially unbounded delays.

Say that the channel from process p_j to p_i has been *flushed at time* LT(m) if p_i will never receive a message m' from p_j with $LT(m') < LT(m)$. Flushing can be achieved by *pinging*. To ping a channel, p_i sends p_j a timestamped inquiry message inq, but without first incrementing $LT(p_i)$. On receiving an inquiry p_j, as usual, sets $LT(p_j) = max(LT(p_j), LT(\text{inq}))$ and replies with an **ack** message containing $LT(p_j)$, without modifying $LT(p_j)$. On receiving the **ack** p_i, as usual, sets $LT(p_i) = max(LT(p_i), LT(\textbf{ack}))$. If no new messages are being multicast, pinging advances $LT(p_i)$ and $LT(p_j)$ to the same value.

The protocol is as follows:

[1] This hypothesis is actually used only in the VT delivery protocol.

1. Before sending message m, process p_i increments $LT(p_i)$ and then timestamps m.
2. On receiving message m, process p_j sets $LT(p_j) = max(LT(p_j), LT(m))$. Then, p_j delays m until for all $k \neq i$, the channel between p_j and p_k has been flushed for time $LT(m)$. p_j does not delay messages received from itself.
3. If m has the minimum timestamp among messages satisfying (2), m may be delivered.

To prove that causal delivery is achieved, consider two messages such that $send(m_1) \rightarrow send(m_2)$, and hence $LT(m_1) < LT(m_2)$. There are two cases:

1. *The same process sends m_1 and m_2.* Because communication is FIFO, m_1 will be received before m_2, and because $LT(m_1) < LT(m_2)$, condition 3 guarantees that m_1 will be delivered before m_2.
2. *Different processes send m_1 and m_2.* According to condition 2, m_2 can only be delivered when all channels have been flushed for $LT(m_2)$. As communication is FIFO, and $LT(m_1) < LT(m_2)$, it follows that m_1 has been received. Condition 3 then guarantees that m_1 will be delivered before m_2.

The communication cost, however, is high: $2n - 3$ messages may be needed to flush channels for every message delivered, hence to multicast one message, $O(n^2)$ messages could be transmitted. For infrequent multicasting, this cost may well be tolerable; the overhead would be unacceptable if incurred frequently. However, in place of pinging, processes can periodically multicast their logical timestamps to all other group members. Receipt of such a multicast flushes the channels: at worst, a received message will be delayed until the recipient has multicast its timestamp and all other processes have done a subsequent timestamp multicast. The overhead of the protocol can now be tuned for a given environment.[2]

3.4.2 *VT* protocol

A much cheaper solution can be derived using vector timestamps; we will refer to this as the *VT* protocol. The idea is basically the same as in the *LT* protocol, but because $VT(m)[k]$ indicates precisely how many multicasts by process p_k precede m, a recipient of m will know precisely how long m must be delayed prior to delivery; namely, until

[2] Readers familiar with the Δ-T real-time protocols of [CASD86] will note the similarity between that protocol and this version of ours. Clock synchronization (on which the Δ-T scheme is based) is normally done using periodic multicasts [ST87]. This modification recalls suggestions made in [Lam78], and makes logical clocks behave like weakly synchronized physical clocks. Clock synchronization algorithms with good message complexity are known, hence substitution of a Δ-T based protocol for the logical clock-based protocol in our "combined" algorithm, below, is an intriguing direction for future study.

$VT(m)[k]$ messages have been delivered from p_k. Since → 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 m until

   $$VT(m)[i] = VT(p_j)[i] + 1$$
   $$\forall k \neq i: \ VT(m)[k] \leq VT(p_j)[k]$$

   Process p_j need not delay messages received from itself.
3. When a message m is delivered, $VT(p_j)[i]$ is incremented (this is simply the vector time update protocol from Section 3.3).

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 1.

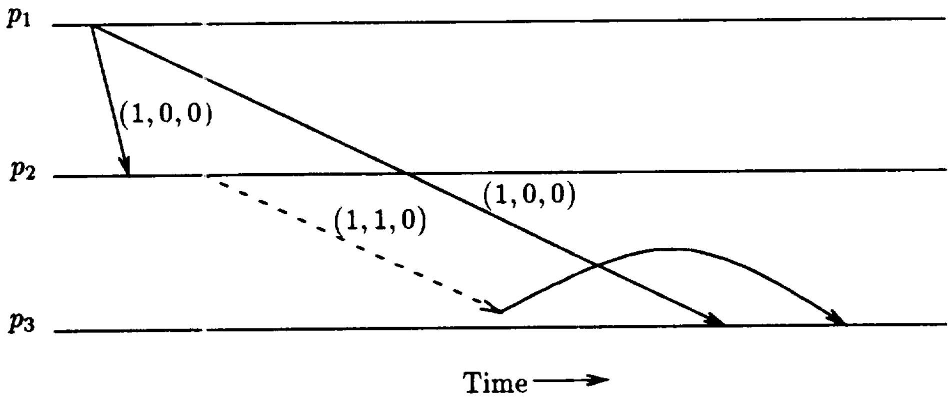

Figure 1: Using the VT rule to delay message delivery

The correctness of the protocol will be proved 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).

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

Case 1. m_1 and m_2 are both transmitted by the same process p_i. Recall that we assumed a lossless, sequenced communication system, hence p_j receives m_1 before 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 $send(m_1) \rightarrow send(m_2)$, 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 $send(m_1) \rightarrow send(m)$. 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 that 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:

$VT(m)[i] \neq VT(p_j)[i] + 1$, or

$\exists k \neq i: \; VT(m)[k] > VT(p_j)[k]$

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

1. $VT(m)[i] \neq VT(p_j)[i] + 1$, that is, m is not the *next message* to be delivered from p_i from p_j. 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 with $VT(m')[i] = VT(p_j)[i] + 1$. If m' is also delayed, it must be under the other case.

2. $\exists k \neq i : VT(m)[k] > VT(p_j)[k]$. Let $n = VT(m)[k]$. The n'th 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. □

4 Extensions to the basic protocol

Neither of the protocols in Section 3 is suitable for use in a virtually synchronous setting with multiple process groups and dynamically changing group views. This section first extends the simple VT **CBCAST** protocol of Section 3.4.2 into one suitable for use with multiple but static process groups, but arrives at a protocol subject to a significant constraint on what we call the *communication structure* of the system. Then, we show how to combine the protocol with other mechanisms, notably the LT **CBCAST** protocol of Section 3.4.1, to overcome this limitation. We arrive at a powerful, general solution.

4.1 Transmission limited to within a single process group

The first extension to the VT protocol is concerned with processes that multicast only within a single process group at a time. This problem is clearly trivial if process groups don't overlap, a property that can be deduced at runtime (see Section 4.4.4). On the other hand, we have assumed that overlap will not be uncommon. Such scenarios motivate the series of changes to the algorithm presented in this section and the ones that follow.

The first change is concerned with processes that belong to multiple groups, e.g. a process p_i belongs to groups g_a and g_b, and multicasts only within groups. Multicasts sent by p_i to g_a must be distinguished from those to g_b, since a process p_j belonging to g_b and not to g_a that receives a message with $VT(m)[j] = k$ will otherwise have no way to determine how many of these k messages were sent to g_b and hence precede m causally. This leads us to extend the single VT clock to multiple VT clocks; VT_a is the logical clock associated with group g_a, and $VT_a[i]$ thus counts multicasts by process p_i to group g_a.[3] 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 VT protocol. 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 $\{g_1, ..., g_n\} \equiv G_j$. Step 2 can be replaced by the following rule:

[3] Clearly, if p_i is not a member of g_a, then $VT_a[i] = 0$, thus allowing a sparse representation of the timestamp. For clarity, we will continue to represent 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.

2' On reception of message m from $p_i \not\equiv p_j$, 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' $\forall g: (g \in G_j): VT_g(m) \leq VT_g(p_j)$.

As above, p_j does not delay messages received from itself.

Figure 2 illustrates the application of this rule in an example with four processes into groups identified as $p_1 ... p_4$. Processes p_1, p_2 and p_3 belong to group G_1, 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.

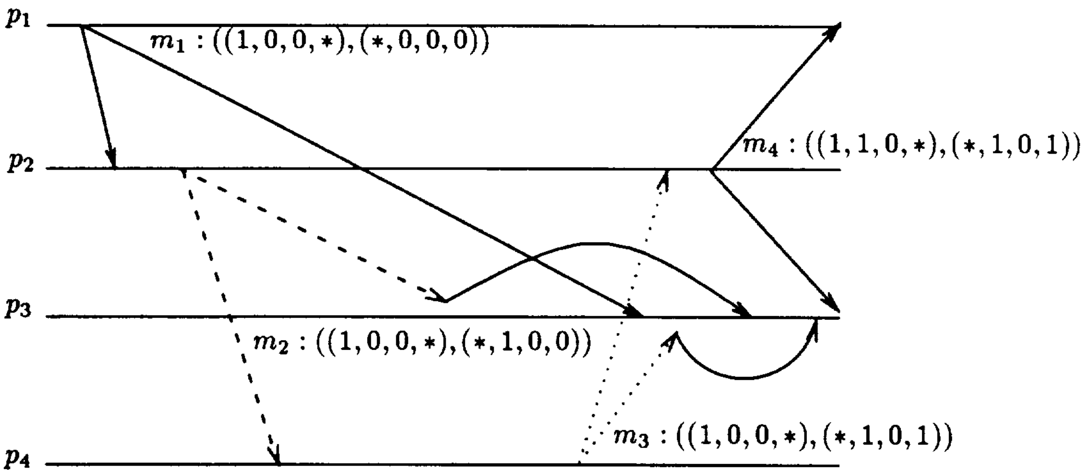
p_1
$m_1 : ((1,0,0,*),(*,0,0,0))$
p_2
$m_4 : ((1,1,0,*),(*,1,0,1))$
p_3
$m_2 : ((1,0,0,*),(*,1,0,0))$
$m_3 : ((1,0,0,*),(*,1,0,1))$
p_4

Figure 2: Messages sent within process groups. $G_1 = \{p_1, p_2, p_3\}$ and $G_2 = \{p_2, p_3, p_4\}$

The proof of Section 3 adapts without difficulty to this new situation; we omit the nearly identical argument. One can understand the modified VT 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, and it was assumed that all communication is done within groups.

4.2 Use of partial vector timestamps

Until the present, we have associated with each message a vector time or vector times having a total size determined by the number of processes and groups comprising the application. Although such a constraint arises in many published **CBCAST** protocols, the resulting vector sizes would rapidly 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.

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 [4] 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. [5] 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 2, message 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 – such cyles will either be simple cycles of length 2, or complex cycles.

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.

Lemma 1: *If a system has an acyclic communication structure, each process in the system only maintains and multicast the* VT *timestamps of groups to which it belongs.*

Notice that under this lemma, the overhead on a message is limited by the size and number of groups to which a process belongs.

We 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.

```
        m1         m2         m3          mk-1       mk
   p1 ----> p2 ----> p3 ----> .... ----> pk ----> pk+1
        g1         g2         g3          gk-1       gk
```

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,

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

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

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 $m_i \xrightarrow{p_j} m_j$ mean that p 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 *k'th* message transmitted by process p_i in group g_a. We will write $m_i \xrightarrow{p} m_j$ 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 $m_i \xrightarrow{p_j} m_j$, where p_j is the sender of m_j.

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 $m_1 ... m_k$ 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.

Proof of Lemma 1: The proof is by induction on l, the length of the message chain $m_1 ... m_k$. 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 3. 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 $m_{r-1} \xrightarrow{p_r} m_r \xrightarrow{p_{r+1}} m_{r+1}$. 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. □

We now proceed to prove the main theorem.

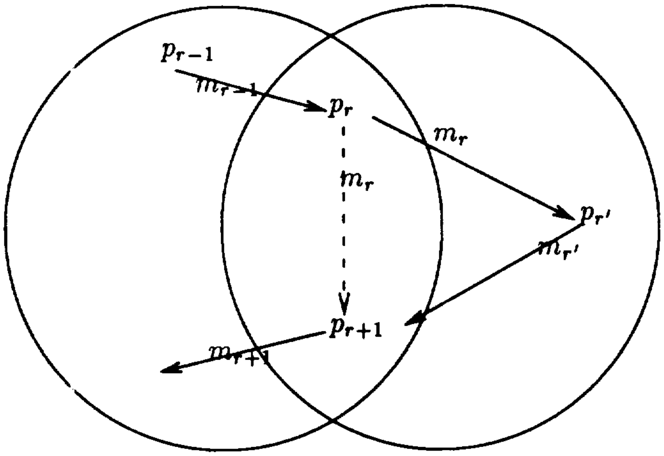

Figure 3: Causal Reversal

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 1, 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.

```
        m1            mc         mc+1
   p1 ---->  ... p2 ----> p3 ---->
        g1            gc         g1
```

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_c arrives at p_3, it will carry a copy of VT_{g1} that indicates that m_1 was sent. This means that m_c will not be delivered at p_3 until m_1 has been delivered there. So m_{c+1} will not be transmitted by p_3 until m_1 has been delivered there. Thus $m_1 \xrightarrow{p_3} m_{c+1}$. 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 1, completing the proof. □

Theorem 1 shows us what timestamps are sufficient in order 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* VT *timestamps corresponding to groups in the biconnected component of CG to which p_i belongs.*

4.3 Extensions to arbitrary communication structures

In general, managing information concerning the biconnected components of CG may be difficult, especially in a dynamic environment. We believe that the most practical use of the above result is in the acyclic case, since a process can conservatively determine that it is not in any cycle by observing that the group of which it is a member overlaps with at most one other group – a completely local test (but see also Section 4.4.4). Consequently, although all our results generalize, the remainder of the paper focuses on the acyclic solution, and we initially implemented only the acyclic solution in ISIS. In this section, we give two protocols that work in more general communication structures. The first protocol does not use any knowledge about the communication structure, but it sometimes imposes delays on message multicasting. The second protocol does use knowledge about the communication structure, but does not impose delays on message multicasting. We then extend both protocols to arbitrary dynamic communication structures.

4.3.1 Conservative solution

Our first solution is denoted the *conservative protocol.* Each multicast m is followed by a second multicast `terminate(m)` signifying that m has reached all of its destinations. The sender of a multicast will normally know when to send the `terminate` as a side-effect of the protocol used to overcome packet loss. The `terminate` message may sent as a separate multicast, but it can also be piggybacked on the next **CBCAST** sent to the same group. A `terminate` message is not itself terminated.

We will say that a group is *active for process p*, if:

1. p is the initiator of a multicast to g that has not terminated, or
2. p has received an unterminated multicast to g, or
3. p has delayed the local delivery of a multicast to g (sent by some other process p').

Note that this is a local property; i.e. process p may compute whether or not it is active for some group g by examining its local state. The *conservative multicast rule* states that a process p may multicast to group g iff g is the only active group for process p or p has no active groups. Multicasts are sent using the VT protocol, as usual. Notice that this rule imposes a delay only when two causally successive messages are sent to *different* groups. The conservative solution could be inefficient, but yields a correct VT protocol. However, the overhead it imposes could be substantial if processes multicast to several different groups in quick succession, and it is subject to potential starvation (this can, however, be overcome).

The conservative solution will work correctly even if group membership changes dynamically.

For brevity, we omit the correctness proof of this solution. The key point is that if p multicasts m to g_2 after g_1 has ceased to be active, then there are no undelivered multicasts m' in g_1 s.t. $m' \rightarrow m$. This can be demonstrated by showing that if g_1 is no longer active and $m' \rightarrow m$, then m' has terminated.

4.3.2 Excluded Groups

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 VT protocol for *most* communication within groups. If there is some g' such that $(g, g') \in X$ we will say that group g is an *excluded group* and some multicasts to or from g will be done using one of the protocols described below.

Keeping track of excluded groups could be difficult; however it is easy to make pessimistic estimates (and we will derive an 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 and the two that follow develop solutions for arbitrary communication structures, assuming that some method such as the previous is used to safely identify excluded groups.

4.3.3 Combining the VT and LT protocols

Recall the LT multicast protocol presented in Section 3. The protocol was inefficient, but required that only a single timestamp be sent on each message. Here, we run the LT and VT protocols simultaneously, piggybacking on each message both LT and VT timestamps, and apply a unified version of the LT and VT delivery schemes on receipt. The LT timestamp is *not* incremented on every broadcast; it is only incremented on certain broadcasts as described below. This greatly reduces the number of extra messages that would be induced by the basic LT algorithm.

Say that m is to be multicast by p to group g. We say that p is *not safe in g* if:

- The last message p received was from some other group g'.
- 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 both its' LT timesamp and its' VT timestamp before multicasting a message to g. Otherwise, it will just increment its' VT timestamp. A message is delivered when it is deliverable according to both the LT delivery rule and the VT delivery rule.

Notice that the pinging overhead of the LT protocol is incurred only when logical clock 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 $LT(p)$ unchanged, p will ping each neighbor processes at most once. Clocks will rapidly stabilize at the maximum existing LT value and pinging will then cease.

Theorem 3: *The combined VT-LT protocol will always deliver messages correctly in arbitrary communication structures.*

Proof: Consider an arbitrary message chain where the first and last messages have overlapping destinations. Without loss of generality, we will assume that $g_1 ... g_k$ are distinct. We wish to show that the last message will be delivered after the first at all such destinations.

```
      m1        m2        m3           mk-1       mk
p1 ----> p2 ----> p3 ----> .... ----> pk ----> pk+1
      g1        g2        g3           gk-1       gk
```

If none of $g_1 ... g_i$ is an excluded group, then, by Lemma 1, m_1 will be delivered before m_k at all overlapping destinations. Now, if some group g_i is excluded, two cases arise - either the last group, g_k is excluded, or some other group is excluded. If g_k is excluded, then p_k will increment its LT timestamp at some point between delivering m_{k-1} and sending m_k. If some other group g_i is excluded, $i < k$, then p_{k+1} will increment its LT timestamp between delivering m_k and sending m_{k+1}. So the LT timestamp of m_k will always be greater than the LT timestamp of m_1, and m_k will be delivered after m_1 at all overlapping destinations. □

4.4 Dynamic membership changes

We now consider the issue of dynamic group membership changes when using the combined protocol. This raises several issues that are addressed in turn: virtually synchronous addressing when joins occur, initializing VT timestamps, atomicity when failures occur, and the problem of detecting properties of CG at runtime, such as when a process determines that its' group adjoins at most on one other and hence always uses the acyclic VT protocol.

4.4.1 Joins

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. Consider a process group g in group view $view_i(g)$. Say that a new view $view_{i+1}(g)$ now becomes defined. There are two cases: $view_{i+1}(g)$ could reflect the addition of a new process, or it could reflect the departure (or failure) of a member. Assume initially that view changes are always due to adding new processes (we handle failures in Section 4.4.3). We will flush communication by having all the processes in $view_{i+1}(g)$ send a message "`flush i+1 of g`", to all other members. During the period after sending such messages and before receiving such a flush message from all members of $view_{i+1}(g)$ 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 members of g under view $i + 1$, it will first have received any messages sent in view i. It follows that all communication sent prior to and during the flush event was done using VT timestamps corresponding to $view_i(g)$, and that all communication subsequent to installing the new view is sent using VT timestamps for $view_{i+1}(g)$. This establishes that multicasts will be virtually synchronous in the sense of Section 2.

4.4.2 Initializing VT fields

Say that process p_j is joining group g_a. Then p_j will need to obtain the current VT values for other group members. Because p_j participates in the flush protocol, this can be achieved by having each process include its VT value in the flush message. p_j will initialize $VT_a[i]$ with the value it receives in the flush message from p_i; p_j initializes $VT_a[j]$ to 0.

4.4.3 Failure atomicity

What about the case where some member of g *fails* during an execution? $view_{i+1}(g)$ will now reflect the departure of some process. Assume that process p_j has received a message m that was multicast by process p_i. If p_i now fails before completing its multicast, there may be some third process p_k that has not yet received a copy of m. To solve this problem, p_j must retain a copy of all delivered messages, transmitting a copy of messages initiated by p_i to other members of $view(g)$ if p_i fails. Processes identify and reject duplicates.

Multicasting now becomes the same two-phase protocol needed to implement the conservative rule. The `terminate` message indicates which messages may be discarded; it can be sent as a separate message or piggybacked on some other multicast.

On receiving $view_k(g)$ indicating that p_i failed, p_j runs this protocol:

1. Close the channel to p_i.

2. For any unterminated multicast m initiated by p_i, send a copy of m to all processes in $view_k(g)$ (duplicates are discarded on reception).

3. Send a `flush` message to all processes in $view_k(g)$.

4. Simulate receipt of `flush` and **ack** messages from p_i as needed by the channel and view flush protocols, and treat any message being sent to p_i as having been delivered in the conservative protocol (Section 4.3.1).

5. After receiving `flush` messages from all processes in $view_k(g)$, discard any messages delayed pending on a message from p_i.

6. p_j ceases to maintain $VT_g[i]$.

Step 2 ensures atomicity and step 4 prevents deadlock in the VT, LT and the conservative protocol. Step 5 relates to chains of messages $m_1 \rightarrow m_2$ where a copy of m_2 has been received but m_1 was lost in a failure; this can only happen if every process that received m_1 has failed (otherwise a copy of m_1 would have been received prior to receipt of the `flush` message). In such a situation, m_2 will never have been deliverable and hence can be discarded.

This touches on an important issue. Consider a chain of communication that arises *external* to a process group but dependent on a multicast within that group. Earlier, we showed that causal delivery is assured by the acyclic VT protocol, but this assumed that multicasts would not be lost. Instead, say that processes p_1 and p_2 belong to group g_1 and that process p_2 also belongs to g_2. p_1 multicasts m_1 to g_1; p_2 receives m_1 and multicasts m_2 to g_2. Now, if p_1 and p_2 both fail, it may be that m_1 is lost but that m_2 is received by the members of $g1 \cap g_2$ that are still operational.

Several cases now arise, all troubling. Consider a process q that receives m_2. If q receives m_2 prior to running the failure protocol, it will discard it under step 5. If q receives m_2 *after* running the failure protocol, however, it will have discarded the VT field corresponding to p_1. m_2 will not be delayed pending receipt of m_1 and hence will ultimately be delivered, violating causality. (q cannot discard m_2 because it may have been delivered elsewhere.) We thus see that both causality and atomicity could be violated by an unfortunate sequence of failures coincident with a particular pattern of communication, and that the system will be unable to detect that this has occurred.

One way to avoid this problem is to require that processes always use the conservative rule of Section 4.3.1, even if the communication structure is known to be acyclic. In our example, this would prevent p_2 from communicating in g_2 until m_1 reached its destinations. Recall that step 4 of the protocol given above prevents the conservative rule from blocking when failures occur.

An alternative is to accept some risk and operate the system unsafely. For example, a process might be permitted to initiate a multicast to group g only if all of *its own* multicasts to other groups have been delivered to at least one other destination process; this yields a protocol tolerant of any single failure. [6]

Given a 1-resilient protocol, the sequence of events that could cause causal delivery to be violated seems quite unlikely. A k-resilient protocol can be built by also delaying receivers; for large k, this reverts to the conservative approach.

We believe that even for a 1-resilient protocol, the scenario in question (two failures that occur in sequence simultaneously with a particular pattern of communication) is extremely improbable. The odds of such a sequence occurring is probably outweighed by the risk of a software bug or hardware problem that would cause causality to be violated for some mundane reason, like corruption of a timestamp or data structure.

Our initial implementation of bypass **CBCAST** uses the conservative solution between all groups; i.e. all groups are excluded. The *VT* protocol is used for communication within a group. This version of ISIS is thus immune to the causality and atomicity problems cited above, but incurs a high overhead if processes multicast to a series of groups in quick succession, which is not uncommon. Our plan is to modify the implementation to use the more optimistic protocols in a 1-resilient manner, but to provide application designers with a way to force the system into a completely safe mode of operation if desired. It should be noted that limitations such as this are common in distributed systems; a review of such problems is included in [BJ89]. We are not alone in advocating a "safe enough" solution in order to increase performance.

4.4.4 Dynamic communication graphs

A minor problem arises in applications having the following special structure:

1. The combined VT-LT protocol is in use.
2. Processes may leave groups other than because of failures (in ISIS, this is uncommon but possible).
3. Such a process may later join 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.

To see this, 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

[6] When using a transport facility that exploits physical multicast such a message will most often have reached *all* of its destinations.

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.

This problem can be overcome in the following manner. Associate with each group a counter of the number of other groups to which it has ever been adjacent; this requires only a trivial extension of the flush protocol. Moreover, say that even after a process p leaves a group g_1, it reports itself as a one-time member of g_1. If p joins some group g_2, the adjacency count for g_2 will now reflect its prior membership, and if a causal chain could possibly arise, multicasts will be under the exclusion rule. Clearly, this solution is conservative and could be costly. On the other hand, say that it is known that all multicasts terminate within some time delay σ. Then one could decrement the adjacency counter for a group after a delay of σ time units without risk. In ISIS, a reasonable value of σ would be on the order of 2-3 seconds.

We have developed more sophisticated solutions to this problem, but omit these because the issue only arises in a small class of applications, and the methods and their proofs are complex.

4.4.5 Recap of the extended protocol

In presenting our algorithm as a basic scheme to which a series of extensions and modifications were made, we may have obscured the overall picture. We conclude the section with a brief summary of the protocol as we intend to use it in ISIS.

The protocol we ultimately plan to use in ISIS is the acyclic VT solution combined with the LT protocol. This protocol piggybacks an LT timestamp and a list of VT timestamps on each message, one *VT* vector for each group to which the sender of the message belongs. In addition to the code for delaying messages upon reception, the protocol implements the channel– and view–flush and terminate algorithms.

Under most conditions the ISIS system will be operated conservatively, excluding groups adjacent to more than one neighboring group. As noted above, neighboring groups can be counted by piggybacking information on the view–flush protocol. Looking to the future, we expect to develop ISIS subsystems that will have special *a-priori* knowledge of the communication structure. These subsystems will make use of an ISIS system call **pg_exclude(gname, TRUE/FALSE)** to indicate the exclusion status of groups. We curently have no plans to develop sophisticated communication topology algorithms for ISIS.

The initial ISIS implementation consists of the VT scheme and the conservative rule, together with the view–flush and terminate protocols. We expect to add the LT extension shortly; the necessary code is small compared to what is already running.

5 Other communication requirements

In this section we consider some minor extensions of the protocol for other common communication requirements.

5.1 A Bypass ABCAST protocol

Readers may wonder if the bypass **CBCAST** protocol can be extended into a fast **ABCAST** mechanism. **ABCAST** is a totally ordered communication protocol: all destinations receive an **ABCAST** message in a single, globally fixed order.

The answer to this question depends on the semantics one associates with **ABCAST** addressing. One way to define **ABCAST** is to say that two **ABCAST**'s to the same *logical address* will be totally ordered, but to make no guarantees about ordering for **ABCAST** messages sent to different addresses. A more powerful alternative is to say that regardless of the destination processes, if two **ABCAST**'s overlap at some set of destinations, they are delivered in the same order. Although ISIS currently supports the latter approach, it is far easier to implement a bypass **ABCAST** with the weaker delivery semantics; the resulting protocol resembles the one in [CM84]. This is in contrast with bypass **CBCAST**, which *always* achieves causal ordering.

Associated with each view $view_i(g)$ of a process group g will be a *token holder* process, $token(g) \in view_i(g)$. If the holder fails, the token is automatically reassigned to a live group member using any well-known, deterministic rule. Assume that each message m is uniquely identified by $uid(m)$.

To **ABCAST** m, a process holding the token uses **CBCAST** to transmit m. If the sender is not holding the token, the **ABCAST** is done in stages:

1. The sender **CBCAST**'s a `needs-order` message containing m. [7] Processes receiving this message delay delivery of m.

2. If a process holding the token receives a `needs-order` message, it CBCAST's a `sets-order` message giving a list of one or more messages, identified by *uid*, and the order in which to deliver them, which it may chose arbitrarily. If desired, a new token holder may also be specified in this message.

3. On receipt of a `sets-order`, a process notes the new token holder and delivers delayed messages in the specified order.

4. On detection of the failure of the token holder, after completing the flush protocol, all processes sort pending ABCAST's and deliver them in any consistent order.

[7] It might appear cheaper to forward such a message directly to the token holder. However, for a moderately large messages such a solution will 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.

This protocol is essentially identical to the replicated data protocol proved correct in [BJ89,Sch88]. Step 4 is correct because the flush ensures that any `set-order` messages will have been delivered atomically, hence all processes will have the same enqueued messages which they deliver immediately before installing the new view.

The cost of doing a bypass **ABCAST** depends on the locations where multicasts originate and 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 randomly and the token is not moved, the cost is $1 + 1/k$ **CBCAST**'s per **ABCAST**, where we assume that one `set-order` message is sent for ordering purposes once for every k **ABCAST**'s. This represents a major improvement over the existing ISIS **ABCAST** protocol. However, because bypass **ABCAST** achieves a weaker form of ordering, it might require changes to existing ISIS applications. We have not yet decided whether to make it the default.

5.2 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 and RPC interactions, are also common. In both cases, causal delivery is desired. Here, we consider the case of point-to-point messages sent by a process p within a group G to which p belongs.

A straightforward way to incorporate point-to-point messages into our VT protocol is to require that they be acknowledged and to inhibit the sending of new multicasts during the period between when such a message is transmitted and when the acknowledgement is received (in the case of an RPC, the reply is the acknowledgement). The recipient is not inhibited, and need not keep a copy of the message. A point-to-point message is timestamped using the sender's logical and vector times, and delivered using the corresponding delivery algorithms, but neither timestamp is incremented prior to transmission. In effect, point-to-point messages are treated as 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.

5.3 Subset multicasts

Some ISIS applications form large process groups but require the ability to multicast to subsets of the total membership. Our protocol is easily extended into one supporting subset multicast, and our initial ISIS implementation supports this as an option. When enabled, a VT vector timestamp of length sn is needed for a group with s senders and n members.

For example, a stock brokerage might support a quote dissemination service with two or three transmitters and hundreds of potential recipients. Rather than form a subgroup for each stock (costly approach if there are many stocks), each multicast could be sent to exactly those group members interested in a given quote. We omit the details of the subset multicast extension.

6 Performance and transport protocol selection

In this section, we discuss the performance of our protocol. We show that the performance of the bypass protocol will be largely dominated by the performance of the underlying layer that is simply concerned with moving data from one site to others. We discuss the design of some alternatives for this layer, which we are currently implementing.

6.1 Complexity and overhead of the protocol

Implementation of the bypass protocol was straightforward in ISIS, requiring less than 1300 lines of code out of the total of 52,000 in the protocol layer of the system. Extensions to support the LT protocol will add little additional code. Initial measurements of performance demonstrate a five to tenfold speedup over the prior ISIS protocols.

Our protocol has an overhead of both space and messages transmitted. The size of a message will be increased by the vector time fields it carries; as noted above, the number of such vectors is determined by the total cardinality of the groups to which the sender belongs directly, and hence will be small. The number of overhead messages sent will depend on the number of non-piggybacked `terminate` messages sent by the conservative protocol and, when implemented, the frequency of LT pinging. In ISIS, LT pinging is expected to be rare and terminate messages are always piggybacked on a subsequent **CBCAST** unless communication in a group quiesces. (As noted before, LT overhead can be bounded using a periodic protocol, if necessary).

We believe that latency, especially when the sender of a multicast must delay before continuing computation, is the most critical and yet unappreciated form of overhead. Delays of this form are extremely noticable. In many systems, there is only one active computation at a given instant in time, or a single computation that holds a lock or other critical resource. Delaying the sender of a multicast may thus have the effect of shutting down the the entire system. In contrast, the delay between when a message is sent and when it reaches a *remote* destination is less relevant to performance. The sender may be delayed in two ways: if the transmission protocol itself is computationally costly, or if a self-addressed multicast cannot be delivered promptly because it is unsafe to do so. Defined in this sense, our method imposes latency on the sender of a multicast only in the conservative protocol, and only when a process switches from multicasting in one group to another, or needs to communicate in one group after receiving in another. Otherwise, the protocol is totally asynchronous. Latency on the transport side is less critical. The dominant source of transport latency is LT pinging, and we plan to quantify this effect by instrumenting ISIS and using simulations.

6.2 Implementation

An interesting feature of the bypass facility is that it assumes very little about communication between processes, and communicates in an extremely regular manner. Specifically, the protocol we ended with sends or multicasts only within groups to which a sending process belongs, and requires only that inter-process communication be sequenced and lossless. The idea of providing an interface by which the bypass multicast protocols could run over a lower-layer protocol provided by the application appealed to us, and as part of the ISIS implementation of bypass **CBCAST** and **ABCAST**, we included an interface permitting this type of extension. We call this lower layer the *multicast transport protocol.* A multicast transport protocol simply delivers messages reliably, in FIFO order, to the groups or processes addressed.

When no special hardware for multicasting is available, the basic ISIS multicast transport protocol is based on UDP (unreliable datagrams). When multicasting hardware is available, ISIS can switch to an experimental multicast transport protocol that takes advantage of such hardware. The remainder of this section details the design, performance and overhead of these multicast transport protocols (in time, size, and messages exchanged per multicast).

6.3 Overhead imposed by the basic VT Protocol

This section breaks down the costs we see in terms of various components of the overhead (create a light weight task, do the I/O, select system call, create the packets, reconstruct them on reception). Figure 4 breaks down the basic CPU costs of sending and receiving messages in our implementation. *These figures are preliminary and will be revised.* These figures are for the combined protocol, but they do not reflect higher level delays that might be imposed by infrequent events such as LT pinging or the view flush. Our figures were derived on a pair of SUN 3/60's doing continuous null RPC's from one to the other. The RPC request was sent in a **CBCAST**; the result returned in a **CBCAST** reply packet. A new lightweight task was created at the receiver to field each RPC request. An ISIS message is fairly complex and allows scatter/gather and arbitrary user-defined and system-checked types. Since no attempt has been made to optimize message data structures for the simple case of a null RPC, this accounts for a a large part of the time spent in the messaging/task layer of the system.

The main conclusion from these measurements is that the **CBCAST** algorithms we derive in this paper are quite inexpensive. Most of the time that a message spends in transit is spent in the lower layers of the system. Clearly, the cost of UNIX messaging is beyond our control, but a great deal can be said about multicast transport.

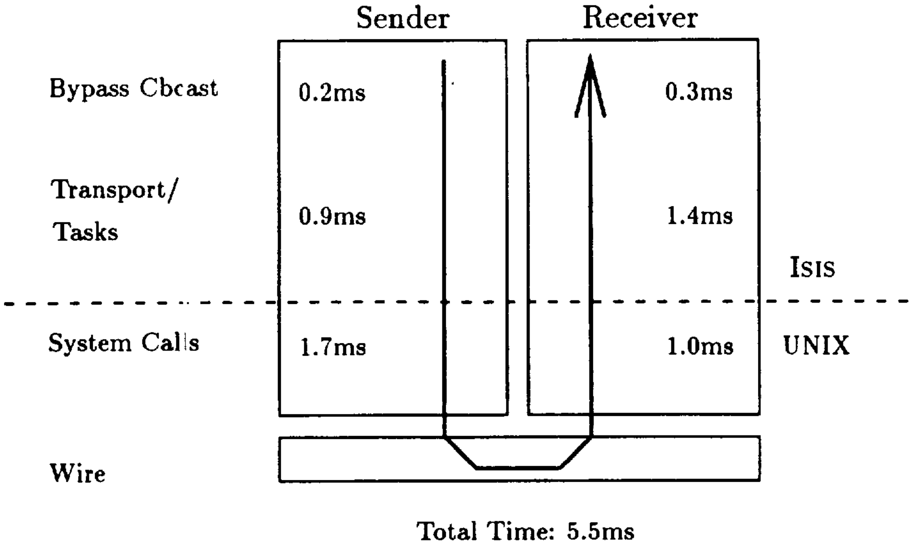

Figure 4: Basic protocol overhead

6.4 Multicast transport protocol selection

The basic Isis multicast transport protocol is designed around a point-to-point model. Each process in a group maintains a two-way reliable data stream with each other process in the group. Whenever possible, acknowledgement information is piggybacked on other packets, such as replies to an RPC or multicast. These streams are maintained independently of each other; for brevity, we omit discussion of such details as flow control and failure detection. This scheme has several advantages; it is relatively easy to understand, as it is based on a well-known communication model. Since it is built on top of unreliable datagrams, it can be easily implemented on any network that provides this service. It has, however, several disadvantages - in particular, it does not scale well. The processing and network transmission costs of communicating with a group rise linearly with the number of processors in the group. In addition, as the number of processes in a group increases, a process sending to the group may experience congestion at the network interface as many acknowledgement or reply packets arrive more or less simultaneously from the other other processes in the group.

We have therefore investigated the design of other multicast transport protocols. An ideal multicast transport protocol would have the following features:

- It would be independent of network topology, but able to take advantage of features of particular networks - e.g. a broadcast subnet.
- The cost of sending a message would be independent of the number of recipients of

that message.

- It would work efficiently for both small and large messages.
- It would have low overhead, latency and high throughput.

It is also important to note that frequently a multicast may give rise to many replies directed to the original sender. We call such an occurrence a *convergecast.* This can lead to congestion at the original multicast sender, with many of the replies being lost. To avoid this, a multicast transport protocol should have some sort of mechanism for co-ordinating and reliably delivering multicast replies. Similar considerations may apply to acknowledgements; however acknowledgements need not be as timely as replies - the multicast transport protocol has more freedom to delay them.

Generally speaking, a reliable multicast transport mechanism will be used in two distinct modes. In the first, *stream* mode, one process will multicast a large amount of data to the group before another process wishes to reply. Multicasting is continuous. This usage could arise in, for example, a trading system, where the transport mechanism is being used to disseminate quotes to trading stations. Another example is a replicated file system where a client workstation is writing a file to a group of file servers. In *rpc* mode, many processes multicast replicated rpc's to the group, where each rpc contains relatively little data, and is much more likely to actually require a reply. Multicasts are not continuous, but bursty. This could arise in maintaining and querying a distributed database or maintaining the state of a distributed game. Note that the application using the multicast transport protocol can provide hints as to which mode it thinks it is operating in. Intermediate modes of usage can of course arise; we do not expect them to be common.

Reliable multicast transport protocols may be divided into two classes; those based on positive acknowledgements, and those based on negative acknowledgements. Many previous proposals for reliable multicast transport protocols have been based on negative acknowledgements, including [KTHB89,AHL89,CM84]. (Some of these protocols, in addition to providing reliable transport, also provide transport ordering properties.) This is because the designers of these protocols believed that a positive acknowledgement from each receiving site would be expensive. We do not believe that this is so.

If a process group is largely communicating in *rpc* mode, reply messages will be converging at the sender in any case. These reply messages can carry positive acknowledgements. In addition, if there are many of these reply messages, they should be scheduled by some mechanism to avoid congestion and message loss at the multicast sender. On the other hand, if a group is largely communicating in *stream* mode, the issue of flow control becomes very important. The sender can't send data faster than the slowest process in the group can receive it; in order to avoid packet loss, there will be flow control packets coming back to the sender from each other process in the group. Again, these packets may carry positive acknowledgments, and again, they must be scheduled in order to avoid congestion problems. The protocol has more flexibility in scheduling these packets than in scheduling reply packets, since they do not contain data that needs to be delivered to the higher level.

There are several possible mechanisms for scheduling packets that are converging on the same destination. One scheme is for the original sender to schedule the packets; it will decide how many concurrent acknowledgments or replies it (and the network) can handle. It then schedules each group of acknowledgements. This scheme involves some extra work by the sender; it has the advantage that the sender can control the rate at which the packets come back depending on whether or not his client is waiting for replies.

Other methods involve the receivers co-operating to ensure that they don't send too many packets to the sender at once. One such method basically involves passing one or several tokens around the group, with the holder of a token having the right to send reply or acknowledgement packets to the original sender. If the replies or acknowledgements are small, they can be put on the token itself, which is returned to the sender when it is full. The main problem with this scheme is that the acknowledgement or reply may take a long time to return to the original sender of a message. This can be overcome by using large window sizes, or by using a large enough number of tokens. Another problem is that the overhead of receiving a message is higher, because an acknowledgement token must be received and transmitted also. This can be overcome by having one token acknowledge several messages, and by piggybacking the acknowledgement token wherever possible. A third problem is that the loss of one acknowledgement packet may cause a message to be retransmitted to multiple destinations. We believe that the extra overhead is acceptable, since packet loss should be rare.

Another receiver-scheduled method for handling acknowledgements or replies is simply to have each acknowledgement be returned at some random time by the recipients. This scheme has been extensively analyzed by [Dan89]; the main problem is that in order to avoid congestion at the original sender, the interval from which the random delays must be picked is very long. It is also of course possible to combine several of the above schemes; for example, acknowledgements could be sender-scheduled in small groups; individual acknowledgements within each group could be further randomly delayed.

We are implementing multicast transport protocols with several of the convergecast-avoidance scheduling strategies described above, and will experiment with them as alternatives to the basic ISIS multicast transport protocol. Our implementations are based on the multicast UDP software of [Dee88], which provides a logical unreliable multicast across internets independently of whether the underlying networks support physical multicast. Full details of the design and implementation of these protocols will be found in [Ste90]. We will include performance measurements for the bypass **CBCAST** and **ABCAST** protocols running over these transport protocols in the final version of the paper.

7 Related Work

There has been a great deal of work on multicast primitives. **CBCAST**-like primitives are described in [BJ87,PBS89,VRB89,SES89,LL86] As noted earlier, our work is most closely related to that of Ladkin and Peterson. Both of these efforts stopped at essentially the point we reached in Section 3 arriving a protocols that would perform well within a single small group, but subject to severe drawbacks in systems with large numbers of processes and of overlapping, dynamically changing process groups. Pragmatic considerations stemming from our desire to use the protocol in ISIS motivated us to take our protocol considerably further. We believe the resulting work to be interesting from a theoretical perspective. Viewed from a practical perspective, a causal multicast protocol that scales well and imposes little overhead under typical conditions certainly represents a valuable advance.

ABCAST-like primitives are reported in [CM84,BJ87,GMS89,PGM85]. Our **ABCAST** protocol is motivated by the Chang-Maxemchuck solution [CM84], but is simpler and faster because it can be expressed in terms of a virtually synchronous bypass **CBCAST**. In particular, our protocol avoids the potentially lengthy delays required by the Chang-Maxemchuck approach prior to committing a message delivery ordering. We believe this argues strongly for a separation of concerns in particular, a decoupling process group management from the communication primitive itself.

We note that of the many protocols described in the literature, very few have been implemented, and many have potentially unbounded overhead or postulate knowledge about the system communication structure that might be complex to deduce. This makes direct performance comparisons difficult, since many published protocols give performance estimates based on simulations or measure dedicated implementations on bare hardware. We are confident that the Isis bypass communication suite gives performance fully competitive with any alternative. The ability to extend the transport layer will enable the system to remain competitive even in settings with novel architectures or special communication hardware.

The ability to run the bypass protocols over new transport protocols raises questions for future investigation. For example, one might run bypass **CBCAST** over a transport layer with known realtime properties. Depending on the nature of these properties, such a composed protocol could satisfy both sets of properties simultaneously, or could favor one over the other. For example, the delay of flushing channels suggests that realtime and virtual synchrony properties are fundamentally incompatible, but this still leaves open the possibility of supporting a choice between weakening the realtime guarantees to ensure that the system will be virtually synchronous and weakening virtual synchrony to ensure that realtime deadlines are always respected. For many applications, such a choice could lead to an extremely effective, tuned solution. Pursuing this idea, we see the Isis system gradually evolving into a more modular structure composed of separable facilities for group view management, enforcing causality, transporting data, and so forth.

For a particular setting, one would select just those facilities actually needed. Such a compositional programming style has been advocated by others, notably Larry Peterson in his research on the Psync system.

8 Conclusions

We have presented a new scheme, the ***bypass*** protocol, for efficiently implementing a reliable, causally ordered multicast primitive. Intended for use in the ISIS toolkit, it offers a way to bypass the most costly aspects of ISIS while benefiting from *virtual synchrony*. The bypass protocol is inexpensive, yields high performance, and scales well. Measured speedups of more than an order of magnitude were obtained when the protocol was implemented within ISIS. 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.

Acknowledgements

The authors are grateful to Keith Marzullo, who suggested we search for the necessary and sufficient conditions of Section 4.2, and made many other useful comments. Tushar Chandra found several bugs in our proofs. Mark Wood, Robert Cooper and Shivakant Misra made extensive suggestions concerning the protocols and presentation, which we greatly appreciate.

References

[AHL89] M.Stella Atkins, Garnik Haftevani, and Wo Shun Luk. An efficient kernel-level dependable multicast protocol for distributed systems. In *Proceedings of the Eighth Symposium on Reliable Distributed Systems*, pages 94–101. IEEE, 1989.

[BJ87] Kenneth P. Birman and Thomas A. Joseph. Reliable communication in the presence of failures. *ACM Transactions on Computer Systems*, 5(1):47–76, February 1987.

[BJ89] Ken Birman and Tommy Joseph. Exploiting replication in distributed systems. In Sape Mullender, editor, *Distributed Systems*, pages 319–368, New York, 1989. ACM Press, Addison-Wesley.

[BJKS88] Kenneth A. Birman, Thomas A. Joseph, Kenneth Kane, and Frank Schmuck. *ISIS — A Distributed Programming Environment User's Guide and Reference Manual.* Department of Computer Science, Cornell University, first edition, March 1988.

[CASD86] Flaviu Cristian, Houtan Aghili, H. Ray Strong, and Danny Dolev. Atomic broadcast: From simple message diffusion to Byzantine agreement. Technical Report RJ5244, IBM Research Laboratory, San Jose, California, July 1986. An earlier version appeared in the 1985 Proceedings of the International Symposium on Fault-Tolerant Computing.

[CM84] J. Chang and N. Maxemchuk. Reliable broadcast protocols. *ACM Transactions on Computer Systems*, 2(3):251–273, August 1984.

[Dan89] Peter Danzig. Finite buffers and fast multicast. In *Proceedings of the ACM Conference on Measurement and Modelling of Computer Systems*, Berkeley, California, 1989. ACM SIGMETRICS.

[Dee88] S. Deering. Multicast routine in internetworks and extended lans. In *Proceedings of theSymposium on Communications Architectures & Protocols*, pages 55–64, Stanford, California, August 1988. ACM SIGCOMM.

[Fid88] C. Fidge. Timestamps in message-passing systems that preserve the partial ordering. In *Proceedings of the 11th Australian Computer Science Conference*, 1988.

[GMS89] Hector Garcia-Molina and Annemarie Spauster. Message ordering in a multicast environment. In *Proceedings 9th International Conference on Distributed Computing Sytems*, pages 354–361. IEEE, June 1989.

[KTHB89] M. Frans Kaashoek, Andrew S. Tanenbaum, Susan Flynn Hummel, and Henri E. Bal. An efficient reliable broadcast protocol. *Operating Systems Review*, 23(4):5–19, October 1989.

[Lam78] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, July 1978.

[LL86] Barbara Liskov and Rivka Ladkin. Highly-available distributed services and fault-tolerant distributed garbage collection. In *Proceedings of the Fifth ACM Symposium on Principles of Distributed Computing*, pages 29–39, Calgary, Alberta, August 1986. ACM SIGOPS-SIGACT.

[Mar84] Keith Marzullo. *Maintaining the Time in a Distributed System.* PhD thesis, Stanford University, Department of Electrical Engineering, June 1984.

[Mat89] F. Mattern. Time and global states in distributed systems. In *Proceedings of the International Workshop on Parallel and Distributed Algorithms.* North-Holland, 1989.

[PBS89] Larry L. Peterson, Nick C. Bucholz, and Richard Schlichting. Preserving and using context information in interprocess communication. *ACM Transactions on Computer Systems*, 7(3):217–246, August 1989.

[PGM85] F. Pitelli and Hector Garcia-Molina. Data processing with triple modular redundancy. Technical Report TR-002-85, Princeton University, June 1985.

[Sch88] Frank Schmuck. *The use of Efficient Broadcast Primitives in Asynchronous Distributed Systems.* PhD thesis, Cornell University, 1988.

[SES89] A. Schiper, J. Eggli, and A Sandoz. A new algorithm to implement causal ordering. In *Proceedings of the 3rd International Workshop on Distributed Algorithms, Lecture Notes on Computer Science 392*, pages 219–232. Springer-Verlag, 1989

[ST87] T. K. Srikanth and Sam Toueg. Optimal clock synchronization. *Journal of the ACM*, 34(3):626–645, July 1987.

[Ste90] Pat Stephenson. *Ordered Broadcast in Internets.* PhD thesis, Cornell University, August 1990. To appear.

[VRB89] Paulo Veríssimo, Luís Rodrigues, and Mário Baptista. Amp: A highly parallel atomic multicast protocol. In *Proceedings of theSymposium on Communications Architectures & Protocols*, pages 83–93, Austin, Texas, September 1989. ACM SIGCOMM.