Programming with Shared Bulletin Boards in Asynchronous Distributed Systems

Kenneth P. Birman
Thomas A. Joseph
Frank Schmuck
Pat Stephenson

TR 86-772

August 1986
(Revised December 1986)

Department of Computer Science
Cornell University
Ithaca, NY 14853

PROGRAMMING WITH SHARED BULLETIN BOARDS IN ASYNCHRONOUS DISTRIBUTED SYSTEMS

Kenneth P. Birman, Thomas A. Joseph, Frank Schmuck, Pat Stephenson

Dept. of Computer Science,
Cornell University, Ithaca, NY 14853.

ABSTRACT

We consider loosely coupled distributed computing systems in which processes interact through shared resources, which are modeled as *bulletin boards*. The first part of the paper formalizes the notion of consistent behavior when unreliable processes concurrently access a bulletin board. This model is interesting both as a tool for showing the correctness of a bboard implementation and also because it provides a mechanism for reasoning about *consistency* in distributed systems, which was previously lacking. The remainder of the paper discusses software techniques for implementing consistent bulletin boards in a network of processors lacking shared memory. Applications for our approach range from asynchronous interprocess communication to mechanisms for achieving mutual exclusion, deadlock detection and for building distributed database systems.

1. Introduction

Two distinct styles of distributed programming are becoming prevalent. The *imperative* style is typified by *remote procedure calls* [Birrell]: a tightly coupled mechanism through which processes interact synchronously. The basic characteristic of this programming style is that one process can force another to execute code on its behalf; if a request cannot be handled promptly there is no alternative but to implement some sort of queueing mechanism. The *advisory* style has received less attention. According to this alternative, each process "publishes" information that eventually becomes accessible to other processes, but without actually forcing them to act on it [Cheriton] [Ahamad]. Advisory systems are thus relatively loosely coupled and asynchronous. A good example of the advisory programming style is the *bulletin-board* abstraction found in some artificial intelligence applications: a collection of processes (expert systems) interact by posting problems and relevant data on a common bulletin board; each process checks the board at its own convenience.

In this paper, we focus on the adaptation of data abstractions like the A.I. bulletin board to a setting characterized by processes distributed among multiple sites (lacking shared memory) in a network, communicating with one another in an advisory manner. Henceforth, we refer to these as *bboard* data

structures, or just *bboards*. Our work was motivated by the desire to provide a simple, easily used mechanism to programmers who do not wish to be involved with the low level details of distributed computing. Bboards provide such an mechanism in a way that interposes minimal overhead. Moreover, our bboard construction tools are suitable for use in systems that employ multiple programming languages and contain programs employing very different representations of bboard data objects or interpretations of the bboard operations themselves.

In any distributed system, process and site failures and recoveries can occur, and those processes that remain operational may need to detect and act on such events. Ensuring that behavior will be correct in the presence of failures is a potentially difficult problem that, if not treated at a system-wide level, can greatly complicate application software. Accordingly, we have integrated a failure detection mechanism into the bboard facility so that processes can use bboards to monitor one another as well as for communication.

The paper develops a formal model that captures the logical behavior of a bboard executing in the presence of failures and recoveries. We show that different forms of correctness may be desired of a bboard, depending on how it will be used, and that the cost and performance of a given implementation are limited by the desired correctness constraints. Next, we discuss a particular bboard implementation, which is based on a set of reliable multicast protocols that we developed and proved correct in [Birman-a] and are using in the $ISIS_2$ system [Birman-b]. These protocols assume that processes and sites fail by halting and that network partitioning is rare. The concluding sections of this paper discuss applications of the bboard approach. We describe bboard implementations of tokens, an A.I. bboard of the sort cited above, a distributed deadlock detector, a replicated file supporting transactional access and concurrent updating [Joseph], a collection of primitives drawn from the Linda S/Net Kernel [Carriero], and a set of primitives for maintaining a message bboard described in [Ahamad]. The replicated file is particularly interesting because it shows how multiple bboards having different correctness constraints can be combined to obtain good performance in a non-trivial application.

2. The bboard abstraction

A bboard is a distributed entity that enables processes to communicate with one another. It typically consists of a set of data objects and a set of operations defined on those objects. Processes accessing a bboard (called its *clients*) do so by invoking one or more of its operations. The bboard accepts such invocations and schedules them for execution in accordance with user-specified ordering constraints. Every client observes the execution of operations; the invoking client receives a return value as well. The order in which the clients observe executions may vary from client to client, but the degree of variation is governed by the *consistency* of the bboard, as described below. The implementation of the data objects themselves are not part of the bboard, the interpretation of operations and their arguments being left to the individual bboard clients. This approach is quite flexible; in Figure 1, for example, we illustrate a scenario in which one bboard client represents a particular object as a simple

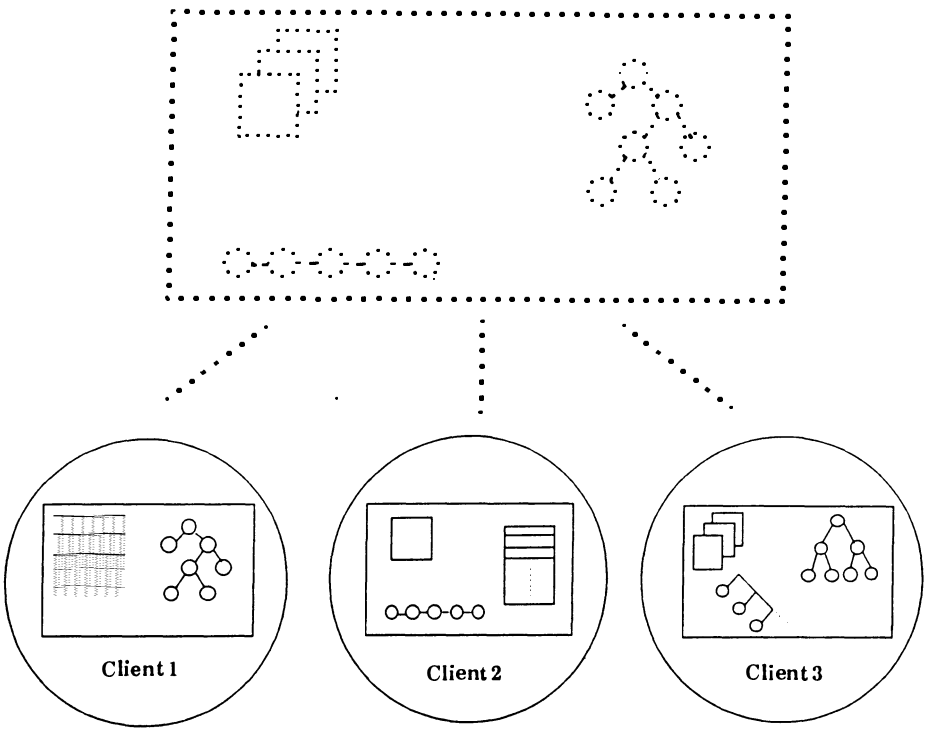

Figure 1: Clients interacting through a bboard

single-valued variable, another maintains a tree structure in which it stores the entire history of values that the variable has taken on, and a third ignores the variable entirely. A bboard can hence be thought of as a distributed scheduler for uninterpreted operations. A consequence of this is that a bboard will only ensure that every client executes the same operations with the same arguments. If an operation is non-deterministic, then different clients might see different outcomes for the same sequence of executions. An assumption we make, however, is that once a sequence of operations is scheduled for execution they can be executed sequentially in the given order. In particular, we do not allow an operation to block during its execution, waiting for an action by some subsequent operation.

Consistency refers to the order in which clients observe the execution of operations. The most natural form of consistency is *total consistency*: all clients observe executions in some single global order. A totally consistent bboard is easy and convenient to use; it appears to be a tightly-coupled shared memory. This is the abstraction favored in the *state machine* approach to distributed computing, formalized in [Lamport-a] [Schneider-a].

For some applications, total consistency is stronger than necessary. Consider the example of a chess program composed of communicating expert subsystems. A *move generator*, coded in LISP, examines the chessboard configuration and posts legal moves for evaluation. A set of *evaluators*, coded in PROLOG, computes the relative strength of black and white's positions for each move, perhaps posting subproblems that should be solved in connection with this and waiting for other evaluators to solve those subproblems. This gives rise to a tree of subproblems, each path along the tree representing configurations resulting from a valid sequence of moves from the current configuration. Meanwhile, a *display expert* (coded in C) depicts the outcome of the computation. In this application, the order in which the evaluators observe the posting of information stemming from *unrelated* branches of the tree is unimportant. On the other hand, it is necessary that the events associated with any particular sequence of moves be observed everywhere in the same order. Then, if a process takes some action because it has observed a posted subproblem, it will also see the data (the "assumptions") on which that sub-problem was based. What is important here is that the information posted on the bboard by one evaluator will often be *causally related* to information that was previously posted, but independent of

information being posted concurrently, and this causal order should be preserved. This leads us to a second type of consistency - *causal consistency.*

To determine whether one invocation causally affects another requires, in general, knowledge of the semantics of the operations invoked, which are not be available to a bboard. However, if there is any means by which information about the execution of a request r by a process P could have reached process P' before it performs r', then r and r' are *potentially causally related* [Lamport-b]. A causally consistent bboard orders all potentially causal executions, that is, executions that could causally affect one another are observed in their causal order by all clients, but the order of other executions may differ between clients. This form of consistency is useful because it is much cheaper to implement than total consistency; hence an application that uses total consistency where causal consistency would suffice incurs unnecessary overhead.

A less stringent form of consistency is *fifo consistency.* In a fifo consistent bboard, any sequence of operations invoked by the *same* client will be observed by all the clients in the order they were invoked, but operations that were invoked by *different* clients may be observed in different orders.

Finally, we consider *weak consistency*, where the orders in which processes observe most events may differ arbitrarily (except relative to failure events, as explained shortly). This form of consistency is useful in applications where a process needs to know whether other processes have carried out certain actions or has to react to process failures, but does not care about the order in which those events occurred. The reader may notice that other researchers (notably [Strom] [Jefferson]) have built systems that achieve causal consistency by starting with minimal consistency and then using a rollback technique. This possibility is not explored in our work.

Whereas consistency refers to the order in which different processes view events occurring in the system, *synchronization* refers to the degree to which processes can control this order. It may be necessary, for example, for a process that updates the value of a variable to ensure that the new value not be stored until some other process has read the old one. In our model, processes achieve this sort of synchronization using *guards.* When a process invokes an operation o on a bboard, it may label the invo-

cation with a guard, which essentially specifies a set of operations that must be completed before o is executed.

The third aspect considered here is fault-tolerance. A fault-tolerant application requires a means of detecting when a process fails, and a means for processes to consistently order the detection of the failure relative to other events in the system. Bboards assist in the design of fault tolerant systems in several ways. First, the bboard itself is resilient to failures: It provides uninterrupted service to all operational clients despite failures in the system. Second, bboards present all operational clients a consistent view of the state (operational or failed) of every client in the system. Third, a bboard allows clients to specify operations to be executed in the case of process failures. This is done by introducing a class of failure events, which can be referred to in guards in the same way as other events like the execution of operations. A guard may specify, for example, that a particular operation should be performed only after a certain process either executes a named operation or fails. Regardless of the type of consistency followed by the bboard, every client observes the same order between a failure event and any other event. Clients can use this to react to failures in mutually consistent ways. In addition, the bboard ensures that every client observes the failure of another only after it has executed all invocations done by the failed client, except for invocations whose guards require that they be observed after the failure. This latter type of guard enables a client to post an asynchronous action that will "clean up" for it after it fails. Here, we assume that processes are asynchronous and fail by halting; details are discussed in the section on implementation.

Finally, we allow the existence of multiple bboards in a system, because it may be useful to have different forms of consistency on different types of data. Indeed, this will be essential in any large system.

3. A formal model for bboards

This section first discusses the basic elements of the formal model: events, histories and guards. The consistency constraints described earlier are then formalized.

3.1. Events and event histories

Our model represents the behavior of a bulletin board in terms of the events perceived by each process in the system. A *system history* for a bboard BB, $H_{BB}=(P, R, H)$, consists of a set of processes P, a set of requests R, and a set $H = \{ (H_p, \leq_p) \mid p \in P \}$ of process histories, one for each process p that uses the bboard. Every request $r \in R$ corresponds to the invocation of an operation on the bboard by some process and its execution. A process history H_p gives the set of events seen by a particular process p and a total order $\leq_p$ on those events. There are several types of events.

1. A distinguished event, *INIT*, which corresponds to the initialization of the bboard and is the first event (with respect to the order $\leq_p$) in every process history.
2. *Invocation* events, denoted by $inv(r)$, where $r \in R$, represent the invocation of an operation by a process. Invocation events are included only in the history of the process that issued the invocation.
3. *Execution* events, denoted by $ex(r)$, where $r \in R$, represent the scheduling of an operation for execution by the bboard.
4. *Termination* events, $term(p)$, denote the termination of a process p, either by failing or by disconnecting itself from the bboard.

It will be convenient to require that all histories H_p continue even after process p has terminated, even though a real process would cease to execute requests. Accordingly, in the model, termination is simply an event after which a process does not invoke additional operations.

These ideas are formalized in the *well-formedness* condition given below. Assume that we are given a system history H_{BB}, and let $E = \bigcup_{p \in P} H_p$ be the set of all events in H_{BB}. Then:

[WF] H_{BB} is *well formed* if the following conditions hold.

(i) Each process history starts with an *INIT* event:

$$\forall p \in P: \ \forall e \in H_p: \ INIT \leq_p e$$

(ii) Every request is invoked by exactly one process:

$$\forall r \in R: \ \exists \ unique \ p \in P: \ inv(r) \in H_p$$

(iii) The execution of a request does not precede its invocation:

$$\forall\, r \in R\colon\ \forall\, p \in P\colon\ \Big[\ inv(r), ex(r) \in H_p\ \Rightarrow\ inv(r) \leq_p ex(r)\ \Big]$$

(iv) A process does not invoke any operations after its termination:

$$\forall\, term(p) \in E\colon\ \forall\, inv(r) \in H_p\colon\ inv(r)\ \leq_p\ term(p)$$

Another property we require of histories is that every process observe the same order on termination events relative to all other events. This is formalized in the *termination ordering* condition.

[T] *Termination Ordering*: If process p terminates then the event $term(p)$ appears in every process history, and all other events should be ordered in the same way relative to $term(p)$:

$$\forall\, term(p) \in E\colon\ \forall\, q \in P\colon\ \Big[\ term(p) \in H_q\ \wedge\ \forall\, e \in H_q, H_p\colon\ (\ e \leq_q term(p)\ )\ \Leftrightarrow\ (\ e \leq_p term(p)\ )\ \Big]$$

3.2. Guards

The guard for a request r specifies a set of events whose executions must occur before r can be executed by a process. This set is said to *satisfy* the guard. A guard may be non-deterministic, meaning that the set of events satisfying the guard need not be unique. In this case, at least one such set of events must occur before r. A guard for a request uses *request names* to refer to other events. A request name is a unique ID assigned to a request by the bboard, and is given to the invoking process at the time a request is invoked. A process cannot predict or unilaterally generate a request name, but can store one or pass one around without restriction. We use the notation *r.name* to denote the name of request r. Similarly *r.proc*, *r.oper*, and *r.guard* to refer to to the process that submitted the request r, the operation invoked by the request, and the guard that was specified for that request. We also assume that every process p has a unique name which we denote by *p.name*.

Guards are defined as follows:

Guard g **for request** r		**S satisfies** g **at process** p **if**	**Comment**
(i)	Φ	S = {*INIT*}	g is trivially satisfied
(ii)	*s.name*	S = {$ex(s)$}	r must observe the execution of request s
(iii)	[*q.name*, *op*]	S = {$ex(s)$} where *s.proc* = q, *s.oper* = *op*	r must observe the execution of

		or S = $\{term(q)\}$	operation *op* invoked by process *q* or the termination of *q*
	$[q.name, \Phi]$	S = $\{ex(s)\}$ where $s.proc = q$ or S = $\{term(q)\}$	*r* must observe the execution of any operation invoked by process *q* or the termination of *q*
	$[\Phi, op]$	S = $\{ex(s)\}$ where $s.oper = op$ or S = $\{term(q) \mid q \in P\}$	*r* must observe the execution of *op* invoked by *any* process *p* or the termination of *all* processes
	$[q.name, TERM]$	S = $\{term(q)\}$	*r* must observe the termination of process *q*
(iv)	*after*(*s.name*) g'	S satisfies g' at *p* and $\forall e \in S$: $ex(s) \leq_p e$	*r* must observe a set of events satisfying g' all of which occur after *s* is executed
(v)	g_1 **and** g_2	S = $S_1 \bigcup S_2$ where S_1 satisfies g_1 at *p* and S_2 satisfies g_2 at *p*	Both guards must be satisfied
(vi)	g_1 **or** g_2	S satisfies g_1 at *p* or S satisfies g_2 at *p*	Either one of the guards must be satisfied

Some aspects of the guard syntax need clarification. First, notice that a guard of the form $[q, op]$ is considered to be satisfied if process *q* fails or terminates, even if it does not invoke operation *op*. Our implementation provides a way to determine during execution whether or not a process *p* has terminated, and to distinguish normal termination from failure. Second, observe that notation (ii) delays an invocation until a particular event $ex(s)$ occurs, whereas notation (iv) delays it until some *subsequent* events (indicated by g') occur. The former form is used when a request *s* is already known. The latter form is used when a request *s* has begun a sequence of actions that will be terminated by a subsequent request s'. Although $s'.name$ is unknown, this sort of guard could be used used provided that a pattern that will match s' can be constructed. Finally, in weak, fifo, or causally consistent bboards, it is important to keep in mind that the set of events satisfying a guard for a particular operation need not be the same in all histories, because the ordering of events in the histories can differ and a guard may be just a pattern. If this behavior is not desired, a guard that can only be satisfied in one way or a totally consistent bboard should be used.

The notion that all executions should occur only after their guards have been satisfied is formalized in the *guard satisfaction* condition.

[GS] *Guard Satisfaction*: For every execute event $ex(r)$ in a history H_p, there is a set S of events preceding $ex(r)$ in H_p that satisfy the guard of r:

$$\forall\, p \in P\colon\ \forall\, ex(r) \in H_p\colon\ \exists\, S \subset H_P\colon \left[S \text{ satisfies } r.guard \text{ in } p \ \wedge\ \forall\, e \in S\colon\ e \leq_p ex(r) \right]$$

3.3. Formal definition of consistency

The type of consistency defines the relationship between the views of the abstract bboard held by different processes by requiring that all processes see the same operations being performed on the bboard and that certain orderings between events be observed. Some aspects of consistency are common to all bboards, such as the requirement that failure events be totally ordered relative to other events, whereas other aspects depend on the degree of consistency the bboard provides its clients. This section formalizes the consistency constraints on bboards.

We begin by formalizing the notion of potential causality introduced in Section 2. Given a well formed system history H_{BB} we define the *potential causality relation* $\rightarrow_C$ to be the reflexive and transitive closure of the relation defined by the following two conditions:

[C1] Every execution of a request is related to its invocation:

$$\forall\, r \in R\colon\ inv(r) \rightarrow_C ex(r)$$

[C2] An invocation event at process p is related to all execution events that precede it in p's history:

$$\forall\, r, s \in R\colon \left[\exists\, p \in P\colon\ ex(r) \leq_p inv(s) \right] \Rightarrow\ ex(r) \rightarrow_C inv(s)$$

This definition of potential causality is very similar to the one in [Lamport-b] because invoking an operation is much like sending a message and executing it is like receiving that message. If $e \rightarrow_C e'$, we say that e *causally affects* e' and that e' *causally depends* on e. Note that it follows from the definition above that if e occurs after e' in real time, then e' cannot causally depend on e.

We now use these definitions to formalize what it means for a bboard to satisfy each of the four types of consistency.

[WC] A system history H_{BB} satisfies weak consistency if it satisfies [WF], [T] and [GS].

[FC] A system history satisfies fifo consistency if it satisfies weak consistency, and if requests issued by the same process with identical guards are executed everywhere in the order they were invoked:

$$\forall\, r, s \in R:\ \left[\exists\, p \in P:\ inv(r) \leq_p inv(s)\right] \wedge (r.guard = s.guard) \Rightarrow \forall\, q \in P:\ ex(r) \leq_q ex(s)$$

[CC] A system history satisfies causal consistency if it satisfies fifo consistency and if causally related operations are executed in the same order everywhere:

$$\forall\, r, s \in R:\ ex(r) \rightarrow_C ex(s) \Rightarrow \forall\, p \in P:\ ex(r) \leq_p ex(s)$$

[TC] Finally, a system history satisfies total consistency if it satisfies causal consistency and if all operations are executed in the same order everywhere:

$$\forall\, r, s \in R:\ \forall\, p, q \in P:\ (\, ex(r) \leq_p ex(s)\,) \Leftrightarrow (\, ex(r) \leq_q ex(s)\,)$$

This means that all process histories are identical except for invocation events.

3.4. Cleanup ordering

In addition to the consistency constraints defined above it is often useful to impose extra conditions on the ordering of events after the termination of a process. For example a process p may use the guard $[p.name, TERM]$ to invoke a cleanup operation to be executed after it terminates. In this situation it is desirable that this operation be executed immediately after the termination of p and before any other operations are executed. Also, any other pending operations invoked by p should be executed before any process observes $term(p)$. This idea is formalized below in what we call the *cleanup ordering* condition on a bboard history. Among the operations invoked by p we distinguish normal operations for which guards are satisfied by operations executed before the termination of p, cleanup operations which become enabled immediately after $term(p)$, and operations which are delayed even further. We require that every other process q observes (i) the execution of normal operations before the termination of p, and (ii) the execution of cleanup operations immediately after $term(p)$. We use the notation e_q^p to denote the last cleanup operation of p observed by q; that is we define the event $e_q^p \in H_q$ such that (a) $term(p) \leq_q e_q^p$, (b) all requests executed between $term(p)$ and e_q^p in H_q were invoked by p, and (c) for all events $e' \in H_q$ that satisfy (a) and (b): $e' \leq_q e_q^p$. (If the first request executed after $term(p)$ in H_q was not invoked by p then $e_q^p = term(p)$.)

[CO] A bboard history H_{BB} satisfies the *cleanup ordering* condition if:

$\forall\ term(p) \in E\colon\ \forall\ r \in R$ such that $inv(r) \in H_p\colon\ \forall\ q \in P$:

(*i*) $\left[\exists\ S \subset H_q\colon\ S\ satisfies\ r.guard\ at\ q\ \wedge\ \forall\ e \in S\colon\ e <_q term(p) \right] \Rightarrow\ ex(r) \leq_q term(p)$

(*ii*) $\left[\exists\ S \subset H_q\colon\ S\ satisfies\ r.guard\ at\ q\ \wedge\ \forall\ e \in S\colon\ e \leq_p e_q^p \right] \Rightarrow\ ex(r) \leq_q e_q^p$

3.5. Relating the model to a physical implementation

Given a system history, Section 3.3 shows how to decide if it is consistent. We say that a physical implementation of a bboard is correct if we can show that all possible executions of the system generate only consistent histories. Every physical state that such a system enters during its execution will be a prefix of a consistent history, that is the sequence of events seen by each process up to a given time can always be extended to a complete consistent history by appending additional execution and termination events. Unfortunately, by this definition, an implementation that never executes any operation is correct. This leads us to a condition on bboards that we call *progress*:

[P] If the guard of a request is satisfied anywhere, then its execution appears in every history.

$$\forall\ r \in R\colon \left[\exists\ p \in P\colon\ \exists\ S \subset H_p\colon\ S\ satisfies\ r.guard\ at\ p \right] \Rightarrow\ \forall\ q \in P\colon\ ex(r) \in H_q$$

A reasonable implementation of bboards should satisfy [P].

4. Consistency hierarchy in advisory systems

We have defined a hierarchy of consistency levels, with weak consistency at the bottom of the hierarchy and total consistency at the top. Each level requires an order to be observed on certain events that the lower levels do not: weak consistency places no ordering requirements, fifo consistency that requests invoked by individual processes be ordered, causal consistency requires that potentially causal events be ordered, and total consistency requires that even non-causal events be ordered. The question arises, however, of whether these levels of consistency are equivalent in the following sense. Is it possible for a process accessing a bboard to execute a protocol that will order events not ordered by the bboard, thus obtaining behavior equivalent to that from a bboard with a higher level of consistency? Under the advisory model of computing, this is not possible. In this model, process P can

only post its information and request that other processes post a response. There is no guarantee that other processes will actually read the posted information, or that they will respond within a finite time. In fact, since P will in general not be aware even of the number of other processes in the system, it will never know when all processes have responded. Even if it is agreed *a priori* that all processes periodically post information regarding events they have recently observed, the fact that there is no bound on the relative speeds of processes means that P could have to wait indefinitely. We note that if such bounds existed, it would be possible for processes to synchronize actions and order events independently of the bboard [Cristian], but this is a significant deviation from the advisory model.

Another question that arises is whether the hierarchy can be extended beyond total consistency. Total consistency requires all individual events to be ordered. A higher form of consistency could require that groups of events be ordered relative to other groups. For example, it could be required that all events by the same process be ordered in the same way relative to all events by other processes. This is a form of *serializability*: The processes behave like transactions. Carrying the analogy further, a still higher level of consistency might provide an ordering on groups of groups of events, and so forth.

Interestingly, once a process has access to an totally consistent *helper* bboard, higher levels of consistency can be achieved using it. We use this technique in the example presented in Section 6.3.

5. Some properties of bboards

Before we describe our implementation, we discuss two issues that arise when using bboards.

5.1. Deadlock

Obviously, processes using a bboard can guard operations in a way that leads to deadlock. However, we show below that if the guards used by a group of processes refer only to events initiated by other members of the group and if all processes in the group eventually terminate, then the operations they request can all be executed eventually. This means that bboards will not spontaneously cause deadlocks in an otherwise deadlock-free system.

A guard of the form [*p.name*, *op*] is automatically satisfied if process p terminates. Therefore after all processes terminate the only portions of guards that may still be unsatisfied are those that refer to request names. Recall that request names cannot be generated unilaterally or predicted by a process; they are returned by the bboard at the time a request is issued. This means that it is not possible to create a circle of requests that use the names of one another in their guards. It follows, then, that after all processes terminate there is a way to sort any unexecuted invocations so that if the guard for r' depends on the execution of r, then r precedes r' in this sorted order. The first guard *r.guard* in this order must be satisfiable, since it only refers to previously executed operations and operations by processes that have terminated (even if *r.guard* uses the notation *after*(*s.name*) g', then by assumption s must have been executed previously and we can inductively show that g' is satisfiable). The delayed operations can hence be executed in this order.

5.2. Consistent cuts

Chandy and Lamport define a *consistent snapshot* [Chandy] in a distributed system with potential causality relation $\rightarrow_C$ to be a set of events S that is closed under potential causality, together with any undelivered messages that are still in inter-process communication channels. It is usual to refer to a consistent snapshot in which the undelivered messages are not recorded as a *consistent cut*. We can then define the events at the *front* of a consistent cut S as the set $\{ e \mid e \in S \ \wedge \ \forall e' \in S\colon\ e' \rightarrow_C e \Rightarrow e' = e \}$.

A number of algorithms in distributed systems (e.g. deadlock detection) operate by verifying or establishing a property along the front of a consistent cut. There is a simple way in which a bboard can be used to record the state of a system of processes along the front of a consistent cut. Assume that each process uses a set of private operations to post changes to its own state on the bboard; no process ever invokes a private operation of some other process. Additionally, there exist read-only "snapshot" operations that examine the state information associated with several processes at once. Then, if the bboard is causally or totally consistent, each snapshot operation is executed on the front of a consistent cut across the process states. Moreover, invocations in these types of bboards satisfy a *containment* pro-

perty: If two snapshot events are causally related, the latter will reflect all the events on which the former is causally dependent. Moreover, if the bboard is totally consistent then this is true even if the snapshots are not causally related -- given any pair of snapshots, one always reflects all the events present in the other.

6. Implementation

This section describes an implementation of bboards that we are undertaking at Cornell. We begin with a brief overview of the environment within which this implementation functions. Some pragmatic objectives that were relevant to the internal implementation strategy we adopted are discussed. Finally, the communication primitives on which the implementation is based are presented together with the bboard algorithms and a proof that the implementation achieves the desired forms of consistency.

Our implementation permits a single process to interact with multiple bboards of differing consistency levels. Moreover, although data items and operations are associated with specific bboards, as in the model, guards can include request names drawn from multiple bboards. This feature turns out to be quite useful in developing bboard-based application software.

6.1. Computing and communications environment

Our work assumes a hierarchically structured network of computers supporting message-based interprocess communication. *Clusters* of sites employ a high speed local area network for communication and communication partitioning does not occur; long-haul links interconnect clusters and are slower and subject to infrequent partitioning failures. In some (rare) situations, a bboard that extends across a partitioned long-haul link will block until communication is restored; this never happens within a cluster. Sites and individual processes both fail by *crashing*: Execution ceases (no undetectably incorrect messages are sent first) and the local states of failed processes are irrevocably lost. Later, we will discuss prospects for recovering state information after a failure by periodically saving checkpoints.

6.2. Overview and Language Features

The implementation is designed as a package of library routines which are accessible from the *C* language under $ISIS_2$ [Birman-b], a new version of the *ISIS* system running on BSD 4.3 UNIX. Eventually, we hope to provide interfaces for other languages and operating systems. Each client program will be linked to the bboard package at compile time. At run time, a client can *create* a new bboard or *enroll* in an existing one. When enrolling, the bboard state can be transferred from some other user using mechanisms provided by $ISIS_2$. It can then issue operations to the bboard using the following mechanisms (the notation will be adapted to suit the host programming language):

BBoards and sessions

bb = **bboard** *(b_name, ops, level)*	Enroll in or create a bboard. A unique name for the bboard is given together with a list of routines to call when an operation it supports should be executed and an indication of the consistency level. A descriptor to use when operations are issued is returned.
terminate *bb:p_name*	Terminate the session. If the same processes wishes to interact further with the bboard it will need to enroll again.
initiate *bb:p_name*	Initiate a new session for the designated bboard and the process with unique name p_name. If a previous session is still active, it is automatically terminated first.

Basic invocation

result := {*guard*} *bb.op (args)*	When the guard is satisfied, invoke the operation on bboard *bb* and store the result in *result*. The object(s) to be accessed are identified in the arguments, which the bboard mechanism does not interpret. In addition to the guard syntax from Sec. 3, the special guard *timeout(secs)* is supported; it is satisfied after *secs* seconds have elapsed. The bboard specifier *bb* may be omitted if the process is accessing only one bboard.

Qualified invocations

rvar:: <*basic invocation*>	After performing the invocation, the request name that it was assigned is stored in *rvar*.
rvar:: **async** <*basic invocation*>	The operation is performed asynchronously. Its request name is stored in *rvar* and execution continues immediately.
result := **join** *rvar*	Execution pauses until the asynchronous invocation to which *rvar* corresponds terminates. The result is then stored in *result*. **Join** can only be executed by the process that originated an asynchronous invocation.
cancel *rvar*	The designated invocation is canceled if it has not yet been executed. **Cancel** can only be executed by the process that originated an asynchronous invocation.

Specification of operations

op *(r: request_name, args)*	The bboard performs an operation by invoking the user-supplied routine **op**, passing it the associated request name and any

	arguments provided by the caller.
Other system functions	
alive (p_name)	*True* if session *p_name* is still active and *false* otherwise.
failed (p_name)	*True* if process that initiated session *p_name* failed while the session was still active, *false* if not.
proc (r_name)	Returns the name of the process that invoked the request *r_name*.
oper (r_name)	Returns the name of the operation invoked in request *r_name*.
flush ()	Delays the process until all of its (non-blocked) requests have been executed in every history.
Client-supplied functions	
deadlock (r: request_name)	The bboard calls this routine, which should be supplied by the client, if it detects a deadlock involving some pending request *r_name* that the client issued. The routine would presumably cancel the request or take other appropriate action.

When we wish to distinguish the bboard interface within a client from the client process itself, we will refer to the bboard interface as a bboard *component*. The distinction is that the code executed by a component is supplied as part of the bboard facility whereas the code executed by a client is provided by the programmer who built that particular bboard. Thus, a *client* invokes guarded operations using the syntax given above and the guard notation defined in Section 3. The corresponding bboard *component* is responsible for packaging the invocation into a message, dispatching the message to the other components of the bboard, delaying the execution of an incoming invocation until its guard is satisfied, and then invoking the operation with the appropriate arguments. The programmer who implements a bboard provides client code for the operations that it supports, in the form of procedures with value/result semantics. These procedures must create and maintain any data objects that are to be used to record the bboard state.

6.3. Underlying communication primitives

Our bboard implementation is built using a set of reliable broadcast primitives that support *virtually synchronous process groups*: sets of processes that observe consistent orderings on events that include communication with group members, failures of members (or the sites where they reside), and recoveries [Birman-a]. The basic approach is to convert these types of physical events into *virtual* ones that satisfy global consistency properties. For example, since the system state that results when a

process fails *before* receiving a message is not distinguishable the one in which it failed *after* receiving the message but before acting on it, claiming that a message was delivered to a failed process is reasonable. Our approach differs from previous work on reliable communication [Schneider-b] [Chang] [Cristian] in several ways. Most notably, although virtually synchronous process groups have dynamically changing membership, an addressing mechanism is supported that can send a broadcast to all members of a group even when membership is changing. Most prior work has addressed message delivery to relatively static sets of sites. Our approach also distinguishes a hierarchy of possible delivery ordering constraints, the cost (in latency and number of messages sent) of the respective protocols relates directly to the degree of synchronization each provides. Other work has tended to fix the level of synchronization, but this imposes inefficiency on some applications.

The essential idea behind the virtually synchronous process group approach is illustrated in Figure 2. Here we contrast a conventional space-time diagram of a distributed system with one in which virtually synchronous broadcasts are used for interactions between clients and process groups. Notice how simple it is to reason about the latter style of execution in contrast to the former one. This observation is developed in [Birman-b].

The broadcast primitives are given below. In what follows, when we say that a broadcast b is ordered before (resp. after) a broadcast b', we mean that the message sent in broadcast b is delivered before (resp. after) the message sent in broadcast b' at all overlapping destination processes. A broadcast b is said to be ordered relative to another broadcast b' if b is ordered either before or after b'. For brevity, we will use the term "broadcast" to refer to the message sent in a broadcast. All the primitives have *guaranteed delivery*: If a broadcast is delivered to any of its destinations, it will eventually be delivered to all of them.

[1] *Group broadcast (GBCAST).* A broadcast made using the *GBCAST* primitive is ordered relative to *all* other broadcasts. If two *GBCAST*'s are initiated by the same process, the one initiated first is ordered before the one initiated second. Additionally, when a process fails, operational processes receive a simulated *GBCAST* that appears to be sent by the failed process, informing

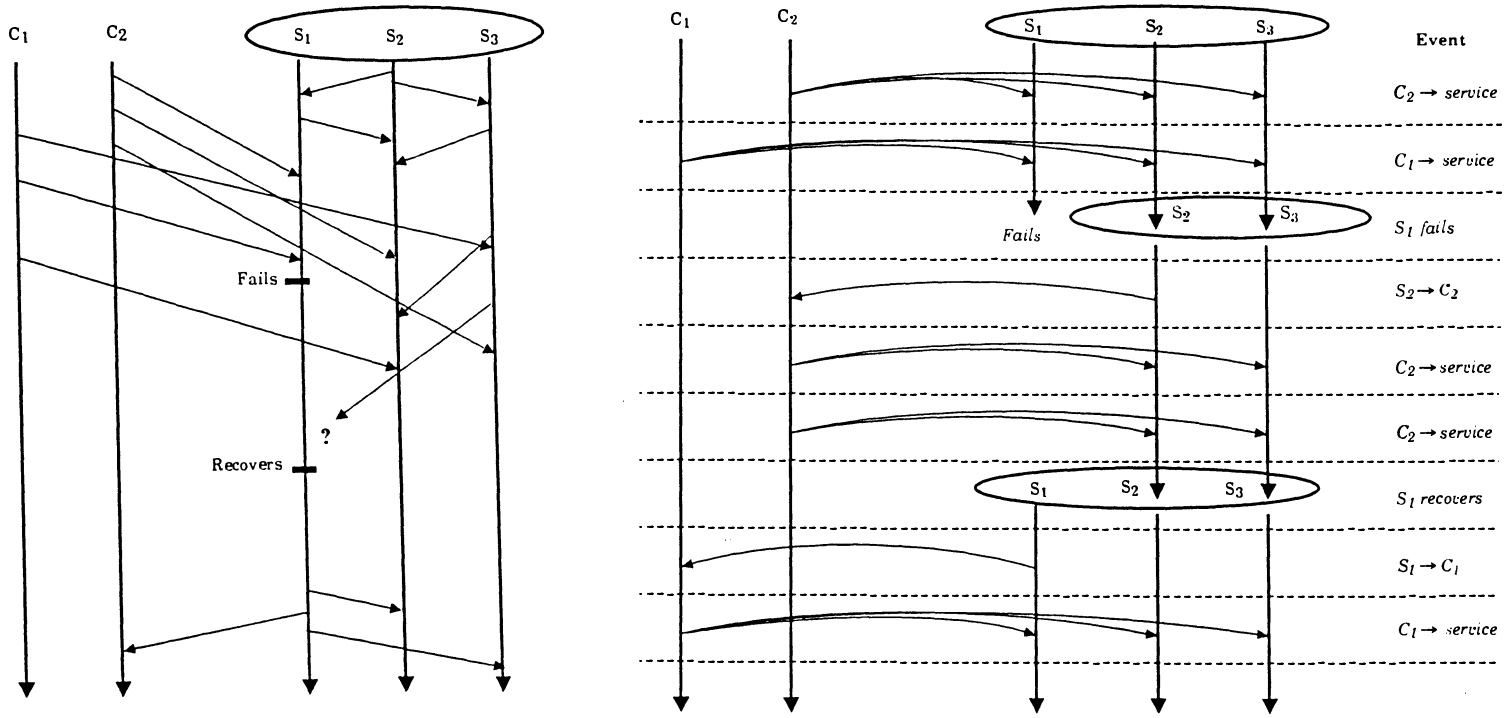

Figure 2: A conventional and virtually synchronous space-time diagram for a distributed system

them of its failure. We call such a *GBCAST* a *failure GBCAST*. A failure *GBCAST* has the further property that is is ordered after any other broadcast from the failed process.

[2] *Minimal broadcast (MBCAST).* This type of broadcast is ordered only relative to *GBCAST*.

[3] *Fifo broadcast (FBCAST).* This type of broadcast delivers messages from the same processes in the order they were sent. In addition, *FBCAST* is ordered with respect to *GBCAST*.

[4] *Causal broadcast (CBCAST).* As in [Lamport-b], we define an *information flow* relation $\rightarrow_I$ as the transitive closure of the relation given by the following rules:

[I1] $b \rightarrow_I b'$ if broadcast b' was issued after b by the same process.

[I2] $b \rightarrow_I b'$ if b was received by some process that subsequently issued b'.

Then, *CBCAST* deliveries are ordered with respect to *GBCAST*, and if $b \rightarrow_I b'$, then b will be

ordered before b'. If two broadcasts are not related under $\rightarrow_I$, then their delivery ordering may vary at different destinations.

[5] *Atomic broadcast (ABCAST).* This type of broadcast satisfies all the properties of *CBCAST* and also ensures that all *ABCAST*'s are ordered relative to one another.

In a bboard implementation, a process will often issue broadcasts asynchronously. This is done by issuing the broadcast and then waiting until a response is received from a single destination (usually within the sending process itself), and then continuing the main thread of computation while the broadcast completes asynchronously. Notice that chains of potentially causally related asynchronous *CBCAST*'s and *ABCAST*'s could now arise. These types of primitives therefore ensure that delivery is *prefix closed:* if broadcast $b \rightarrow_I b'$, then if b' is delivered to any destination (and that destination stays operational), then b will be delivered to all of its operational destinations. Thus, gaps in a sequence of causally related broadcasts are not permitted.

Although the primitives have costs that vary depending on the degree of ordering they provide, none is very costly even in our unoptimized implementation. The cheapest primitives, *CBCAST*, *FBCAST* and *MBCAST* are one phase broadcast protocols using an underlying acknowledgement and garbage collection scheme to achieve reliability. These protocols can achieve particularly high throughput by piggybacking. *ABCAST* is a two-phase protocol, requiring acknowledgements from remote processes before message delivery can occur, and *GBCAST* requires a third flushing phase that can be merged into the first phase, but will still slow the protocol down. However, *GBCAST*'s are used infrequently. We believe that our protocol implementations will be competitive with any foreseeable alternatives, including broadcast protocols that achieve much weaker correctness properties.

6.4. Basic bboard implementation

Within the above framework, implementation of the bboard package is straightforward. Each bboard is created with a single component. When a process wishes to enroll in a pre-existing bboard, *GBCAST* is used to broadcast its intention. On reception of an enrollment *GBCAST* a *coordinator-cohort* algorithm is used to transfer the bboard state to the new client. In such an algorithm, one

bboard component is in charge of the state transfer and the others back it up; one restarts the state transfer should the coordinator fail [Birman-b]. The client code provides routines for packing and unpacking the state into messages, a systems engineering aspect that we will not address in greater detail here. Since *GBCAST* is totally ordered with respect to other broadcast events, all components have received the same messages and hence are in equivalent states when the state transfer takes place. *GBCAST* is also used to inform bboard components when a component fails, and they use this information when evaluating guards.

When a component of the bboard is presented with an invocation, the following occurs. First, a request name is generated for the invocation. Next, the information corresponding to the operation to perform, the arguments, the guard, and the generated request name are packaged into a message and broadcast to all components (including the one that issued the invocation). We denote the sending of such a message for a request r as $send(r)$. The broadcast primitive used depends on the consistency level of the bboard: *ABCAST* is used for total consistency, *CBCAST* for causal consistency, *FBCAST* for fifo consistency, and *MBCAST* for weak consistency. The caller then blocks until the operation is executed as described below and it receives the result, except if the invocation is asynchronous, in which case the caller resumes execution immediately.

When a request is delivered to a component (an action we denote by $recv(r)$), the message received is added to a *wait queue*, which preserves the order in which messages are delivered. Messages in the wait queue of a component are processed as follows. Starting at the head of the queue (the earliest delivered message), the guard is evaluated to see whether a set of operations satisfying its guard have been executed on the local copy of the bboard. If the guard is satisfied, the operation in the message is executed by invoking the appropriate procedure with the given arguments. If the invocation was issued by the local client, the result of the execution is returned to it, otherwise the result is ignored. The message is then removed from the queue. If the guard is not satisfied, the next message in the queue is examined. Each time an operation is executed, the guards for previously examined invocations may become satisfied, hence the wait queue is reexamined from its head. When no more operations in the wait queue can be executed, the bboard component waits for the next message to be

delivered, and processes it as above.

When a process p fails or terminates, a *GBCAST* is used to inform the other bboard components of the event *term*(p). The reception of such a message *term*(p) is handled as follows. First, all requests that were invoked by p are moved to the head of the wait queue (without changing their order relative to one another). Then *term*(p) is "executed" by making the information about p's failure available to the client (the call *alive*(p.*name*) will now return *false*), and reevaluating guards in the wait queue based on this information.

In our initial implementation, all bboard components will save the request name of each executed operation until the process that issued it terminates. The request names generated by a terminated process are discarded, although the termination status (whether or not the process failed) is saved indefinitely. This approach is simple and should entail low overhead, provided that individual processes do not execute huge numbers of bboard operations. Possible optimizations are discussed below.

Notice that a large class of deadlocks can be detected by constructing a process "wait-for" graph from the guards of delayed operations. We are implementing an algorithm to do this as part of the bboard component code; it triggers calls to the clients' deadlock exception handlers when necessary.

Finally, the *cancel* operation is transmitted using *GBCAST*. This means that all components have received the same set of invocations at the time a *cancel* request is received. Hence, unless the invocation has already been performed everywhere, it is canceled at all the components.

6.5. Progress

Each client executes operations sequentially in the order the corresponding messages are removed from the wait-queue. We assume that there are no errors in the (user-supplied) implementations of bboard operations and hence once an operation has been scheduled for execution, it will complete successfully. We show that the progress condition described in Section 3.5 is satisfied by our implementation: If the guard for a request can be satisfied at any process, it will eventually be executed at all of them. Assume that the guard for a request r is satisfied at a component p. The implementation is

such that r will be immediately executed at p. Each request is broadcast to all components, and the broadcast primitives have guaranteed delivery. So, every component will eventually receive a message relating to r, the requests that satisfied its guard at p, the requests that satisfied the guards of those requests, etc. Hence, r will be executed everywhere.

6.6. Correctness

We show that even though the order in which operations are executed may differ from client to client, the execution yields the desired level of consistency. Recall that in our formal model a process history H_p continues after process p terminates or fails. This means that if a site crashes, we just pretend that the local bboard component still receives messages and the client still executes the corresponding operations. Since every bboard component only serves its local client (which has terminated in this case) no contradictions will arise. For simplicity we also assume that all processes join the bboard simultaneously at the time of the *INIT* event.

We will first present correctness proofs ignoring the fact that the order of requests in the wait queue may be changed after a termination message is received. Afterwards we will show that the consistency of the bboard is preserved under this reodering.

6.6.1. Weak consistency

In a weakly consistent bboard, messages about invocations are transmitted using *MBCAST*, which is ordered relative to the *GBCAST* messages used to transmit failure information. Consider a history generated by a run of such a system.

The resulting set of process histories satisfies well formedness [WF] by construction. The guard evaluation mechanism described above clearly satisfies [GS]. The implementation satisfies [T] because *GBCAST* is totally ordered relative to other types of broadcasts. We discuss this in more detail since a similar argument will be used when we examine causal consistency and cleanup ordering.

The properties of *GBCAST* ensure that all termination events appear in the same order in every process history. Now consider an execution event in H_q such that $ex(r) \leq_q term(p)$. *GBCAST* ensures

that the request message for r will be received everywhere before the termination message of p. However, since guards are evaluated locally at each component it might be conceivable that another process q' finds $r.guard$ not satisfied and executes r after $term(p)$, which would violate [T]. But $r.guard$ was satisfied at q by a set S of termination and execution events preceding $ex(r)$ at q. The termination and request messages related to these events also precede the $GBCAST$ of $term(p)$ and are therefore received by q' before $term(p)$. The same holds for events at q that were necessary to satisfy the guards of events in S, etc. It follows that by the time q' receives the message $term(p)$ it will be able to execute the same requests that were executed at q prior to $term(p)$. Since the implementation executes a request as soon as it is able to satisfy its guard, [T] is satisfied. Notice that it is immediate from this that the implementation also satisfies [P].

6.6.2. Fifo consistency

To show that the implementation achieves fifo consistency, consider two invocations $inv(r) \leq_p inv(s)$ by process p, where $r.guard = s.guard$. If $FBCAST$s are used to transmit requests, then every other component will receive them and place them on the wait queue in the order they were invoked. Since they have identical guards, their guards will be satisfied at the same time, and they will be executed in the required order.

6.6.3. Causal consistency

For causal consistency we have to show that if $ex(r) \rightarrow_C ex(s)$ then r will be executed at every client before s. We will prove this by induction on the length of the chain of events that establish the causal relationship between $ex(r)$ and $ex(s)$. Since $\rightarrow_C$ and $\leq_p$ are both reflexive, the base case, $r=s$, is trivially satisfied. Now, for the general case, say that s was invoked by process p. Since $ex(s)$ is directly causally related only to $inv(s)$, and since an invocation event is directly causally related only to execution events that precede it at the process that issued it, we must have

$$ex(r) \rightarrow_C ex(r') \leq_p inv(s) \rightarrow_C ex(s),$$

where r and r' may be the same. By our induction hypothesis we can assume that at every process $q \in P$, $ex(r) \leq_q ex(r')$. Therefore we only have to show that $ex(r') \leq_q ex(s)$, for all $q \in P$. Since

$ex(r') \leq_p inv(s)$, p must have received r' before it sent out the request for s. This establishes an information flow $send(r') \rightarrow_l send(s)$. The properties of *CBCAST* ensure that any process q will receive r' before s. Then q would have executed r' before s unless it decided that $r'.guard$ was not satisfied. But $r'.guard$ was satisfied at p by a set S of events preceding $ex(r')$ at p. By an argument similar to that in Section 6.6.1, we can show that by the time q receives s, it will also have received all events needed to satisfy $r'.guard$, and will hence execute r' before s.

6.6.4. Total consistency

The argument in this case is trivial because *ABCAST* satisfies all the properties that *CBCAST* does, but also delivers messages to the wait-queue in the same order at each component. Each component follows the same algorithm to process messages in their wait-queues; hence all operations are executed in the same order everywhere.

6.6.5. Cleanup Ordering

In the case where a *term* event is received, the guard evaluation algorithm changes the order of events in the wait-queue. We will now show that (i) this does not violate consistency, and (ii) leads to an implementation that satisfies [CO].

Notice that the reordering of requests that takes place when the event $term(p)$ is received does not change the relative order of operations invoked by the same process. Therefore fifo consistency is preserved. Now consider an event e that was moved ahead of an earlier event e' in the wait-queue at q because of the reordering. (Clearly, e' cannot be an invocation by p.) We will prove by contradiction that e does not causally depend on e'. Assume that e does indeed causally depend on e', that is, there is a causal chain of events linking the execution of e' at some component, say q', to the invocation of e at p. Consider the case where e' was executed at q' *before* the failure *GBCAST* for p was received there. We can show, as in Section 6.6.1, that messages relating to the events that satisfied the guard of e' at q' must have arrived at q before the failure *GBCAST* is received at q. Hence, e' would have already been executed at q and would not even be on the wait-queue in the first place. The other case,

where e' is executed *after* q' receives the failure *GBCAST*, also leads to a contradiction. The process p invokes e before it terminates. Hence the execution of e' must occur after the invocation of e in real time, which contradicts the assumption that e causally depends on e'. Hence, moving e ahead of e' does not contradict causal consistency. Finally, since *GBCAST* is ordered relative to all other broadcasts, and since every bboard component reorders requests in the same way after a termination event, total consistency is preserved.

Recall that a termination message is processed only when there are no executable requests in the wait queue. Since all requests invoked by p are delivered before $term(p)$, we immediately see that [CO(i)] is satisfied. Furthermore, when the event $term(p)$ is received, all requests invoked by p are moved to the head of the wait-queue. Therefore cleanup operations posted by p are executed as soon as possible, as required by [CO(ii)].

6.7. Optimizations

We now consider two optimizations to the above algorithm. First, in the case of read-only operations, it is not necessary to broadcast the requests to all bboard components, provided that the guards of read-only requests do not refer to request names of other read-only requests (a form of synchronization that serves no obvious practical purpose). Since a read-only operation will not change the bboard state, and its result is needed only at the component where the invocation occurred, such an operation can be placed directly on the wait-queue at the local site, and correctness will not be compromised. However, the issue now arises of how guard satisfaction can be determined in the case where a read-only request r is referenced in the guard of some other (non read-only) request r'. Since r' is not read-only, however, it will be broadcast to all bboard components including the one where r was executed. That component will discover that $ex(r)$ satisfies some part of the guard for r'. Rather than satisfying the guard locally, it broadcasts the request name for r to all bboard components, including itself, using the broadcast type appropriate to the bboard. On receiving a message containing request names for read-only operations, all recipients apply this information to the guards of operations on the wait-queue. Since the information flow relation is preserved, the correctness arguments above apply. This

optimization will be supported by our implementation.

A second possible optimization would allow a process to cache subsequences of the request names generated by other bboard components, while keeping the complete sequence of its own request names. An interrogation mechanism could then be used to inquire about request names, operating much like the mechanism described above. Our implementation will not initially support this optimization.

6.8. Checkpointing the bboard state

If a checkpoint/rollback algorithm is used in an arbitrary nondeterministic system, and the system enters a state in which messages that were sent *prior* to rolling back are received *after* rollback, consistency could be violated (a message may not be reissued in the state that results after rollback) [Koo]. In a bboard, however, checkpointing is straightforward because *GBCAST* ordered relative to all other bboard events. To establish a checkpoint, a *GBCAST* is issued to invoke a checkpointing operation in each client, which writes a checkpoint and a timestamp to stable storage. If all the clients of a bboard fail, clients that recover run an algorithm to determine the last ones that failed [Skeen]. Members of this set recover from their last checkpoint (unless they failed while writing it); others must re-enroll in the bboard. An implementation of this mechanism is included in $ISIS_2$ [Birman-b].

7. Applications

The bboard paradigm is broadly applicable. For example, the chess bboard discussed in the introduction might be implemented as a causally consistent bboard with operations to post and read problems and data. This would guarantee that if an expert process starts working on a problem, it will also find relevant data and previously posted solutions to relevant subproblems. Similarly, existing advisory communication primitives can easily be recast into the bboard framework. For example, the Linda S/Net kernel provides four primitives, *IN*, *OUT*, *READ* and *EVAL*, to manipulate a collection of tuples comprising a shared memory [Carriero]. *OUT* adds a tuple to the tuple space. *IN* finds a tuple that matches some pattern and removes it from the tuple space. *READ* performs the same operation, but without removing the tuple. *EVAL* adds an unevaluated tuple whose evaluation begins as soon as

the tuple enters the tuple space. These operations could easily be implemented in a totally consistent bboard. The *message bulletin-board* mechanisms described in [Ahamad] could likewise be implemented, using a weakly consistent bboard.

Below, we show how three well known problems can be solved using bboards. A token passing example demonstrates the use of totally consistent bboards to achieve fair, efficient mutual exclusion on a shared resource. A deadlock detector illustrates how fifo bboards might be used to maintain a non-trivial distributed data structure; a deadlock check will discover deadlock if and only if one is really present. Finally, a bboard implementation of a transactional replicated file shows how totally and causally consistent bboards can be combined to implement an efficient transactional database system.

Several of the applications consist of two parts: an *interface* that callers use to access the bboard, and the bboard operations themselves. The bboard designer can employ any desired access rule in the interface code, and if callers never access the bboard directly, it will be enforced. We envision that programmers will generally implement bboards within abstract data types, in which case this structuring is completely natural.

7.1. Token passing

A distributed token can easily be implemented using an totally consistent bboard. In the implementation we describe below, a process attempting to acquire the token is given it immediately if the token is free. If the token is in use, processes waiting for it queue up and compete to acquire the token after it is released.

The token is represented by a record containing a field *holder* that stores the request name of the event that caused the token to be acquired by the current holder (Φ if there is no holder). Any additional fields needed by the application can be added to the token structure. The operations on the bboard are *grab* and *free*, and are modeled after the usual implementation of semaphores using atomic instructions [Peterson]. *Grab* is used while acquiring the token. If the token is free, invoking *grab* causes the *holder* field to be set to the request name corresponding to the invocation. If the token is in

use, *grab* does nothing, and the caller deduces that it must wait by examining *proc*(holder) (a system function, which gives the name of the process that made the request) after it returns. *Free* is used when releasing the token, and sets the value of the *holder* field to Φ.

A process wishing to acquire or release a token does so using the interface routines *acquire* and *release*, which in turn invoke the bboard operations described above. The correctness of the solution follows from the fact that invocations on the bboard are totally ordered; hence not more than one process acquires the token at a time. Notice how the guard is used to avoid busy waiting when an attempt is made to acquire the token while it is held by some other process. If the first attempt to grab the token fails, each iteration of the while loop delays the next grab() operation until the process that holds the token *proc*(token.holder) has either failed or released it; either event being subsequent to the acquisition event (token.holder). The solution is slightly inefficient in that every process issues a *grab()* operation each time a *free()* is done; the design of a more efficient solution (for example, one that maintains a queue of waiting processes so that a process only issues a *grab()* if it is the "next" holder of the token) is left as an exercise.

```
-- Interface procedures (used to issue requests to the bboard) --
var token : token_type;

procedure acquire()
begin
  grab();
  while proc(token.holder) # my_pname
    { after(token.holder)[proc(token.holder),  free] } grab();
end acquire;

procedure release()
begin
  free();
end release;
```

```
-- The actual bboard operations --
function grab(r : request_name)
begin
  if token.holder = Φ or not alive(proc(token.holder)) then
    token.holder := r;
end grab;

procedure free(r : request_name);
begin
  token.holder := Φ;
end free;
```

7.2. Deadlock detection

Deadlock detection is an example of a non-trivial problem that has an elegant solution when expressed in terms of our bboard approach. Consider the *RPC deadlock detection* problem. A collection of processes are interacting by remote procedure call. A process that has issued such a call must wait for the destination process to reply before it can proceed. Hence, a circular chain of calls results in a deadlock. A waiting process periodically checks for deadlock and if it detects a deadlock that includes it, it cancels its request.

One way to solve this problem is to implement bboard operations to send and receive RPC messages and to rely on the deadlock exception mechanism provided by the bboard interface. However, a more elegant solution that makes no assumptions about how the RPC abstraction is implemented uses a fifo bboard to maintain a *wait-for graph* G, consisting of a set of nodes corresponding to the processes with wait-for edges inserted between two nodes if a process is waiting for an RPC response from another process. A process p will *insert* an edge (p, q) before it makes a remote call to process q. Upon completion of the request q will *delete* that edge just before it returns a result to p. Deadlock is detected if there is a cycle in G. We will show that fifo consistency is sufficient to ensure that there will be a cycle in G if and only if the system is deadlocked. We can even allow processes to use asynchronous calls for inserting and deleting edges in G. The solution we get is very cheap since no process ever has to wait for a bboard operation to complete.

The bboard supports the operations *insert*, *delete*, and a read only operation to *check* for a cycle in the graph, as shown below. We assume that processes keep a form of *call serial number*, *csn* for short, that makes it possible to distinguish multiple calls between the same pair of processes. A process inserting or deleting an edge passes the *csn* of the RPC as a parameter of the operation. For every pair of processes, the bboard also stores the *csn* of the most recent RPC for which an edge was added or deleted. This allows the bboard to detect and ignore an insertion request that arrives after the corresponding edge has already been deleted. Similarly deletion requests are ignored if an edge referring to a more recent RPC is present.

To prove that this bboard is correct, we must establish that if a deadlock occurs it will eventually be detected and that if a wait-for cycle is detected in G, it corresponds to a deadlock in the real system. Because the bboard is distributed, it is not immediate that these properties hold: some deadlock detec-

---

```
-- Deadlock Detection Bboard
  – The wait-for graph G is represented by a two dimensional matrix E
  – containing one entry for each possible edge (p, q) in G.
  – The request names returned by the bboard are not used in this example.

procedure insert(r : request_name; p, q : process_name; csn : call_serial_number);
begin
   if csn > E[p, q].last_csn then
      E[p, q].edge_present := true;
      E[p, q].last_csn := csn;
end insert;

procedure delete(r : request_name; p, q : process_name; csn : call_serial_number);
   if csn ≥ E[p, q].last_csn then
      E[p, q].edge_present := false;
      E[p, q].last_csn := csn;
end delete;

procedure check(r : request_name, p : process_name): readonly;
begin
   if p is contained in some cycle in G then
      return true;
   else
      return false;
end check;
```

---

tion algorithms tend to find *phantom* deadlocks, which result when wait-for edges from different stages of a computation are assembled into a single, inconsistent snapshot, representing a system state that never occurred. For example, Q might at some time have waited for P, and P may now be waiting for Q. If the old wait-for edge representing Q waiting for P is included into G, a phantom deadlock would be discovered between P and Q and P might abort itself unnecessarily.

The correctness proof is as follows. Assume a deadlock has occurred, involving a set of processes $p_1, p_2, \cdots p_n$, such that p_i waits for an RPC to be answered by p_{i+1} and p_n waits for p_1. Then each p_i has invoked an operation to insert the edge (p_i, p_{i+1}) into G. Since p_{i+1} is deadlocked it will never invoke the operation that would delete the edge (p_i, p_{i+1}) from G. Hence after all insert operations have been executed, a cycle will be present in G.

Now assume a cycle $p_1 \rightarrow p_2 \rightarrow \cdots \rightarrow p_n$ is detected in G. Let C_1 be the remote call for which p_1 inserted the edge (p_1, p_2) into G; similarly C_2 is the RPC corresponding to (p_2, p_3). We assert that if p_2 ever returns a result for C_1, it will do so only after having received an answer for C_2. The assertion is obviously true if p_2 made the call C_2 before receiving the request for C_1 since we assume that processes are blocked while they are waiting for an RPC. The same is true if p_2 made the (nested) call C_2 while it was processing C_1. A contradiction would arise only in the case that p_2 answered C_1 before making the call C_2. But then p_2 must have invoked *delete*(p_1, p_2) before *insert*(p_2, p_3). and fifo consistency implies that these operations will be executed in that order. Hence G would never contain edges corresponding to C_1 and C_2 at the same time. By the same argument we show that p_3 will not respond to C_2 until it received an answer for the call C_3 corresponding to the edge (p_3, p_4), and so forth. It follows that $p_1, p_2, \cdots, p_n$ are deadlocked.

7.3. Serializable access to concurrently updated data items

Using an totally consistent bboard together with a causally consistent bboard, a transactional mechanism supporting asynchronous updates to a replicated database can be implemented. Each transaction consists of a *begin* operation followed by sequence of *read* and *write* operations terminated by a *commit* or *abort*, with the usual semantics. A transaction that fails before committing is automatically

aborted. Two-phase locking is used to achieve serializability, and a write-ahead log to implement abort [Gray].

The approach is as follows. A totally consistent bboard, denoted *LOCKS*, stores a set of lock variables. These are acquired just like the tokens of the previous example, but are released by the commit or abort of the transaction that holds the lock (for brevity, only the interface code is given below). A causally consistent bboard, denoted DB, stores the log and database items. The *begin* operation posts an asynchronous cleanup operation; it will be described shortly. *Read* returns the current value of a variable. *Write* first logs the old version of the data item being updated and then performs the update. Because the log record is written before the update is done and the bboard supports causal consistency, the semantics of a write-ahead log are achieved: Regardless of how asynchronously the update executes, log records are always written everywhere before the corresponding update is done. Finally, *commit* logs a commit record and then terminates the session, while *abort* just terminates the session. Termination enables the cleanup operation, which checks to see if the transaction committed (termination due to a failure is treated as an abort). If not, it rewrites the old values of any variables that have been changed. Then it deletes any log records written by the terminated process. Completion of the cleanup operation, in turn, enables the release of any locks acquired by the transaction; locking is thus two-phase. Moreover, since this establishes a causal chain between the termination of a process and any subsequent process that acquires a lock from it, subsequent processes will observe the updates that have been done even if these are asynchronous. A formal treatment of this type of causal chaining is given in [Joseph].

The code for the interface used to communicate with the DB bboard and the bboard operations themselves is given below. As in the case of the token, the interface procedures are not really part of the bboards, but rather are used to communicate with them in a stylized fashion. We omit the detailed management of the log data structure, which is implemented by routines log_write, log_delete, restore_from_log and not_logged. We also use a very informal notation to pass references to data items, although in practice this would be replaced with a symbolic addressing mechanism.

```
--- Transactional operations: interface ---

procedure BEGIN() returns request_name   -- post a cleanup operation, return its request name
begin
  termevent:: async { [my_pname, TERM] } log_cleanup(my_pname);
  return termevent;
end BEGIN;

procedure LOCK(termevent : request_name, x : data_item)
begin
  ACQUIRE(termevent, x.lock);                -- see below.
end LOCK;

procedure READ(x : data_item)
begin
        return read(x);
end READ;

procedure WRITE(x : data_item; value: data)
begin
  log_append("write", "x", x.value);         -- log old value
  write(x,value);                            -- update x
end WRITE;

procedure COMMIT()
begin
   log_append("commit", "p", my_pname);      -- log commit record
   terminate my_pname;                       -- end session.
end COMMIT;

procedure ABORT()
begin
  terminate my_pname;                        -- just end session.
end ABORT;
```

```
--- Operations on the LOG part of the bboard --
procedure append(r : request_name, oper : log_operation, item : data_item, value : data)
begin                                                -- log commit requests and first write request
  if oper = "commit" or (oper = "write" and not_logged("write", item)) then
     log_write(oper, item, value);
end append;

procedure cleanup(r : request_name, p_name : process_name)
  if not_logged("commit", p_name) then               -- committed?
    restore_from_log(p_name);                        -- no, roll-back DB
  delete_log_records(p_name);                        -- done.
```

```
end cleanup
```

```
--- Operations on the DB part of the bboard ---
procedure readonly read(r : request, x : data_item)
begin
  return x.data;                                    -- just return the value
end read;

procedure write(r : request_name, x : data_item, value : data)
begin
  x.data := value;                                  -- just set the value
end write;
```

```
--- The interface to the LOCK bboard ---
procedure ACQUIRE(termevent : request_name, lock : lock_type)
var temp : lock_type;
begin
  -- Post an asynchronous operation to release the lock after commit/abort.
  async { termevent } free(lock)
  acquire(lock.mutex);
end ACQUIRE;
```

The implementation needs some discussion. First, examine the asynchronous guarded operations that are issued for log cleanup and lock release. For each session, *begin* creates an asynchronous log cleanup operation that waits until the session ends. It does this using a guard that will not be enabled until the invoking process terminates. The request name of the cleanup operation is noted in the variable *termevent*. Later, when locks are acquired, they post an asynchronous release operation guarded by {termevent}. That is, *after* the cleanup, lock release events become enabled. These release any locks that the process held prior to ending the session. The locking algorithm itself can be based on the token passing example, and is omitted for brevity. Since our bboard implementation satisfies [CO] we see that if a failure occurs, the cleanup operation will not execute until any pending updates have completed.

A replicated database implemented in the above manner should perform quite well. Updates will be asynchronous as in [Joseph], and correctness and fault-tolerance will follow from the fact that the DB is causally consistent. In fact, good performance using these techniques was measured in our previous work, and reported in [Birman-c] [Joseph]. A weakness of the above implementation is that only one class of locks is supported, hence although reads are local, there is no notion of a local read-lock. This limitation could certainly be overcome in a more sophisticated implementation.

8. Conclusions

We have proposed that advisory bulletin boards be considered as an alternative to more imperative styles of interaction in fault-tolerant distributed systems, and illustrated the approach with a series of examples that are straightforward when implemented as bboards and more complex when implemented using other programming methodologies. Moreover, the model itself is interesting as a tool for formalizing and reasoning about the consistency when processes interact using fault-tolerant broadcasts. Our work leaves open several problems. For example, one might try to determine the weakest level of consistency applicable to a given problem, and to identify the rules governing the combination of bboards as was done in Section 7.3. The guard evaluation algorithm also merits further study.

We do not view bboards as the only facility to be used in distributed systems, and indeed continue to believe that the mechanisms proposed in our previous work (resilient objects, virtually synchronous process groups) can play an important role. Rather, it is our feeling that if a diversity of fault-tolerant programming tools can be provided to distributed systems architects, then they will ultimately find it as easy to build fault-tolerant distributed software as it currently is to build fault-*intolerant* non-distributed software.

9. Acknowledgements

We are grateful to the members of the ANSA project and to the students who attended Sam Toueg's distributed systems class in fall 1986, for their many thoughtful comments.

10. References

[Ahamad] Ahamad, M. and Bernstein, A. An application of name based addressing to low level distributed algorithms. *IEEE Trans. on Software Engineering* **TSE-11**, 1 (Jan 1985), 59-66.

[Birman-a] Birman, K. and Joseph, T. Reliable communication in the presence of failures. *ACM Transactions on Computer Systems* **5**, 1 (Feb. 1987).

[Birman-b] Birman, K. and Joseph, T. The principle of virtual synchrony in distributed systems. *Forthcoming.*

[Birman-c] Birman, K. Replication and fault-tolerance in the ISIS system. *Proc. 10th ACM SIGOPS Symposium on Operating Systems Principles.* Orcas Island, Washington, Dec. 1985, 79-86.

[Birrell] Birrell, A., Nelson, B. Implementing remote procedure calls. *ACM Transactions on Computer Systems* **2**, 1 (Feb. 1984), 39-59.

[Carriero] Carriero, N., Gelernter, D. The S/Net's Linda Kernel. *10th ACM Symposium on Operating Systems Principles*, appearing as *Operating Systems Review* **19**, 5 (Dec. 1985), 160.

[Chandy] Chandy, M., Lamport, L. Distributed snapshots: Determining global states of distributed systems, *ACM Transactions on Computer Systems* **3**, 1 (Feb. 1985), 63-75.

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

[Cheriton] Cheriton, D. Preliminary thoughts on problem oriented shared memory: a decentralized approach to distributed systems. *Operating Systems Review* **19**, 4 (Oct. 1985), 26-33.

[Cristian] Cristian, F., Aghili, H., Strong, R., Dolev, D. Atomic broadcast: From simple message diffusion to Byzantine agreement. IBM Technical Report RJ 4540 (48668) 12/10/84.

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

[Jefferson] Jefferson, D. Virtual time. USC Technical report TR-83-213, University of Southern California, Los Angeles, May 1983.

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

[Koo] Koo, R., Toueg, S. Checkpointing and Rollback-Recovery for Distributed Systems. *Proc. 1986 Fall Joint Computer Conference*, Austin, TX (Nov. 1986), 1150-1158. *To appear: IEEE Transactions on Software Engineering.*

[Lamport-a] Lamport, L. Using time instead of timeout for fault-tolerance in distributed systems. *ACM TOPLAS* **6**, 2 (April 1984), 254-280.

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

[Peterson] Peterson, J. and Silbershatz, A. *Operating System Concepts*, 2nd edition, Addison-Wesley Publishing Company, 1985.

[Schneider-a] Schneider, F. Synchronization in distributed programs. *ACM TOPLAS* **4**, 2 (April 1982), 179-195.

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

[Skeen] Skeen, D. Determining the last process to fail. *ACM Transactions on Computing Systems* **3**, 1, Feb. 1985, 15-30.

[Strom] Strom, R. and Yemini, S. Optimistic recovery in distributed systems. *ACM Transactions on Computing Systems* **3**, 3 (April 1985), 204-226.