Reliable Communication in the Presence of Failures*†

Kenneth P. Birman‡
Thomas A. Joseph

TR 85-694
July 1985
(Revised August 1986)

Department of Computer Science
Cornell University
Ithaca, NY 14853

* This work was supported by the Defense Advanced Research Projects Agency (DoD) under ARPA order 5378, Contract MDA903-85-C-0124, and by the National Science Foundation under grant DCR-8412582.
† Accepted for publication: ACM TOCS.
‡ Authors' address; Department of Computer Science, Cornell University, Ithaca, NY 14853

Reliable Communication in the Presence of Failures

Kenneth P. Birman and Thomas A. Joseph
Department of Computer Science
Cornell University, Ithaca, New York

We report on the design and correctness of a communication facility for a distributed computer system. The facility provides support for *fault tolerant process groups* in the form of a family of reliable multicast protocols that can be used both in local and wide-area networks. These protocols attain high levels of concurrency while respecting application-specific delivery ordering constraints, and have varying cost and performance that depends on the degree of ordering desired. In particular, a protocol that enforces *causal* delivery orderings is introduced, and shown to be a valuable alternative to conventional asynchronous communication protocols. The facility also ensures that the processes belonging to a fault tolerant process group will observe consistent orderings of events affecting the group as a whole, including process failures, recoveries, migration, and dynamic changes to group properties like member rankings. A review of several uses for the protocols in the *ISIS* system, which supports fault-tolerant resilient objects and bulletin boards, illustrates the significant simplification of higher-level algorithms made possible by our approach.

Categories and Subject Descriptors: C.2.4 **[Computer-Communication Networks]**: Distributed Systems -- *Distributed applications, distributed databases*; C.4 **[Computer Systems Organization]**: Performance of Systems -- *Reliability, availability, and serviceability*; D.4.1 **[Operating Systems]**: Process Management -- *Concurrency, synchronization*; D.4.5 **[Operating Systems]**: Reliability -- *Fault-tolerance*; H.2.2 **[Database Management]**: Physical Design -- *Recovery and restart.*

General Terms: Reliability.

Additional Key Words and Phrases: Reliable broadcast, fault-tolerant process groups, atomic broadcast.

This work was supported by the Defense Advanced Research Projects Agency (DoD) under ARPA order 5378, Contract MDA903-85-C-0124, and by the National Science Foundation under grant DCR-8412582.

Authors' address: Dept. of Computer Science, Cornell University, Ithaca, N.Y. 14853

1. Introduction

This paper presents a set of communication primitives for supporting distributed computations in an environment where failures could occur. We are primarily concerned with *halting* failures, whereby a process stops executing without performing any incorrect actions. Each distributed computation is represented as a set of events operating on a process-state and a partial order on those events, corresponding to the thread of control. The types of events considered include local computations by a process, broadcasts from a process to a set of processes, broadcasts subject to predetermined ordering constraints, process failures, and process recoveries.

Our premise is that event orderings should be subsumed into the communication layer of a distributed system. In addition, since increasing concurrency generally improves performance in distributed systems, we ask how much communication-level concurrency can be achieved while still respecting event ordering constraints specified by the computations. An important feature of our approach is that it enables a process to deduce the event orderings that will be observed by other processes in the system. This simplifies higher-level code and permits distributed computations to be implemented with reduced risk of inconsistent actions being taken. The approach is formulated in the context of *fault-tolerant process groups*, which consist of a collection of processes that are cooperating to perform a distributed computation, and interacting using our communication protocols. In particular, when the term *broadcast* is used below, it refers to the transmission of a message from a process to the members of a process group (and possibly some additional processes), not to all sites or processes in the system, as has often been the case in prior work on broadcast protocols.

An example will illustrate the class of problems that we address here. Consider a process p that is updating a replicated data item maintained by a set of *data managers*. Assume that this update is performed using a *reliable broadcast:* if any data manager receives the broadcast and remains operational, all data managers will receive it. If p fails, a data manager could observe any of several outcomes:

1. The data manager receives the update and then detects the failure.
2. It detects the failure and receives the update later.
3. It detects the failure and the update is not delivered (anywhere).

In an asynchronous system, a data manager may not be able to differentiate between the second and third outcomes in finite time. Moreover, if some data managers experience the first outcome and others the second one, the system must still behave correctly. One way to address problems such as these is for each process to run an agreement protocol to decide on what action to take after it detects a failure [Skeen-a]. This approach could be slow because it is synchronous, and expensive because each process has to run such a protocol. Another possibility is to discard messages that are received by a process after it has learned that the sender has failed. However, inconsistencies may arise if messages are discarded by one process but retained by another one that learns of the failure later. A third alternative, representative of the general approach of this paper, is to construct a broadcast protocol that orders messages relative to failure and recovery events such that these problems do not arise. In the approach we develop here, the data managers would form a fault-tolerant process group. The communication primitives ensure that every data manager experiences the same sequence of events, hence a data manager can perform an update immediately upon receiving the corresponding message. Likewise, it can take a recovery action immediately after detecting a failure, because no other data manager will observe an inconsistent ordering of events.

The remainder of the paper is structured as follows. Section 2 discusses the presumed environment in more detail. Section 3 describes the communication primitives, and Section 4 gives protocols to implement them in a local network. Finally, Section 5 shows how we applied the primitives to a fault-tolerant system that we have implemented at Cornell.

2. System characteristics

The type of distributed system we consider consists of a collection of processes possessing local states and communicating by messages. Processes do not share memory or maintain closely synchronized clocks, although they do have access to timers with which a reasonable notion of "timeout" can be defined. The term *failure* denotes a *halting* failure: a process ceases execution without taking any (visible) incorrect or malicious actions [Schlichting]. No information survives a failure (by fault-tolerance we refer to continued operation in the presence of failures, not recovery from "stable" storage). If the site at which a failed process was executing remains operational, we assume that the failure is detected (e.g. by the operating system) and that any interested parties are notified. On the other hand, if a site crashes, all the processes executing on it fail, and processes at other sites can detect this only by timeouts. The communication system can also fail: it can lose and duplicate messages, or deliver them out of order. Our protocols may block, but do not take erroneous actions, if the system *partitions* into subgroups of sites within which communication remains possible but between which it is degraded or impossible.

In a broader sense, our assumption is that communication networks are structured hierarchically, into clusters of local sites that do not experience internal partitioning, interconnected by long-haul communication links which may fail but can be reestablished rapidly. The protocols given in this paper address the local case first, and then show how it can be extended transparently to cover the full hierarchical setting.

Clearly, failure detection by timeout cannot be more reliable than the underlying communication system: a series of message losses can always mimic a failure. Moreover, the order in which failures are perceived to have occurred may vary from process to process. These observations lead us to adopt a *logical* approach to failure handling, rather than a *physical* one. That is, instead of a process acting directly after it detects a failure, which could lead to inconsistent actions, a protocol is run to reach agreement with other processes that a failure event has occurred, and to order it with respect to other events. This is meaningful because we have the freedom to pretend that events like message delivery took place either before or after the failure, provided that no evidence to the contrary survived it. The basic property of a logical failure is that after a process learns of such an event or observes the relative ordering of such events, it will never communicate with another process whose state is inconsistent with this information. The issue of partitioning is addressed by preventing communication with a cluster when a majority of its sites have failed. This ensures that there is at most a single set of operational sites within a cluster, and that this set is in an internally consistent state.

3. Fault tolerant process groups

In fault-tolerant systems, it is frequently necessary for the members of a group of processes to be able to monitor one another. They can then take actions based on failures, recoveries, or changes in the status of group members. As an example, consider a fault-tolerant server that is implemented using a group of processes as follows. A request for the service is broadcast to all the members of the group. The operational process having the smallest ID responds to the request. For this implementation to function correctly, it is necessary that all the members of the group have the same view of which members are operational, and of the ID's assigned to each member if these can change. Otherwise no member may respond (as may happen if all operational members believe that a failed process with a smaller ID is still operational), or more than one member may do so (if an operational member believes that a process with a smaller ID has failed when it has not). Further, if there has been a change in the status (operational / failed) of a member, it is necessary for all the processes to agree on whether a request should be handled before or after the change in status, so that they may consistently decide on which process should respond to the request. Although these problems could be addressed by running a protocol each time a failure or recovery is suspected, and / or by executing a protocol to reach consensus on the group's state before responding to each request, it would be expensive and complex to do so. A simpler method, described below, is to provide a *process group* abstraction such that changes in the properties of the group (including failures and recoveries of group members) are ordered with respect to ongoing broadcasts.

The notion of structuring a distributed system into sets of cooperating processes is not new. The V system [Cheriton] and CIRCUS [Cooper] both made use of process group (or troupe) mechanisms for this purpose. However, the difficult problems arising when one tries to employ this approach in fault tolerant applications that also employ highly asynchronous or concurrent algorithms have not been addressed in any systematic way.

It is natural to ask whether existing broadcast primitives, such as *atomic broadcast* [Chang] [Cristian] or *reliable broadcast* [Schneider] can be modified to solve this problem. For several reasons this is impractical, although one of our protocols is indeed similar to an atomic broadcast. First, the existing protocols provide for delivery to *all sites* in a distributed system, whereas our focus is on delivery to *just some* processes, several (or all) of which could reside at a single site. In fact, the sender may not even know the set of processes that should receive a message, since this could include the members of a process group that was growing when the broadcast was issued. Secondly, we wish to provide at least one lightweight asynchronous communication primitive. Conventional atomic broadcast protocols provide a globally consistent delivery ordering, for which clients pay a performance penalty. In [Chang] this takes the form of latency while forwarding the message to a process that has permission to establish broadcast orderings, while [Cristian] delays delivery for a period determined from bounds on the accuracy with which clocks are synchronized and on the inter-site message delivery latency. The lightweight protocol we present below, *CBCAST*, involves minimal delivery latency and is heavily used in the systems discussed in Sec. 5. Finally, previous protocols have tended to assume that the network consists of a small set of closely coupled sites, such as nodes on an ethernet. Our work can be used in a hierarchical distributed system as well.

The remainder of this section formalizes the behavior of fault-tolerant process groups by defining three broadcast primitives: *group broadcast* (*GBCAST*), *atomic broadcast* (*ABCAST*) and *causal broadcast* (*CBCAST*). Their individual behavior is first discussed, and then at the end of the section we summarize the composite behavior that they provide. All the broadcast primitives are *atomic*, that is, a broadcast made to a set of processes is eventually received by all operational destinations, even in the presence of failures. We initially assume that the set of destinations is known at the time a broadcast is issued; later we show how to relax this using a *group addressing protocol*. Issues relating to asynchronous use of the protocols are deferred to the end of the section.

The discussion assumes that each broadcast B has a unique identifier, which we denote as $ID(B)$. The process that initiates a broadcast B is denoted $SENDER(B)$, while the set of processes to which B is sent is denoted $DESTS(B)$.

3.1. Using the group broadcast primitive to maintain process group views

A *process group view* (or just *view*) is a snapshot of the membership and global properties of a process group at some (logical) instant in time. This subsection introduces a *group broadcast* primitive, *GBCAST*, that can be used to inform operational group members when another member fails, recovèrs, joins or withdraws voluntarily, or when some other change to a global property of the group occurs. Our goal is to make it possible for each member to maintain a local copy of the view, updating it on reception of a *GBCAST* message, and acting on it directly without needing any further agreement protocols. This requires that the receipt of a *GBCAST* be ordered relative to other events in the same way at each member. Hence, the system "looks" as if reception was indeed simultaneous (provided that members do not compare the wall clock times at which a particular *GBCAST* was delivered).

The group broadcast primitive is invoked as *GBCAST*(*action*, *G*), where *action* describes the event that has occurred (i.e. "p has failed" or "the new member ranking is: ..."). Here, G is a view, and is computed using an iterative address resolution protocol given in Sec. 3.6. Additionally, when a process p fails, the system automatically initiates a *GBCAST*(*"p has failed"*, *G*) on its behalf. If the failure involves only a single process, then this *GBCAST* can be issued by a supervisory process at the site where the failure occurred. If a site crashed, then the software handling failure detection (Sec. 4.1) initiates *GBCAST*'s for any processes at the failed site belonging to process groups at other sites. *GBCAST*'s that transmit failure information will be referred to as *failure GBCAST*'s.

GBCAST satisfies the following ordering constraints. First, the order in which *GBCAST*'s are delivered relative to the delivery of all other sorts of broadcasts (including other *GBCAST*'s) is the same at all overlapping destinations. Additionally, we require that a failure *GBCAST* be delivered after any other messages sent by the failed process. Thus, once a process is observed to fail, it will never be heard from again.

Notice that if process group members record each new view on stable storage before using it, then even if all members of a process group fail, a simple algorithm based on the one in [Skeen-b] can be used to determine the last ones to fail, which are generally the sites with the most up to date recovery information.

We know of only one system that has used a *GBCAST*-like primitive. ADAPLEX, a database system, employs a protocol called *exclude* to order replicated updates in a database system with respect to failure [Goodman], much as *GBCAST* is ordered with respect to other broadcast types. However, no attempt was made to apply this idea in a broader context.

3.2. The atomic broadcast primitive

Consider a set of processes that maintain copies of a replicated data structure representing a queue. If items are inserted into and removed from each copy of the queue in the same order, no inconsistencies will arise among copies. The *ABCAST* primitive is provided for applications such as this, where the order in which data are received at a destination must be the same as the order at other destinations, even though this order is not determined in advance. *ABCAST* is invoked as *ABCAST(msg, label, dests)*, where *msg* is the message to be broadcast, *label* is a string of characters, and *dests* is the set of processes to which the message must be delivered. *ABCAST*'s are atomic: every operational destination receives *msg*, or none does. In addition, if two *ABCAST*'s with the same label have destinations in common, they will be delivered in the same order at all such destinations. The replicated queue described above can thus be implemented by using *ABCAST*'s to broadcast insert or delete instructions to the various copies, using a queue-id for the *ABCAST* label.

An interesting question concerns the behavior of *ABCAST* if a recipient fails immediately after delivering a copy of a message: are the operational destinations required to employ the same delivery ordering as was used by the failed process, or does it suffice for them to just use a mutually consistent order? The protocol we present in Sec. 4 provides only the latter form of consistency, although the only failure scenario that yields a different ordering is improbable. The interested reader may wish to construct this scenario as an exercise and to devise an alternative implementation that avoids this problem. (This would require an additional phase in the protocol.)

3.3. The causal broadcast primitive

For some applications, it is not sufficient that broadcasts are received in the same order at overlapping destinations - it is also necessary that this order be the same as some predetermined one. As an example, consider a computation that first sets copies of a replicated variable to zero and later increments the variable. Here, it is not enough for the two operations to be carried out in the same order at all copies - the increment must always occur second. However, if independent computations were to access such a replicated variable, some other method would normally be used to synchronize the accesses, making it unlikely that both would broadcast updates concurrently. In this case, a consistent delivery order is unnecessary. The *causal broadcast* primitive, *CBCAST(msg, clabel, dests)* is used to enforce a delivery ordering when desired, but with minimal synchronization. Here, *clabel* is a label that can be compared with other *clabels* using a system-wide algorithm, to yield a partial order on *CBCAST*'s. We write $clabel_1 \xrightarrow{c} clabel_2$, if $clabel_1$ and $clabel_2$ are comparable, and $clabel_1$ is less than $clabel_2$. Note that we allow for *clabels* to be incomparable, that is for neither $clabel_1 \xrightarrow{c} clabel_2$ nor $clabel_2 \xrightarrow{c} clabel_1$ to hold. We use $CLABEL(B)$ to represent the *clabel* of broadcast B, and for brevity write $B \xrightarrow{c} B'$ to mean $CLABEL(B) \xrightarrow{c} CLABEL(B')$. An application uses *clabels* to indicate the order in which broadcasts should be delivered.

What constraints do *clabels* place on the order of broadcast deliveries? Some orderings specified by *clabels* are trivially satisfied. For example, if two *CBCAST*'s have no destinations in common, there is no real constraint on the order of message delivery, regardless of how their labels may compare. On the other hand, some specifiable orderings are unenforceable. A *CBCAST* with a *clabel* less than one that has already been delivered clearly cannot be delivered in the desired order. This calls for a restriction on allowable *clabels*. Fortunately, most applications require an order to be enforced between two broadcasts only if the outcome of one could causally affect the other. The notion of *potential causality* in an asynchronous distributed system in which information is exchanged only by transmitting messages is studied in [Lamport]. In such a system, a broadcast B is said to be *potentially causally related* to a broadcast B' only if they were sent by the same process and B' occurred after B, or if B was delivered at $SENDER(B')$ before B' was sent (or there is a chain of such receivers and senders linking B to B'). We restrict labels on $CBCASTS$ to disallow $CLABEL(B')$ from being less than that of $CLABEL(B)$ if they have the same sender and B' is sent after B, or if B was delivered at $SENDER(B')$ before B' was sent.[1] Such orderings cannot be enforced unless the system has knowledge of which future broadcasts a broadcast must wait for and because such information is usually not available anyway, this is not a major restriction.

A broadcast primitive could be designed that orders any two broadcasts that are potentially causally related. This is stronger than necessary, however. Consider a broadcast B made by a process p to update copies of a replicated variable x. Let this be followed by a broadcast B' by p to update copies of y. Even though there is a potential causal relation between B and B' (because B' occurred after B), there may be no real causal relation between them. In this case, there would be no reason to order the delivery of B before that of B', and to unnecessarily order such broadcasts is inefficient because it limits the possible concurrency in the system. The $CBCAST$ primitive uses *clabels* to identify which of the potential causal relationships are significant. Essentially, it orders broadcasts relative to each other if they are potentially causal *and* if the *clabels* indicate that the potential causal relationships are significant.

We now formally define the ordering properties of $CBCAST$'s. Given $\xrightarrow{c}$ as above, let the relation *precedes* between $CBCAST$'s be the transitive closure of the following two relations:

A. B *precedes* B' if $B \xrightarrow{c} B'$ and the same process p sends B before it sends B'.

---

[1]More accurately, if a broadcast *is* labeled in this way, the $CBCAST$ primitive does not guarantee that this order will be observed.

B. *B precedes B'* if $B \xrightarrow{c} B'$ and B is delivered at *SENDER*(B') before B' is sent.

Then *CBCAST*'s have the following properties. They are atomic, and if *B precedes B'*, then B is delivered before B' at any overlapping destination.

The *CBCAST* primitive may seem to be too weak because it cannot enforce orderings that may be desired between broadcasts that are not potentially causal. Consider a process p that instructs a set of devices, "place wine bottles under taps," and a process q that orders, "open taps." Clearly, it is desirable that the first broadcast be delivered everywhere before the second. However, in an asynchronous system in which there is no upper bound on message delivery times, the only way this can be implemented is to require that the devices send q a message when the wine bottles have indeed been placed under the tap. These messages causally relate the broadcast from p to that from q, and *CBCAST*'s can then be used to enforce the desired ordering. In general there will be little or no occasion to order asynchronous broadcasts that are not potentially causal. Thus the *CBCAST* is strong enough for most applications.

The ability to specify a *clabel* permits the *CBCAST* user to exploit the maximum degree of concurrency and asynchrony possible without compromising the correctness of a computation. Note that the accuracy with which *clabels* represent the dependency between broadcasts could limit concurrency: if *B precedes B'*, *CBCAST* will deliver B first even if the semantics of B and B' are such that they are actually independent. On the other hand, if it is impractical to deduce or to concisely represent causal relationships, timestamps generated using a *logical clock* [Lamport] can be substituted for the *clabel* and the arithmetic comparison operator used for $\xrightarrow{c}$. The result is a conservative version of *CBCAST* that respects potential causality. Some of our work uses this weakened version of *CBCAST* despite the loss of concurrency that it entails.

3.4. Additional broadcast primitives

The primitives described above are relatively orthogonal, in that they address different aspects of the ordering problem, although there is a sense in which *ABCAST* is stronger than *CBCAST* because it constrains the delivery order for all events, not just some. Other primitives that might sometimes be useful include *causal atomic broadcast*, which provides a global ordering and also respects causality, and *minimal broadcast*, which provides guaranteed delivery, but without respecting any ordering constraints. A causal atomic protocol could be constructed using the *ABCAST* protocol we give in Sec. 3, with *CBCAST* as an underlying primitive, hence we omit any further discussion of this protocol. Minimal broadcast results when *CBCAST*'s are invoked with *clabels* that violate potential causality.

3.5. Synchronous and asynchronous uses of the primitives.

Many systems employ remote procedure calls (RPC) internally, as a lowest level primitive for interaction between processes. It should be evident that all of our broadcast primitives can be used to implement replicated remote procedure calls as in Cooper [Cooper]: the caller would simply pause until replies have been received from all the participants (observation of a failure constitutes a reply in this case).[2] We term such a use of the primitives *synchronous*, to distinguish it from from an *asynchronous* broadcast in which no replies, or just one reply, suffices.

In the work we report later, *GBCAST* and *ABCAST* are normally invoked synchronously, to implement a remote procedure call by one member on all the members of its process group. However, *CBCAST*, which is the most frequently used overall, is almost never invoked synchronously. Asynchronous *CBCAST*'s are the source of most concurrency in the *ISIS* system: although the delivery ordering is assured, transmission can be delayed to enable a message to be piggybacked on another, or to schedule IO within the system as a whole. While the system cannot defer an asynchronous broadcast indefinitely, the ability to defer it a little, without delaying some computation by doing so, permits load to be smoothed. Since *CBCAST* respects the delivery orderings on which a computation might depend, and is ordered with respect to failures, the concurrency introduced does not complicate higher level algorithms. Moreover, the protocol itself is extremely cheap.

---

[2]Process group members observe the failure of other group members when they receive a failure *GBCAST*. In other situations, a process uses a *monitoring* facility described in Sec. 4.1 to detect failures of other processes.

A problem is introduced by our decision to allow asynchronous broadcasts: the atomic reception property must now be extended to address causally related sequences of asynchronous messages. If a failure were to result in some broadcasts being delivered to all their destinations but others that precede them not being delivered anywhere, inconsistency might result even if the destinations do not overlap. We therefore extend the atomicity property as follows. If process s receives a message m, and subsequently sends a message m' to process t before failing, then unless t fails as well, m must be delivered to its remaining (operational) destinations. This is because the state of t may indirectly depend on m. The costs of the protocols are not affected by this change.

A second problem arises when the pragmatic implications of asynchrony are considered. In the event of a failure, a suffix of a sequence of asynchronous broadcasts could be lost and the system state would still be internally consistent according to the above rule. A process that is about to take some action that may leave an externally visible side-effect will hence need a way to pause until it is guaranteed that asynchronous broadcasts that precede it have actually been delivered. For example, consider a process that asynchronously broadcasts a checkpoint to a set of backup processes. If it fails while the broadcast is still in progress at other sites, it might not be delivered to any backup (c.f. definition of *atomicity*), and the failure handling action would not occur. One way to address this is for a sender to send a message requesting acknowledgements from the destinations, and to wait until the destinations are observed to fail or the acknowledgements are received. Rather than having to do this each time, a **flush** primitive is provided, which blocks its caller until all its pending asynchronous broadcasts have been delivered to their (operational) destinations. Occasional calls to **flush** do not eliminate the benefit of using *CBCAST* asynchronously. Unless the system has built up a considerable backlog of undelivered broadcast messages, which should be rare, **flush** will only pause while transmission of the last few broadcasts completes.

3.6. Group addressing

All of our protocols require that a sender explicitly name the set of destination processes for each broadcast. A problem arises if a sender wishes to broadcast to *all* the members of a process group. If the group grows after the broadcast is initiated but before it is delivered, any new members would not receive it. A way to resolve this is for each process group member to number its process group views sequentially. Any process can then *cache* (possibly out of date) process group membership information and view numbers for groups with which it communicates. To transmit a *GBCAST*, *ABCAST*, or *CBCAST* to all members of a group G, the cached information would be used to compute $DESTS(B)$, and the view number included in the message. On delivery, if a recipient finds that the process group view has changed, it rejects the message. Since all recipients have the same view when they receive the message, they all reject it if any does so. A rejected broadcast can then be retransmitted to an updated set of destinations, and the cache updated. A similar technique was proposed in [Cooper] for communication with process troupes that have been dynamically reconfigured.

The reader may be troubled by the fact that this algorithm does not distinguish between a situation in which a process receives a broadcast and then fails and one in which the process fails first and the broadcast is never delivered to it. In fact, both cases are treated as if atomic delivery occurred in the view that existed prior to the failure, because the resulting system states are indistinguishable. (Recall that a failure results in the loss of all information at a site.) This is consistent with our view of atomic delivery as a logical property rather than a physical one. This approach is very similar to the one discussed in Sec. 2 in connection with site failures.

Some care is needed when updating the cache, to ensure that the *CBCAST* delivery order is preserved. In particular, consider three *CBCAST*'s $A \xrightarrow{c} B \xrightarrow{c} C$, and assume that A and B have been transmitted using incorrect destinations. If the cache is updated promptly after A is rejected, C could be transmitted using the corrected destinations before B is rejected and retransmitted. C might then be delivered before the retransmission of B occurs, which would violate the causal order. This problem is avoided by invoking **flush** before changing the contents of a cache.

For technical reasons that we will discuss in a future paper [Birman-b], an iterated delivery is undesirable when hierarchical process group relationships are supported, as is the case in a system we are currently building. Consequently, our implementation of *GBCAST* is such that the addressing protocol never iterates in the common situation where a member of a process group broadcasts to the other members of the group (as opposed to a broadcast originating outside the group). This is because we lock the view when the group membership is growing, as described in Sec. 4.3.3.

3.7. Checkpointing the group state and transferring state during recovery

A common problem faced by the programmer using fault-tolerant process groups is to manipulate the "current state" of a group, for example when a checkpoint must be made or when initializing a process that wishes to join or recover. Checkpointing is straightforward: any process can issue a *GBCAST* to the membership of the group (including itself), and when this *GBCAST* is received it is safe to make the checkpoint immediately. The resulting checkpoints will be consistent because they are computed at the same instant in (logical) time, relative to other events.

There are two ways to transfer the state to a recovering process. The most straightforward solution is to view the recovery *GBCAST* as a synchronous RPC that returns a state vector (with a state contribution from each current member of the group) and has the side effect of changing the view to include the new member. The state transfer is fault-tolerant because it is so redundant: unless all current members fail, at least one member will return the present state. However, if states can be large, as in a database, this approach is impractical. A more sophisticated mechanism uses what we call a *coordinator / cohort* scheme, and is motivated by our prior work on resilient objects in *ISIS* [Birman-a] [Birman-a]. At the time the recovery *GBCAST* is received, one process group member is designated the coordinator for the state transfer, and the others back it up as its cohorts. The coordinator uses any convenient protocol (perhaps a stream of *CBCAST*'s) to transfer its state to the recovering process, and when the transfer is finished it uses *CBCAST* to send a completion message to the other group members. While the state transfer is underway, the new group member may receive other sorts of messages, since the view will already have changed to include it. It buffers these to be processed after the transfer terminates. If the coordinator fails before completing the transfer, the cohorts will receive a failure *GBCAST*, and one then takes over to restart or resume the transfer.

3.8. Example

Figure 1 illustrates the communication patterns that might arise when two clients communicate with a process group. The figure is drawn to make communication look synchronous, since recipients will generally be programmed as if this were the case. Paths taken by reply messages are not shown because the group mechanism is compatible with a variety of interaction mechanisms. These include "informatory" interactions in which no reply is needed, coordinator-cohort interactions where a single reply is sent on behalf of the group as a whole [Birman-a], and process troupe implementations in which all members reply to each message [Cooper].

4. Implementation of the communication abstraction

This section gives implementations for the communication abstraction described previously, targeted to a collection of computers interconnected by a local network. We first cover the case of a cluster of sites within which communication is assumed to be rapid and partitioning unlikely. We begin by discussing the transformation of the "raw" environment of a typical cluster into one giving very uniform failure and communication behavior at all sites. Next, the *ABCAST*, *CBCAST* and *GBCAST* protocols are given and a garbage collection mechanism is described. Finally, the section ends by addressing cluster interconnection issues. Figure 2 illustrates the overall system structure.

4.1. The inter-site layer

The inter-site communication layer converts halting failures and admissible communication failures (message loss, delayed delivery, and out-of-order delivery) into a *site view* abstraction, defined below. The layer provides two primitives: **send(m, dest)** for sending message *m* to site *dest*, and **status(m)**, which returns *sent* if the destination has acknowledged receipt of the message or if a failure protocol has been started for the destination site, as described below. Intuitively, a message has been *sent* if the future behavior of the system will be consistent with the message having already been delivered.

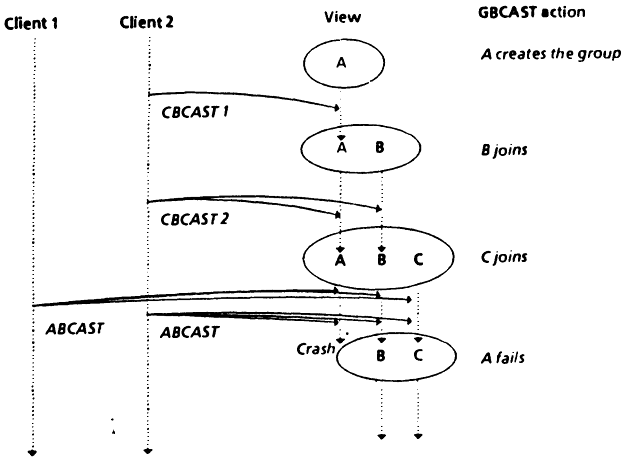

Figure 1: Clients communicating with a process group

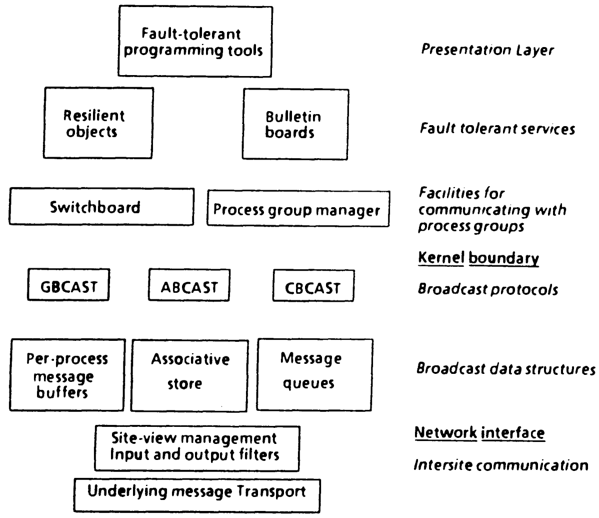

Figure 2: ARCHITECTURE OF THE COMMUNICATION SUBSYSTEM

Processes executing the protocols also use a second interface to the inter-site layer which provides a *process monitoring service*, which a process invokes when it is waiting for a reply from another process. This service watches for changes in site-views,* which can signify the failure of one or more monitored processes, and also interrogates a *process manager* on the site where a process is running if a reasonable delay elapses with no reply from the monitored process. If a failure is detected, the monitor service generates a *failure reply* message on behalf of the failed process, and sends it to the process that requested the monitoring service. The monitor service is integrated into our message delivery subsystem in a way that ensures that any reply sent by the failed process will be delivered before failure messages are generated for it. Throughout the remainder of this section, when we say that a process detects a failure of another process and completes a protocol, we refer to this monitoring mechanism.

The intersite layer employs a windowed acknowledgement protocol for ordered, loss-less site-to-site message transmission. Depending on the properties of the inter-site transport layer, the complexity of this layer will vary. In our current implementation, it consists of a filtering mechanism that is essentially bound to the network device drivers at each site. To detect failures, each site sends a "hello" message to all other sites periodically; if a hello message is not received from a site within a reasonable period, it is assumed to have failed, triggering a protocol to change the site view. If a site is slow to send messages, it may be considered to have failed and forced to run a recovery protocol (the probability of error can be made small by picking a large timeout interval or introducing a protocol phase that allows other sites to prevent execution of the failure algorithm if they believe that the site in question is actually operational). A site *incarnation* number is incremented each time a site recovers; henceforth, the term "site" always means "incarnation of a site." Messages from a failed incarnation are discarded by the input filter, and a *you are dead* message is returned to the sender. Messages addressed to a different incarnation than the current one are similarly discarded.

4.2. Site view management

The site view management layer ensures that each site in the system has a consistent picture of site failures and recoveries occurring in the system. Each site has a *site view*, which is the set of sites it deems to be operational, with their respective incarnation numbers. A site view is changed when a site fails or recovers. A *site view sequence*, denoted V_0, V_1, $\cdots$ is a sequence of site views, reflecting these changes. The site view management protocol described in this subsection ensures that each operational site goes through the same sequence of site views. Later, the protocols take advantage of this to recover from failures without first running any special agreement protocols.

Each site maintains a copy of the site view sequence, initialized in some consistent way when the system cold-starts. The sites in a view can be ordered uniquely according to the view in which they first became operational, with ties broken by site-id. The "oldest" site in this ordering is called the *view manager*, and is responsible for initiating the view management protocol when it detects a site failure or recovery. If a site determines that all sites older than itself have failed, it takes over as the new view manager. Note that the sequence of view managers is a stable property: extensions to the view sequence extend the sequence of managers without changing the subsequence on which sites have already agreed.

The view management protocol is based on a two phase commit protocol. Let $V_0, V_1, \cdots, V_t$ be the current site view sequence.

1. On detecting failures or recoveries, the view manager computes a proposed *view extension* $V_{t+1}, V_{t+2}, \cdots, V_{t+k}$. (If no failures occur during the execution of the protocol, the length of the extension is 1, that is, V_{t+1} contains all the changes to the current site view. Failures occurring during the execution of the protocol may cause the site view sequence to be extended by more than one view, as described shortly). It ceases to accept messages from site-incarnations not in V_{t+k} and sends the proposed view extension to the sites in V_{t+k}.

2. On receiving a proposed view extension, a site first ceases to accept messages from site-incarnations not in V_{t+k}.

   a. If the site has not previously received a proposed extension, or the new one includes all the changes (failures and recoveries) recorded in the old one, the site saves the new proposed extension. Then, it replies to the view manager with a *positive acknowledgement.*

   b. Otherwise, the site has previously received a proposed view extension recording events that are not included in the new one. It replies with a *negative acknowledgement,* giving the events that were missing.

3. The view manager collects acknowledgements.

   a. If all the acknowledgements were positive, it sends a *commit message* for the proposed extension to all sites in V_{t+k}.

   b. If additional failures or recoveries have been detected, or negative acknowledgements were received, the view manager updates its proposed extension and re-executes from step 1.

   If the view manager fails, a new site takes over as view manager, and proceeds as follows:

4. If this new view manager has an uncommitted view extension, the previous view manager may have sent some commit messages before failing. It appends a new site view containing the failure of the old view manager to its pending extension and starts the protocol from step 1.

5. If the new view manager has received a committed extension, and has no pending one, it must assume that some sites did not receive the commit. It appends a new view to the most recently committed extension and continues from step 1. Participants ignore a committed prefix of a proposed extension.

To establish the correctness of the protocol, consider the cases that can arise:

1. If the view manager doesn't fail, all sites obtain the same committed view extensions.
2. If the view manager fails and any site has a committed view extension, than all sites have acknowledged that extension. The new view manager will eventually commit the extension everywhere.
3. If the view manager fails after it has distributed a proposed extension to a subset of sites, and that proposed extension is not known to the new view manager, then any site knowing the extension will send a negative acknowledgement to the new coordinator when the protocol is restarted, and the coordinator will then distribute it during an additional protocol phase.

The following issues arise because sites may detect failures and recoveries of other sites at different times and in arbitrary order. First, the order in which view managers commit site views becomes the order accepted by the system, even if individual sites may have detected failures and recoveries in a different order. Second, a view manager may erroneously decide that a site has failed (because it is slow to respond). In this case, all sites consider the site in question to have failed,[3] and respond to any message from it with a "you are dead" message. Such a site is said to be *killed*, as it is forced to undergo recovery with a new incarnation number. Third, it is possible for a site a to believe that a site b has failed, for b to believe that a has failed, and for each of them to consider themselves as the view manager. In this situation, one or both will be killed, otherwise some site would have to acknowledge two contradictory views from two different view managers, which cannot happen.

[3]This is true unless the network becomes *partitioned*, that is, a group of sites continues to remain operational, but unable to communicate with the other sites. If network partitioning can occur, erroneous actions can be prevented by requiring that sites cease to operate if number of operational sites in a view drops below a quorum.

Notice that if it is desired that the system be able to recover if all sites fail, a simplified protocol based on the one in [Skeen-b] can be run to reconstruct the view sequence from copies saved on non-volatile storage. □

4.3. The protocols

This section gives implementations for *ABCAST*, *CBCAST*, and *GBCAST*, deferring garbage collection issues to Sec. 4.4. The broadcast protocols order messages addressed to a process as necessary, and place them on the *delivery queue* for the process, as illustrated in Fig. 3. A process removes messages from its delivery queue in FIFO order. The other queues are used to buffer messages before they are placed on the delivery queue.

4.3.1. ABCAST protocol

Our *ABCAST* protocol is based on a two-phase protocol by Skeen [Skeen-c]. The protocol maintains a set of priority queues for each process, one for each *ABCAST* label, in which it buffers messages before placing them on the delivery queue. We assume that priority values are integers, with a process-ID appended as a suffix to disambiguate the priorities assigned by different processes. Each message in the buffers is tagged *deliverable* or *undeliverable.* The protocol to implement *ABCAST(msg, label, dests)* is as follows:

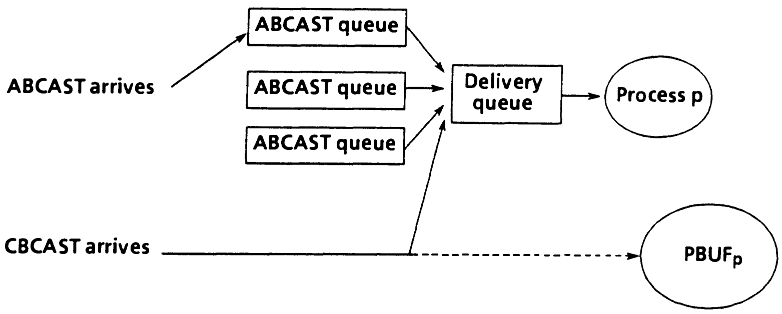

Figure 3: Data structures used by ABCAST and CBCAST primitives

1. The sender transmits *msg* to its destinations.

2. Each recipient adds the message to the priority queue associated with *label*, tagging it as *undeliverable*. It assigns this message a priority larger than the priority of any message that was placed in the queue, with the process-id of the recipient as a suffix. It then informs the sender of the priority it assigned to the message.

3. The sender collects responses from recipients that remain operational. It then computes the maximum value of all the priorities it received, and sends this value back to the recipients.

4. The recipients change the priority of the message to the value they receive from the sender, tag the message as *deliverable*, and re-sort their priority queues. They then transfer messages from the priority queue to the delivery queue in order of increasing priority, until the priority queue becomes empty or the message with the lowest priority is undeliverable. In the latter case, no more messages are transferred until the message at the head of the queue becomes deliverable.

If a failure occurs, any site that has a message tagged *undeliverable* from a failed sender detects this using the monitoring mechanism and can then take over as the new coordinator to complete the protocol. It does so by interrogating participants about the status of the message. A participant being interrogated either has never received the message, or responds with the priority and tag. The new coordinator collects responses. If any process had marked the message *deliverable*, the new coordinator distributes the corresponding priority to the other processes (step 3). Otherwise, it resumes from step 1. Note that this scheme requires that each process retain information about messages even after they are placed on the delivery queue; garbage collection is discussed in Sec. 4.4.

Correctness. The protocol is atomic because before any recipient tags a message as *deliverable*, all destinations must have received copies of it. If a failure occurs after that, a destination that has a copy tagged *undeliverable* will complete the protocol. Thus if the message is delivered at any destination, it will be delivered at all of them.

We now show that every message is delivered in the same order at all overlapping destinations. If the priorities of any two messages were assigned by the same process, they cannot be equal. If they were assigned by different processes, the process-ID that is suffixed can be used to order them, should the priority values be equal. Thus every deliverable message has a *unique* priority assigned to it. Messages addressed to overlapping destinations are delivered everywhere in this order. Because the final priority is the maximum of all assigned priorities, the priority of an *undeliverable* message never becomes smaller than that of a message that has already been delivered. Thus, if the message at the head of the queue is tagged as *deliverable*, it can always be safely delivered. □

4.3.2. CBCAST protocol

Our *CBCAST* protocol operates by ensuring that whenever a message B is sent from a process p to a process q, a copy of every undelivered message B' that *precedes* B (as in Sec. 3.3) is also sent to q with B, even if q is not a destination for B'. Thus a message may travel from process to process before it reaches a destination, and multiple copies could be delivered by different routes (duplicates are discarded). It follows that if a message B is delivered to a process q, then copies of all messages addressed to q that *precede* B also arrive with B, or have arrived earlier. Messages addressed to q can therefore be delivered in order. We first describe a simple but inefficient *CBCAST* implementation, then show how its efficiency may be improved.

For each process p, there is a message buffer BUF_p, which contains copies of messages sent to and from p, as well as copies of messages that arrive at p en route to other processes. Every message B in BUF_p has fields $ID(B)$ and $REM\_DESTS(B)$ associated with it. When p performs an *CBCAST(msg, clabel, dests)*, the message is placed in BUF_p, and $REM\_DESTS(B)$ is initialized to *dests*. If $p \in REM\_DESTS(B)$, a copy of the message is placed on the delivery queue for p, and p is removed from $REM\_DESTS(B)$. The process p can then continue as if the message has already been sent. Messages in BUF_p are later scheduled for transmission to BUF_q, for each destination q. The decision as to when this occurs can be based on advice from higher level algorithms (a message that requires a response would presumably be transmitted as soon as possible to minimize waiting time, while asynchronous messages for which no replies are needed could be delayed longer), or on factors like the load on the network. We assume only that all messages are scheduled for transmission within finite time. For now, we also assume that a copy of any message placed in BUF_p remains in the buffer indefinitely.

A message B is transmitted from BUF_p at site s to BUF_q at site t as follows:

1. A *transfer packet* $<B_1, B_2, \cdots>$ is first created including all messages B' in BUF_p such that $B' \xrightarrow{c} B$ and $REM\_DESTS(B')$ is non-empty. The messages are sorted so that if $B_i \xrightarrow{c} B_j$, then $i < j$.
2. The transfer packet is then transmitted from site s to site t.
3. When the packet has been sent, for each B_i that it contained, q is deleted from $REM\_DESTS(B_i)$, if it was listed there.

When process q receives a packet $<B_1, B_2, \cdots>$, the following is done for each i, in increasing order of i:

4. If $ID(B_i)$ is already associated with a message in BUF_q, then B_i is a duplicate and is discarded.
5. If $q \in REM\_DESTS(B_i)$, B_i is placed on the delivery queue for q, q is removed from $REM\_DESTS(B_i)$, and a copy of B_i is placed in BUF_q.
6. Otherwise, B_i is a message in transit to another process, and it is simply placed in BUF_q.

Correctness. Any process q that receives a message adds a copy of it to BUF_q, unless q is the only remaining destination. Since all messages in BUF_q are scheduled for transmission within finite time, it follows that if any site has received a message and does not fail, the message will eventually be delivered to all the destinations that remain operational. Thus, the protocol is atomic.

To show that messages are delivered in the correct order, it suffices to show that for every pair of messages B and B' delivered to q, if B *precedes* B', then B is placed on the delivery queue before B'. We first prove that a copy of B will have been placed in $BUF_{SENDER(B')}$ when B' was first placed there. Then any transfer packet that contains B' will also contain B, and B will be ordered before B' in it. Thus when the first transfer packet containing B' arrives at q, a copy of B will also be received. If B has not arrived in an earlier packet (in which case it was placed on the delivery queue at that time), B will now be placed on the delivery queue before B'.

It follows from the definition of the relation *precedes* that if B *precedes* B', there is a sequence of $CBCASTS$ $B = B_0, B_1, \cdots, B_n = B'$ such that for all i, $0 < i \leq n$, $B_{i-1} \xrightarrow{c} B_i$, $SENDER(B_i) \in REM\_DESTS(B_{i-1})$, and B_{i-1} is received at $SENDER(B_i)$ before B_i is sent. The proof that a copy of B will have been placed in $BUF_{SENDER(B')}$ when B' is first placed there is by induction on n, the length of the shortest sequence satisfying the properties above. If $n = 0$, $B = B'$, and the result follows immediately. Assume that the hypothesis is valid for $n = k$. If $n = k + 1$, consider the messages B and B_k. By the induction hypothesis, a copy of B will have been placed in $BUF_{SENDER(B_k)}$ when B_k is first placed there. Hence, any transfer packet carrying a copy of B_k will also carry a copy of B. We know that $B_k \xrightarrow{c} B_{k+1}$, $SENDER(B_{k+1}) \in REM\_DESTS(B_k)$, and B_k is received at $SENDER(B_{k+1})$ before B_{k+1} is sent. Hence, a copy of B will arrive at $SENDER(B_{k+1})$, and be placed in $BUF_{SENDER(B_{k+1})}$ before B_{k+1} is delivered. This gives us the required result. □

There are a number of ways in which the protocol above can be optimized:

1. Although the protocol was stated in terms of packets sent from process to process, these packets could be combined to form larger inter-site packets. One inter-site packet could suffice to transfer messages from a set of processes at one site to all destination processes at another. The packet reception rules would be amended to deliver all the messages in a packet that have local destinations at once, and to correctly update the associated *PBUF*'s.

2. Rather than keeping a copy of a message in the buffer of each process at a site, the buffers could contain pointers into a common message pool for all processes at the same site. Then, each message would be represented at most once at each site.

3. To avoid sending a copy of the same message from process p to process q more than once, a field *SENT_TO(B)* can be associated with each message B and updated each time a packet containing the message is sent. The packet generation rules can then be further amended to include B in a packet to a site only if it has not already been sent there. If desired, it would also be possible to transmit *SENT_TO* information from site to site periodically, so that other sites can avoid sending duplicates.

The problem of deleting a message after it has reached all its destinations (*REM_DESTS* becomes empty) is discussed in Section 4.4.

4.3.3. GBCAST protocol

A *GBCAST(action, G)* must be ordered relative to other *GBCAST*'s to G, as well as relative to *ABCAST*'s and *CBCAST*'s. In addition failure *GBCAST*'s must be delivered *after* every message from the failed process. These aspects are treated as separate problems in the description of the protocol, then optimizations yielding a more efficient implementation are given. Fig. 4 shows the additional data structures needed to support the *GBCAST* protocol.

The first part is carried out only for failure *GBCAST*'s, and ensures that all messages from a failed process are ordered before the *GBCAST*. Say that the process that failed is f.

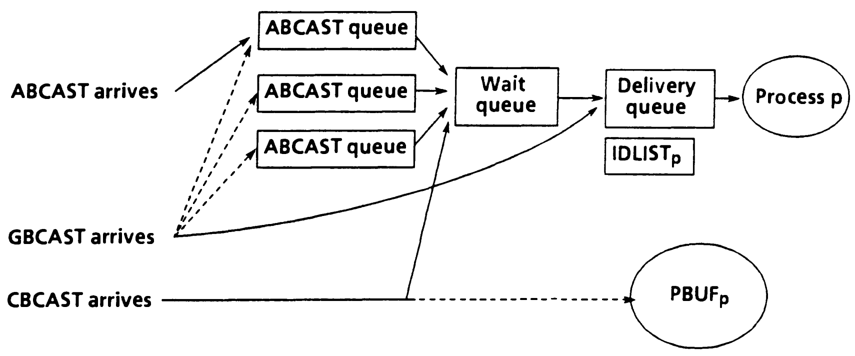

Figure 4: Data structures used during GBCAST protocol

1.1 The process p running the protocol sends a message to *all* processes in the system, informing them of the start of the failure *GBCAST* for f.

1.2 A process q receiving this message schedules for transmission any message B in BUF_q sent by f that includes a member of G in *REM_DESTS*(B). It then waits until the status of these messages turns to *sent*.

1.3 If q belongs to G, q waits until all *ABCAST*'s from f have become deliverable. This will happen eventually because some process (perhaps q itself) will take over to complete the *ABCAST* protocol.

1.4 The process q then sends an acknowledgement to p. This part of the *GBCAST* ends when acknowledgements have been received from all operational processes.

The second part of the protocol is based on the *ABCAST* protocol, and orders *GBCAST*'s to the same group relative to one another, and *GBCAST*'s relative to *ABCAST*'s.

2.1 The process p distributes the message *action* to the members of the process group G.

2.2 A recipient q places copies of the message on *all ABCAST* priority queues, tagging them *undeliverable*. We assume that there is always a (possibly empty) queue for *every* possible *ABCAST* label. It assigns it a priority greater than that of any message that has been placed on *any* of the *ABCAST* queues, and sends this priority value back to p (all copies receive the same priority).

2.3 After collecting the responses, p sends the maximum of all the values it received to the members of G, which change the priority accordingly and resort their queues. Unlike in the *ABCAST* protocol, the messages are not tagged *deliverable* at this time. Thus when a *GBCAST* message reaches the head of a *ABCAST* priority queue, further delivery of messages from that queue will be suspended.

2.3 When the *GBCAST* message reaches the head of *all ABCAST* queues, the next part is begun.

The third part orders *GBCAST*'s relative to *CBCAST*'s. We assume that the *CBCAST* protocol is modified to maintain a list $IDlist_p$ for each process p, containing ID's for *CBCAST* messages that have been placed on the delivery queue of p. For now, assume that the list includes the ID's of *all* such messages. The goal of the protocol is for processes in G to agree on a list of *CBCAST* messages to be ordered *before* the *GBCAST*, and to deliver messages accordingly. The third phase executes as follows.

3.1 The process p initiating the protocol contacts all members of G.

3.2 A participant q establishes a FIFO *wait queue* (unless one already exists). Until the *GBCAST* protocol completes, messages that would have been placed on the delivery queue at q by the *CBCAST* protocols are placed on this queue instead.

3.3 If any message B in $IDlist_q$ is in $PBUF_q$ and the remaining destinations of B include sites in G, q must assume that those sites have not yet received a copy of B. Any such message is scheduled for transmission to the destinations in $REM\_DESTS(B) \cap G$, and q waits until they have been sent. It then sends $IDlist_q$ to p.

3.4 After collecting these messages, p merges all the lists it received, calling this the *before* list. It sends the *before* list to all participants. When a participant q receives this list, any message that was transmitted during Step 3.3 must have arrived, and is on the wait queue unless it has already been delivered. Similarly, during Step 1.2, all *CBCAST* messages from a failed process were either placed on the wait queue or delivered.

Finally, messages are transferred in order to the delivery queue, and normal delivery resumes:

4.1 Each participant q does the following. For each *CBCAST* B in its wait queue, if B is in the *before* list, or if there is some B' in the *before* list and $B \xrightarrow{c} B'$, or if the *GBCAST* is for a failure of process f and $SENDER(B) = f$, then B is added to the list.

4.2 Any messages in the wait queue which are also in the *before* list are now transferred to the delivery queue, preserving their relative order. The *GBCAST* message is then placed on the delivery queue.

4.3 If there are no other *GBCAST* protocols in progress, p appends the contents of the wait queue to the delivery queue and deletes the wait queue.

4.4 The *GBCAST* messages are removed from the heads of the *ABCAST* queues, allowing *ABCAST* messages to be delivered.

If a failure occurs, any participant can restart the protocol from the beginning. As with *ABCAST*, participants reply using the deliverable priority of the *GBCAST* message if they know it; all other steps of the protocol are idempotent and can be repeated without ill effect.

Correctness. *GBCAST* is atomic because no participant can deliver a *GBCAST* message until all have received it, hence if any delivers it, all can restart the protocol.[4]

GBCAST's to the same process group are ordered in the same way at every member because each *GBCAST* is assigned a unique priority value (Step 2.3), and is delivered in this order.

GBCAST's are ordered consistently with respect to *ABCAST*'s because a copy of the message is placed on each *ABCAST* queue, and the second part of the *GBCAST* protocol is the same as the *ABCAST* protocol.

To show that *GBCAST*'s are ordered in the same way relative to *CBCAST*'s, we must show that if an *CBCAST* is delivered before the *GBCAST* at a member of G, it will be delivered before the *GBCAST* at any other member that it is sent to. The *CBCAST*'s delivered before a *GBCAST* at a process q are those placed in the delivery queue before the wait queue is established, as well as the *CBCAST*'s in the *before* list that are delivered in Step 4.2. Now, any *CBCAST* in the delivery queue before the wait queue is established must be in $IDlist_q$ in Step 3.2 and is hence in the *before* list. Also, any message delivered in Step 4.2 is in the *before* list, or *precedes* some message in the *before* list. It suffices to show that any message delivered by q arrives in the wait queue of any other destinations in G before Step 4.2 is executed there. This, however, is immediate

[4]By the same reasoning, if one member of a process group initiates a checkpoint using a *GBCAST* (Sec. 3.7), all members that stay operational long enough will do so as well.

because a copy of any such message will have been in $PBUF_q$ during Step 3.3, hence q did not respond until it had confirmed its delivery.

Selecting some of the messages from the wait queue to be delivered ahead of others could conceivably upset the *CBCAST* delivery order. But, assume that *CBCAST* B is before B' on some wait queue, and that B' is delivered during step 4.2 but B is not. Clearly, $\neg\, B \xrightarrow{c} B'$, since Step 4.1 would otherwise have added B to the *before* list. Thus, the *CBCAST* delivery constraints are respected.

Finally, observe that because of the flush performed during part 1, the protocol does not begin executing until all messages from a failed process f have been delivered to their destinations. Hence, such messages are either on the delivery queue for the destinations or on a wait queue, if some other *GBCAST* protocol was executing at the time. Step 4.1 then ensures that the *GBCAST* is delivered after any other message from f. This completes the proof. □

Optimizations. The *GBCAST* protocol can be optimized simply by merging steps together. Moreover, the flush that is done in part 1 could be invoked directly from the view management protocol - then, instead of doing this on a per-process basis, which would be extremely costly, it would occur on a per-site basis, at relatively low cost. If this were done, a 2 round protocol would result, not counting the cost of the flush, and performance should be acceptable. A method for controlling the length of *IDlist*'s is given below.

Locking process group views. Recall from the end of Sec. 3 that it is desirable for a process group member to be able to transmit atomically to all other members of its group with the assurance that the transmission will not iterate in the addressing protocol. *GBCAST* can readily be changed to provide this behavior. To do so, it is necessary to associate a *lock* with the process group view; while the view is locked, new *GBCAST*, *ABCAST*, and *CBCAST* operations cannot be initiated. During the first phase of a *GBCAST* operation that increases the group membership, this lock would be acquired; if it cannot be acquired (because some other *GBCAST* has done so), the *GBCAST* would be interrupted and retried. It is not necessary to lock the view for *GBCAST*'s that do not increase the group membership. Once the view is locked, it suffices to order all pending *ABCAST* and *CBCAST* events before the *GBCAST* using the mechanisms discussed above. The view lock would then be released when the *GBCAST* delivery takes place. Recall that broadcast participants learn about the failure of a coordinator using the monitoring mechanism, which is independent of the process group view mechanism. Thus, the presence of these locks does not change the way that these protocols terminate after failure of the coordinator.

4.4. An associative store and distributed garbage collection facility

We now define an *associative store* mechanism, which is used by the above protocols to manage the information associated with message id's. Each site s maintains a local store denoted $STORE_s$. The contents of a store are tuples (*id*, *alist*), where *id* is a broadcast ID and *alist* is a list of zero or more *attributes*. A set of operations is defined on the store for each site (there is no facility for accessing the store at a remote site). The operation **st_add(id)** creates an empty list for the designated ID, **st_insert(id, aname, value)** adds an attribute with name *aname* and value *value* to the list, **st_find(id, aname)** looks up an attribute, and **st_delete(id, aname)** deletes an attribute (but not the ID). The special attribute *DISPOSABLE* is inserted when an entry will no longer be referenced. In the *ABCAST* and *GBCAST* protocols, an ID becomes DISPOSABLE at a site running (or completing) the protocol after it transmits commit messages. In the *CBCAST* protocol, an ID becomes DISPOSABLE at a site when the corresponding *REM_DESTS* field is empty.

We now give a method for deleting information associated with a message-id after the ID is marked as *DISPOSABLE* by some site. The method defines a *delete action* which is taken when an ID is discarded; at the end of the section we give these delete actions. Since copies of messages may be transmitted to a site while it is running the garbage collection protocol, and message ID's are used to avoid delivery of duplicate copies of messages, some care must be taken to ensure that copies of a message will not be received after its ID is deleted. Accordingly, the algorithm employs an additional field associated with each message ID in the store, *DONT_SEND*, which is initially null and subsequently lists sites that have run the protocol.

1. Periodically, each site a makes a list of tuples, $(ID, DONT\_SEND)$ for $DISPOSABLE$ ID's. It invokes the *delete action* for each listed ID, and then, to each site s, transmits a list $\{ID_i\}$ containing any ID_i that is $DISPOSABLE$ such that s is not in $DONT\_SEND_i$.

2. On receiving a list $<ID_0, ID_1, \cdots>$, site b takes the following actions.

   a. If ID_i is not already in $STORE_b$, it is added.

   b. The REM_DESTS field associated with the ID is made empty and it is marked as $DISPOSABLE$. This ensures that b will not send additional copies of the message and that the ID will eventually be deleted from $STORE_b$.

   c. It adds a to the $DONT\_SEND$ field associated with this ID in $STORE_b$.

   After processing the list, b sends an acknowledgement to a.

3. After receiving acknowledgements from all operational sites, a deletes the ID, the $DONT\_SEND$ field, and and other information associated with the ID from $STORE_a$. The $DONT\_SEND$ field prevents a site from adding an ID to its store after deleting it, i.e. when some other site executes the protocol to delete the ID from its own store.

$ABCAST$ and $GBCAST$ have no special delete action -- the priority information that they saved is discarded automatically when the protocol completes. The delete action for $CBCAST$ is to remove the message ID from the *idlist* of any processes that have received a copy of it, and to delete the message itself from $PBUF_p$ for any processes p at the site. Thus, the length of an idlist will be determined by the number of active broadcasts, which should be small.

Correctness. This follows because no site deletes a message-ID until all operational sites have sent acknowledgements in step 2, but after step 2.c duplicates of a message will no longer be sent to a site that has run the protocol. □

4.5. CBCAST flush implementation

Flush is invoked in two ways, each having a slightly different implementation. When a process invokes **flush**, the $CBCAST$ algorithm is such that if any $CBCAST$ B is active, a copy of B

will be present in $PBUF_p$ for any process p that might take actions causally dependent on the delivery of B. Hence, it suffices to schedule all messages in $PBUF_p$ for transmission, and then wait until all have been sent, in the sense of Sec. 4.1.

If **flush** is invoked for a group address change, a stronger condition is needed, namely that there is no active *CBCAST* that could still be rejected. This is satisfied by doing an *CBCAST* requesting that group members return an acknowledgement. If this *CBCAST* is ordered after all messages that have been sent previously, the acknowledgement will not be received until the messages in question have all been accepted.

4.6. Wide area networks

The above protocols do not address inter-cluster communication, but can be extended to do so. Our approach assumes that in a large network, the notion of a site view is meaningful only in the context of a particular cluster. Accordingly, we do not attempt to extend the failure agreement protocol to include sites outside a cluster, but instead require that a process monitoring an *external* process (that is, external to its cluster) learn about the failure of that process indirectly, from a monitoring facility actually resident in the remote cluster. The monitoring facility of Sec. 4.1 can be extended to implement this transparently. Next, we limit the extent to which *CBCAST* messages can propagate through the system by requiring that a *CBCAST* be sent directly to its external destinations, instead of being piggybacked, while still respecting the transmission ordering rules given above (since *precedes* is acyclic, *CBCAST* messages can always be transmitted in the order corresponding to a topological sort on their *clabels*, without doing any piggybacking). *CBCAST* messages are sent directly to their external destinations, hence the flush phase that precedes a failure *GBCAST*'s need only be run within the cluster where a failed process or site resided. Finally, we require that messages from the first phase of a *GBCAST* or *ABCAST* protocol, as well as all *CBCAST* messages, be sent to local sites before the first transmission to an external site is initiated. This implies that the local participants all know of the protocol before any external participant learns of it. Thus, even though the flush phase of a failure *GBCAST* is carried out locally, it will definitely flush pending broadcasts when it starts, unless all local participants have failed. Moreover, in the case of *GBCAST* and *ABCAST* this permits a performance optimization whereby the new coordinator can be located in the same cluster as the initial one. The protocols are otherwise unchanged, and process groups that extend over cluster boundaries can be supported transparently. Notice also that it is now possible to run the garbage collection algorithm within a cluster. Although it remains necessary to inform external sites when information associated with an *ID* can be deleted, those sites are never sources of piggybacked *CBCAST* messages, hence they can initiate the delete action for an *ID* immediately upon learning that it is deletable, and the *DONT_SEND* mechanism is not needed.

Thus, with minor modifications, fault-tolerant process groups can be supported transparently in hierarchical wide area networks. The local protocols are unaffected, hence the approach will take advantage of communication locality. Tuning the I/O scheduling policies for a network of this sort represents a challenging problem for future investigation.

4.7. Liveness

In the interest of brevity, we omit a formal proof that the protocols given above are free of deadlock and livelock.

5. Applications

A local-area implementation of the communication subsystem proposed here is being undertaken as part of the *ISIS* project at Cornell. *ISIS* is a distributed computing system that provides several forms of support for fault-tolerant computing. A prototype that was completed in January 1985 transforms non-distributed abstract type specifications into fault-tolerant, distributed implementations, called *resilient objects* [Birman-a]. Resilient objects achieve fault tolerance by replicating the code and data managed by the object at more than one site. The resulting *components* synchronize their actions to provide the effect of a single-site object. In the presence of failures, any ongoing operation at a failed component is continued by an operational one. Also, a resilient object continues to accept and process new operations as long as at least one component is operational. Finally, failed components recover automatically when the site at which they reside is restarted.

The initial version of *ISIS* used a simple communication layer that provided an atomic broadcast with no ordering properties. This was unsatisfactory for two reasons. First, the implementation grew very complex because of the need to include, in various parts of the system, protocols to preclude orderings that might lead to inconsistencies, especially in the presence of failures. Second, the high degree of synchronization resulting from these protocols lowered system performance. When reimplemented using a preliminary version of the primitives presented here, the system became much simpler, and performance improved due to the highly concurrent nature of the primitives.

The overall structure of the new system is layered (Fig. 2.). The lowest levels, which we view as part of the kernel, include the site view manager, input filter, and output filter described earlier. Also included in the kernel is the associative store and the per-process buffer pool. A special kernel process is run at high priority to implement the protocols. Client processes communicate with one another either directly using the protocols described here, or indirectly using resilient objects and fault-tolerant *bulletin boards* [Birman-b]. The latter provide an asynchronous interface to shared data structures on which information can be "posted," with varying levels of consistency depending on the intended use (this paradigm will be familiar to readers from the A.I. community). High level support for process groups includes a process group manager, which assists in creation, destruction, and communication with process groups, and a communication switchboard that facilities connection establishment.

The communication primitives are used at all levels of our current work. In the interest of brevity, however, we restrict ourselves to a survey of just a few ways in which they are employed.

5.1. Updating replicated data

When replicated data are updated, care must be taken to ensure that the updates occur in the same order at all copies. Otherwise, the copies can become inconsistent. In an environment where no broadcast ordering properties are guaranteed, this is done by preceding an update to a local copy by a broadcast to the remote copies and waiting for confirmation from the remote copies that the update has been carried out, before allowing another local update to occur. This kind of synchronization means that the rate at which updates can occur is limited by the time it takes for a message to travel a round trip, which can be unacceptably high. If *CBCAST*'s are instead used to instruct remote copies to perform updates, an update can be considered complete when the local update is carried out. No further synchronization is required, because the properties of *CBCAST*'s guarantee that all the copies receive the update, and do so in the required order [Joseph]. The rate at which updates can now be performed is now the rate at which local updates can be done, which is usually much higher than the previous case. At the same time, the protocol for carrying out a replicated update is much simpler, as it consists of a single *CBCAST*.

5.2. Coordinator-cohort computations

In *ISIS*, one of the components of a resilient object is designated as the *coordinator* for the execution of a particular operation. The others, its *cohorts*, act as passive backups. If the coordinator fails, a cohort takes over as and restarts the request.

The process group abstraction facilitates the implementation of coordinator-cohort computations. The components of a resilient object are placed in the same process group, and each request to perform an operation is transmitted to all the components using *CBCAST*. Since each component has the same process group view, they can independently decide on a unique coordinator for the request by using the same algorithm, without running an agreement protocol.[5]

The *GBCAST* ordering properties prevent inconsistencies from arising when failures or recoveries occur. After a failure, cohorts can pick a new coordinator consistently, and because all have received the same messages from the previous coordinator, the object data is in a consistent state at all components. When a component recovers, it uses *GBCAST* to rejoin the group, hence

[5] In *ISIS* this is done as follows. If a request arrives from site s, the coordinator is the site t in the process group view minimizing $abs(t - s)$. This normally locates the coordinator for a computation at the same site as the site where the request originated, which improves response time.

all the operational components receive the *GBCAST* in the same state and any can transfer data to reinitialize the recovering component.

5.3. Managing locks on replicated data

Lock-based concurrency control is the most common method for obtaining serializability [Bernstein] [Gray]. The usual locking method for replicated data is to obtain write locks on all copies and read locks on only one. This means that if the site at which a read lock is obtained were to fail, all information about this read lock would be lost. In the *ISIS* recovery scheme, as in many that use a saved state for recovery, it is necessary for the executions to be deterministic, and a change in serialization order after failure would violate this. This implies that for *ISIS* to provide roll-forward executions after failure, information about read locks must be replicated as well. Unfortunately, whereas replicating write-locks is reasonable, acquiring read locks at all sites would be terribly inefficient. Instead, the ordering properties of the broadcast primitives are used to obtain an equivalent effect.

A read-lock is first obtained locally. Then, a *read lock registration* message is *CBCAST* to the other copies of the data item. The sender immediately continues execution, as if its read-lock were already replicated, although the message may not actually have been delivered anywhere. If the sender fails before any message leaves the site, the effect is as if the read never occurred (recall that a failure destroys all information at a site). If, on the other hand, a site has received any message m sent after the lock acquisition, the *GBCAST* protocol for the failure will ensure that the read-lock registration message is delivered before the failure is detected by the process managing the lock. Thus, the read-lock behaves like a fully replicated one.

Unlike a read-lock, a write-lock must be explicitly granted by all components of an object. However, a deadlock could occur if concurrent write-lock requests on the same data item are granted in different orders by different components. This problem can be avoided by using *ABCAST*'s for write lock acquisition requests. If the data item name is used as a *ABCAST* label, write lock requests on the same data item are ordered in the same way at all components, and deadlock is avoided.

5.4. Performance issues

A prototype communication layer similar to the one described here has been operation since Jan. 1985 [Birman-a]. Instrumentation of a collection of resilient objects yielded performance measures that shed light on the way these primitives can affect a real distributed system. One, the response time for a typical request, measures the critical path before a reply can be issued to a caller.˙ We considered a fault-tolerant file object distributed to 3 sites (SUN workstations). A request that acquires a replicated write-lock, updates a replicated data item, and then responds to its caller sends its reply after about 0.3 seconds; additional updates delay the response by 0.1 seconds each (the difference reflects the one-time cost of concurrency control). When *ISIS* is run in a synchronous mode, verifying that each update has actually completed before the coordinator undertakes any subsequent operations, such a computation requires 0.85 seconds, with additional updates requiring 0.5 seconds each. Moreover, the performance of the synchronous version degrades as the number of sites increases, while the concurrent version gives the same performance regardless of the number of participating sites. Thus, concurrent communication primitives can have a substantial impact on performance.

When a high level of concurrency is achieved in a distributed computation, it can remain active after replying to the process that initiated it. To isolate the effect of concurrency on the above figures, the total elapsed time between the issuing of the request and the true termination of the operation can be measured. In *ISIS*, we find that a single asynchronous update terminates about 0.2 seconds after returning a result, with subsequent updates delaying termination by about 0.05 seconds each, and with linear degradation as the number of sites increases. In practical terms, when a resilient calendar application was executed on two SUN terminals sitting side by side, a calendar update caused both screens to refresh essentially simultaneously. Considering that this version of UNIX on the SUN 2 is not known for blinding speed, the performance we have achieved is completely satisfactory.

Finally, we examined the effect of piggybacking on the performance of the *ISIS* prototype. To do this, we placed the file object under a "distributed load," presenting operations to it at multiple sites and measuring the mean delay before a response was computed and returned. As the load rises, a backlog of asynchronous updates begins to form, and the *CBCAST* implementation takes advantage of this to begin piggybacking multiple messages on each packet. Because the computing time in a simple object such as this is largely spent reading request messages, and only a small percentage of these require a response, efficiency can rise dramatically if a single incoming packet carries several messages. Precisely this effect was observed: for objects distributed over small numbers of sites (2 to 6), performance under relatively heavily loads (a load of 7 operations per second) was nearly as good as for a nondistributed object under a very light load (<1 operation per second), and far better than for a nondistributed object under the same heavy load. This is because the concurrent update algorithm concentrates the real processing at a coordinator (cohorts do very little). Thus, if different requests have different coordinators, each does less work than a single coordinator performing both requests. Moreover, the benefit of replication more than outweighs the overhead associated with asynchronously broadcasting the updates to cohort processes. Thus, in *ISIS* at least, the primitives are tremendously valuable.

To summarize, for a variety of distributed application in *ISIS*, and no doubt in other systems as well, the communication primitives described in this paper permit extremely good performance -- but with the ability to tolerate failure as well. Moreover, they actually simplify the design of distributed software and reduce the probability that subtle synchronization or concurrency related bugs will arise. The fault-tolerant process group approach to distributed computing appears to be a major improvement over alternative programming methodologies for this domain.

6. Limitations

One weakness of the work described in this paper is its tendency to block in the presence of partitioning failures, when two or more subgroups of operational sites form, within which communication remains possible, but between which it is degraded or impossible. We are now investigating the adaptation of methods from [ElAbbadi-a] [ElAbbadi-b] to address this issue. We are also examining the possibility of integrating communication primitives with synchronized clocks, for use in shared memory systems and tightly coupled multi-processors.

A second limitation is our implicit assumption that within each cluster, failures and recoveries will be sufficiently infrequent to permit the site-view protocol to terminate. We believe that these assumptions hold in most existing distributed systems. In less benign environments, however, where this form of stabilization might not occur, it is not clear that our approach to fault tolerance would perform satisfactorily.

7. Conclusions

The experience of implementing a substantial fault-tolerant system left us with insights into the properties to be desired from a communication subsystem. The broadcast primitives described in this paper present a simple interface, achieve a high level of concurrency, can be used in both local and wide area networks, and are applicable to software ranging from distributed database systems to the fault-tolerant objects and bulletin boards provided by *ISIS*. Because they are integrated with failure handling mechanisms and respect desired event orderings, they introduce a desirable form of determinism into distributed computation without compromising efficiency. A consequence is that high-level algorithms are greatly simplified, reducing the probability of error. We believe that this is a very promising and practical approach to building large fault-tolerant distributed systems, and the only one that leads to confidence in the correctness of the resulting software.

8. Acknowledgments

The evolution of this paper has been influenced by many of our colleagues, to whom we are deeply grateful. Particular thanks go to Amr El Abbadi, Ozalp Babaoglu, Eric Cooper, Thomas Raeuchle, and Pat Stephenson, for their many detailed comments. We are also indebted to Jay Misra and Mani Chandy for discussions and comments about a very early draft of this paper, to Dale Skeen, who helped found the *ISIS* group and was responsible for the ordering algorithm used in the *ABCAST* protocol, and to the members of the ANSA project in Cambridge England for stimulating discussions about the issues raised herein. Finally, the comments of the referees are gratefully acknowledged.

9. References

[Bernstein] Bernstein, P., Goodman, N. Concurrency control algorithms for replicated database systems. *ACM Computing Surveys* **13**, 2 (June 1981), 185-222.

[Birman-a] Birman, K. Replication and availability in the *ISIS* system. 0th ACM Symposium on Operating Systems Principles, appearing as *Operating Systems Review* **19**, 5 (Dec. 1985), 79-86.

[Birman-b] Birman, K., Joseph, T., Stephenson, P. Programming with fault-tolerant bulletin boards. Dept. of Computer Science, Cornell Univ., TR 86-???, August 1986.

[Chang] Chang, J, Maxemchuk, N. Reliable broadcast protocols. *ACM TOCS* **2**, 3 (Aug. 1984), 251-273.

[Cooper] Cooper, E. Replicated distributed programs. *10th ACM Symposium on Operating Systems Principles*, appearing as *Operating Systems Review* **19**, 5 (Dec. 1985), 63-78.

[Cristian] Cristian, F., *et al.* Atomic broadcast: From simple message diffusion to Byzantine agreement. IBM Technical Report RJ 4540 (48668), Oct. 1984.

[El Abbadi-a] El Abbadi, A., Skeen, D., Cristian, F. An efficient algorithm for replicated data management. *Proc. 4th ACM Symp. on Principles of Database Systems*, Portland, OR, March 1985, 215-229.

[El Abbadi-b] El Abbadi, A., Toueg, S. Availability in partitioned, replicated databases. *Proc. 5th ACM Symp. on Principles of Database Systems*, Boston, MA, March 1986.

[Gray] Gray, J. Notes on database operating systems. *Lecture notes in computer science* **60,** Goos and Hartmannis, *eds.* Springer Verlag 1978.

[Goodman] Goodman, N., *et al.* A recovery algorithm for a distributed database system. *Proc 2nd ACM PODS*, Atlanta, GA (March 1983), 8-15.

[Joseph] Joseph, T. and Birman, K. Low cost management of replicated data in fault-tolerant distributed systems. *ACM TOCS* **4**, 1, Feb. 1986, 54-70.

[Lamport] Lamport, L. Time, clocks, and the ordering of events in a distributed system. *CACM* **21**, 7, July 1978, 558-565.

[Schlicting] Schlicting, R, Schneider, F. Fail-stop processors: An approach to designing fault-tolerant distributed computing systems. *ACM TOCS* **1**, 3, August 1983, 222-238.

[Schneider] Schneider, F., Gries, D., Schlicting, R. Reliable broadcast protocols. *Science of Computer Programming* **3**, 2 (March 1984).

[Skeen-a] Skeen, D. Crash recovery in distributed database systems. Ph.D. dissertation, Department of EECS, U.C. Berkeley, 1980.

[Skeen-b] Skeen, D. Determining the last process to fail. *ACM TOCS* **3**, 1, Feb. 1985, 15-30.

[Skeen-c] Skeen, D. Unpublished communication.