Exploiting Virtual Synchrony in Distributed Systems

Kenneth P. Birman*
Thomas A. Joseph

TR 87-811
February 1987

Department of Computer Science
Cornell University
Ithaca, New York 14853-7501

*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. The views, opinions and findings contained in this report are those of the authors and should not be construed as an official Department of Defense position, policy, or decision.

EXPLOITING VIRTUAL SYNCHRONY IN DISTRIBUTED SYSTEMS[1]

Kenneth P. Birman, Thomas A. Joseph

Department of Computer Science,
Cornell University, Ithaca, New York 14853.

ABSTRACT

We describe applications of a new software abstraction called the *virtually synchronous process group*. Such a group consists of a set of processes that cooperate to implement some distributed behavior in an environment where events like broadcasts to the group as an entity, process failures, and process recoveries appear to occur synchronously. The utility of this approach is illustrated by solving a number of classical problems using our methods. Many are problems that are quite difficult in the absence of some sort of support, and all are easily solved in the context of the mechanisms we propose here. We then describe a new version of the *ISIS* system, which is based on this abstraction. $ISIS_2$ provides a number of high level mechanisms that facilitate the use of process groups in application software design, including addressing support for atomic communication with the members of a single or several groups (even when their membership is changing), group RPC constructs, a package of distributed programming tools, a fault-tolerant asynchronous bulletin board mechanism, and resilient objects.

1. A tool kit for building distributed systems

Programming an application that requires the cooperation of a number of processes in a distributed system is a difficult and complex task. It would seem desirable to simplify this task by constructing a distributed systems "tool kit" to assist in solving the subproblems that arise most commonly. The processes in a distributed system need to synchronize their actions with respect to one another, but at the same time should attempt to make full use of the available concurrency. Processes, which may be on different machines, must also be able to obtain a consistent view of the actions of other processes. If the application is to be able to tolerate the failure of some of the processes, the task is further complicated by the need to detect failures and to code algorithms for dynamic reconfiguration after they occur. This often means that processes must agree upon the timing and relative order of failures. Thus, the desired tool kit would contain facilities for synchronization, achieving consistency, responding to failures in a simple and uniform manner, etc. With such a tool kit at hand, construction of higher level

[1]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. The views, opinions and findings contained in this report are those of the authors and should not be construed as an official Department of Defense position, policy, or decision.

software would be greatly simplified. This paper describes a software abstraction, the *virtually synchronous process group*, that makes it possible to build a tool kit having these characteristics. It then presents the algorithms the tool kit uses and the interface it supports.

Virtually synchronous process groups arose out of the efforts of our group to build a prototype of the *ISIS* system, $ISIS_1$ [Birman-b]. We started with an essentially ad-hoc system structure, but were eventually forced to abandon it in favor of this new approach, which provides a light-weight abstraction based on communication and addressing support for groups of processes that must cooperate in the presence of failures. We found that it was surprisingly easy to build an efficient, highly concurrent implementation of resilient objects on top of a layer supporting virtually synchronous process groups. It became possible to argue the correctness of our algorithms, which had not been easy earlier. Continued work with virtually synchronous process groups has now convinced us that they represent an extremely powerful tool for other types of distributed computing as well. This has led us to provide access to the process group abstraction at a lower level than in $ISIS_1$, resulting in a new version of the system, $ISIS_2$, which we discuss here.

Neither the notion of building systems from process groups nor that of providing an idealized communication abstraction is a new one [Cheriton-a] [Lamport-a] [Schneider-a]. What we have done in the *ISIS* project is essentially to unify these mechanisms and simultaneously to optimize their behavior in the situations that arise most commonly when failures can occur. The result is a system capable of satisfying even demanding practitioners that is at the same time rigorous in a formal sense.

2. Virtual synchrony in distributed systems

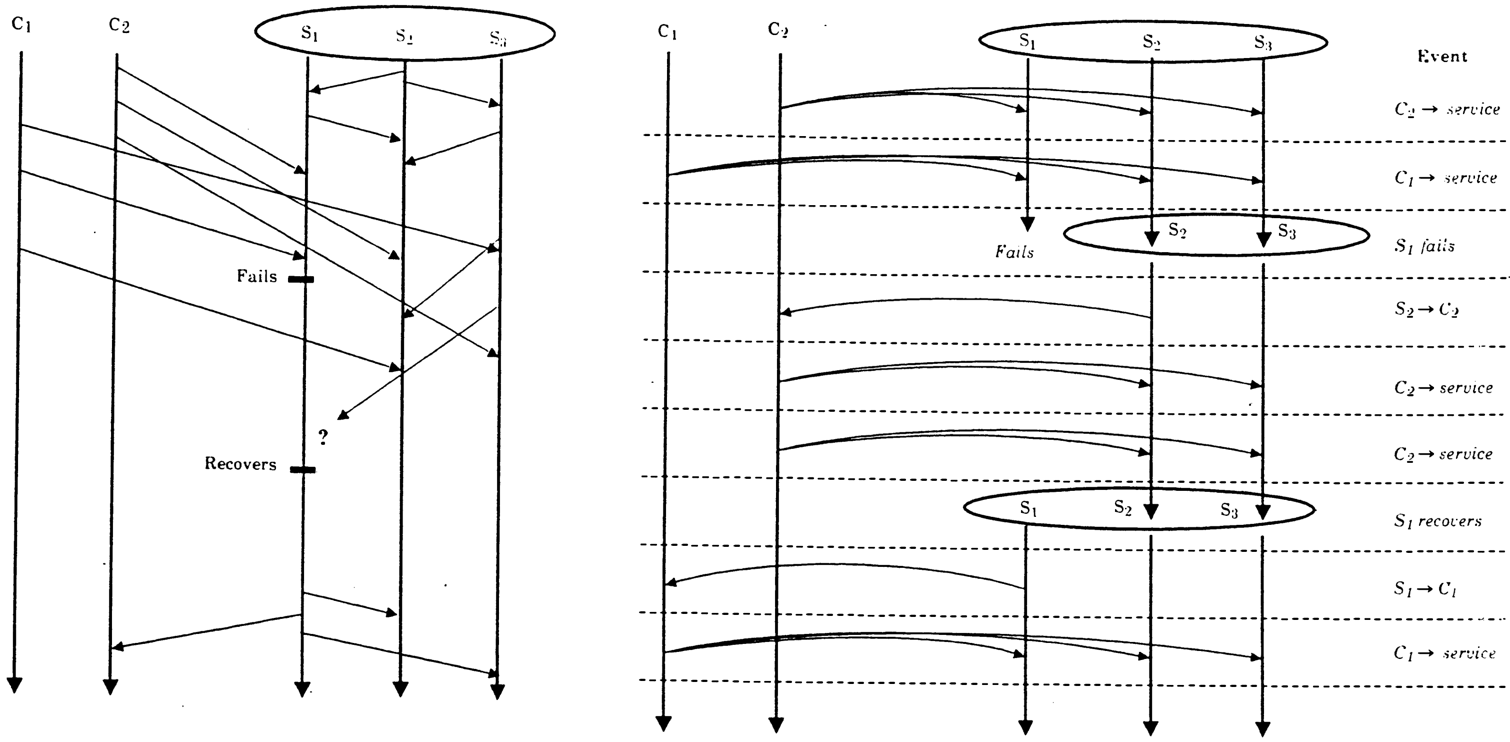
What makes the problems of synchronization, consistency and failure detection hard to tackle in a distributed setting is the asynchronous propagation of information among processes. In the absence of shared memory, the only way a process can learn of the behavior of other processes is through messages it receives. Likewise, the failure of a process is detected when a timeout occurs while waiting for a message from it. Since message transmission times vary from process to process, and change with the load on the system, messages relating to a single event may arrive at different processes at different times, and in different orders relative to other messages. This makes it difficult for a set of processes to maintain a consistent view of the actions of the set, or for them to coordinate their actions efficiently.

One way out of the problem would be to require that processes communicate only using broadcast protocols[2] that make message exchange synchronous and atomic [Chang] [Cristian] [Schneider-b]. Cooperating processes could then operate in lock step, exchanging messages (synchronously) with one another at the end of each step. Maintaining a consistent view of one another is then easy, as each process is always in the same point in its computation as any other. Synchronization is simple for the same reason. Process failures can be detected consistently by having the communication subsystem monitor the status of all processes and, if one fails, broadcast a message indicating that this has occurred. All operational processes will then learn of its failure simultaneously, in the next step.

Figure 1a illustrates a conventional space-time diagram of a distributed system. C_1 and C_2 are two client processes communicating with a distributed service implemented by three processes S_1, S_2 and S_3. The arrows pointing downward represent time, while the other arrows refer to messages being passed from one process to another. Figure 1b shows an environment where communication is by synchronous broadcasts. The simplicity in the second case is apparent, particularly when one realizes that the computation in Figure 1a represents at most a fragment of the one in Figure 1b.

Synchronous broadcasts are clearly too expensive to be of general applicability. Nonetheless, the example illustrates how trivial the issues of synchronization, consistency and failure detection become in an environment where there are strict constraints on the order in which messages are delivered to processes and in which there is a means of detecting process failures consistently. The approach is expensive because it requires *all* broadcasts to be ordered relative to one another, regardless of whether the application needs this to maintain consistency. This brings up a natural question: Is it possible to provide a family of broadcast primitives that provide well-defined but varying guarantees on the order in which messages are delivered? The implementation of each primitive could then be optimized to

[2]This is a broadcast to a set of processes, not to all the machines connected to a local network with hardware broadcast capabilities.



Figures 1a, 1b: Conventional and synchronous space-time diagrams for distributed systems

yield the best performance relative to the level of synchronization guaranteed by the primitive. A programmer could then choose the primitives appropriate to the application, making use of their ordering properties. The resulting program would be simpler and less error-prone, and because the issues of event ordering are isolated at one place (in the implementation of the broadcast primitives), optimizations that cannot be made at the application level can now be made on a system-wide basis.

This leads us to the concept of *virtual synchrony*. In an environment of virtual synchrony, the sequence of events (broadcasts and process failures) observed by each process is equivalent to one that could have occurred if the processes were operating in an environment where broadcasts and failure detection were synchronous. This is done by providing a family of broadcast primitives that guarantee a range of ordering properties on message delivery, and addressing support for *virtually synchronous process groups*[3] -- sets of processes whose membership may vary dynamically but which can be addressed as a unit. Our treatment of process failures is novel: A failure of one of the members of a virtually synchronous process group is made to appear as a broadcast to the operational members of the group, as if the last action taken by the failed process was to send a broadcast to the other processes informing them of its failure. This "failure broadcast" has well-defined ordering properties relative to other broadcasts. Thus the non-determinism associated with a process failure is masked from the application level, and a virtually synchronous process group can handle a process failure in much the same way as it would respond to any other event.

The implementation aspects of our process group approach have been presented in detail in [Birman-a]. The protocols we use assume that failures cause processes and sites to crash by halting (if a site fails, all the processes residing there fail too), that such a failure results in the loss of any volatile information stored in the failed process or the site[4], and that precautions must be taken to confirm the validity of non-volatile information that could be used after recovery. The communication subsystem can fail by losing messages or partitioning. It is also assumed that although clocks may be synchronized, the precision that results is low in comparison to typical inter-site message latencies. We believe that these assumptions are reasonable for a wide range of networks and are relatively independent of current technology.

2.1. The broadcast primitives

Below, we list our broadcast primitives. All of them guarantee that if a message broadcast using any of these primitives is received by one of the destination processes, it will eventually be received by all of them. If a destination process fails while a broadcast is in progress, we treat it as having received the message prior to failing. Since we assume that no volatile information stored by a process survives a failure, the scenario in which the failed process received the message before failing is a perfectly valid one from the point of view of the operational processes -- they will never obtain information inconsistent with this assumption. What we mean by guaranteed delivery, then, is that the failure of a

[3]Other systems that employ process groups, notably V [Cheriton-a], do not provide virtually synchronous behavior.

[4]We do not consider Byzantine failures, where a process may fail by taking maliciously incorrect actions.

destination process will never prevent *operational* destinations from receiving a message. It is possible, however, for the sending process to fail and for non of the operational destinations to receive the message. We treat this as if the sender had failed before initiating the broadcast. Again, unless information is written to non-volatile storage, this assumption will not be contradicted (a flush mechanism that permits safe access to non-volatile storage is given in [Birman-a]). For now, we assume that the set of destination processes is static and known at the time the broadcast is initiated, an assumption we relax in Sec. 2.2.

[1] *Group broadcast (GBCAST).* A broadcast made using the *GBCAST* primitive is ordered[5] 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 members of groups to which it belonged receive a simulated *GBCAST* from the failed process, informing them of its failure. We call such a *GBCAST* a *failure GBCAST*. A failure *GBCAST* has the further property that it is ordered after any other broadcast from the failed process.

[2] *Atomic broadcast (ABCAST).* This type of broadcast takes a label as a parameter and is ordered relative to *GBCAST*'s and to other *ABCAST*'s having the same label. If two *ABCAST*'s with the same label are initiated by the same process, the first one is ordered before the second.

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

The next two primitives respect *potential causality*. Following the ideas in [Lamport-b], we say that a broadcast b is *potentially causally before* a broadcast b' if b was initiated by a process before it initiated b', if b was delivered to a process before it initiated b', or if b and b' are related under the transitive closure of these two rules. Intuitively, if b is potentially causally before b', there may have been a flow of information from the point at which b was issued to the point when b' was delivered.

[4] *Causal broadcast (CBCAST).* This type of broadcast preserves potential causality. *CBCAST*'s are ordered relative to *GBCAST*'s and to other broadcasts that they are potentially causally related

[5] When we say that a broadcast a is ordered before (resp. after) a broadcast b, we mean that the message sent in broadcast a is delivered before (resp. after) the message sent in broadcast b at all overlapping destination processes. A broadcast a is said to be ordered relative to another broadcast b if a is ordered either before or after b.

to. Notice that *CBCAST* is similar to the message passing primitive used in the NIL system [Strom] and by Jefferson in his work on *virtual time* [Jefferson], but whereas these approaches optimistically deliver messages and then retract them when potential causality is subsequently found to have been violated, *CBCAST* always respects potential causality and never rolls back.

[5] *Causal atomic broadcast (CABCAST).* This primitive is like *ABCAST* in that all *CABCAST*'s are ordered relative to other *CABCAST*'s with the same label and relative to *GBCAST*'s. In addition, the order of potentially causal broadcasts is preserved.

The causal broadcast primitives have one further property which is a consequence of the causal ordering rule. If a process issues a chain of *CABCAST*'s and *CBCAST*'s asynchronously (by resuming execution while they are still in progress), then failures cannot leave a *gap* in the chain. For example, if a process asynchronously initiates broadcasts a and b in that order before it fails, and if b is delivered to some destination that stays operational, then a will be delivered to its destinations too, even if the broadcasts had no destinations in common. Thus, if b contains information that refers to a, no inconsistency will arise.

2.2. Addressing issues

Integrated with the primitives listed above is an *addressing mechanism* that binds *group identifiers* to lists of members (again, see [Birman-a] for details). Each process is treated as a singleton group whose identifier is formed from its process name. New group identifiers can be created at run time, and processes can join or leave these groups using facilities detailed in Section 4. A group identifier is very similar to a *capability*: it is unique and hard to forge, and can be stored, copied, or passed from process to process. Using the addressing mechanism, a process can broadcast to a single virtually synchronous process group or several groups at once, even though the membership of the groups may be changing dynamically. When a broadcast is sent to such a group while its membership is changing, the guarantee is that the broadcast will be delivered to all the processes that were in the group either before or after the change, and not to some intermediate or overlapping set of processes.

3. Applications of virtual synchrony

We now have the means at our disposal to construct some powerful software abstractions -- the tools cited in the introduction of this paper. This section reviews some of these abstractions and shows how virtual synchrony can be exploited to develop simple and efficient solutions for them. The next section focuses on the interface to the tool kit as well as some of the other high level facilities that $ISIS_2$ provides.

3.1. Group remote procedure calls

Using the broadcast primitives, a process can send messages to a single process group or a list of process groups, with ordering properties that will depend on the choice of broadcast primitive. By waiting for one or more responses, the semantics of an *ordered group RPC* can be obtained [Birrell] [Cooper]. A client calling one or more servers generates a *tag* and includes this in the message it sends to the servers. The recipients pack their reply into a new message, and broadcast[6] it back to the client, along with the tag. The RPC facility collects as many replies as the client requested, using the tag to match up requests with responses. By integrating the facility into the GBCAST failure handling mechanism, we can arrange that a waiting client will receive a failure GBCAST if a destination process fails (even if it is not a member of any process group to which the failed process belonged). It follows that a client is never left waiting if failures prevent it from receiving as many replies as it expected, and will never be faced with a situation where a delayed response arrives after the client concludes that some server process has failed.

Thus, a highly flexible RPC interface results: A client can wait for a single reply and then continue computing while other remote computation continues asynchronously (perhaps sending a subsequent message to interrupt such remote computation, as in the case of a parallel search). It can wait for a quorum of replies [Gifford] [Herlihy], perhaps implementing a data structure like the directories used in TABS [Daniels]. Or, it can wait for as many replies as are possible, either treating each reply

[6] It is often desirable to use broadcasts for point to point communication, to benefit from their ordering properties and the asynchronous delivery guarantees they provide.

individually or *collating* them as was done in CIRCUS [Cooper].

3.2. Global task decomposition and dynamic reconfiguration

In the *V* system, process groups are used to implement diverse distributed services, sometimes using non-identical components that divide the group data or task into parts, one for each group member [Cheriton-a]. Task assignment can then be varied dynamically, for example to accommodate process migration, failures, and recoveries, or to balance the workload on the system as a whole. *V* implements a probabilistically reliable broadcast protocol with which clients issue requests to such groups. It is our belief that as distributed systems get faster, even a small probability of unexpected behavior could become a major difficulty for programmers undertaking to build highly automated or critical software. In contrast, our broadcast primitives permit the construction of subsystems that are reliable *even when task assignment is varied dynamically.*

To support this, a global data structure representing task assignments would be maintained in each of the members of a virtually synchronous process group, and changes to it made using GBCAST. Additionally, all group membership changes would be transmitted using GBCAST. Because GBCAST is totally ordered relative to other communication events, all group members use the same task assignment and the same list of group members when a given request is received, and because failures and recoveries are also ordered in the same way at each member, all can react to such events in a consistent manner. For example, a name service could implement a decomposition rule whereby requests will be executed at the site where the requesting process resides if possible, using any deterministic rule to decide which member will respond to a request from some other site. This would tend to minimize the delay in responding to requests. Moreover, if a decision is made to transmit all requests to the service using ABCAST, a rule could be implemented whereby requests originating at sites where no member resides are handled by the the member "currently" responsible for the fewest requests. Because the ABCAST delivery ordering is globally fixed, all members can deduce the disposition of each request. Here, the added cost of transmitting requests using *ABCAST* might be justified by the better load sharing that results.

When a the task decomposition rule used by a group is changed dynamically, it is often necessary to transfer state from one process to another. For example, when a process joins a group, it may need to know the current state of certain data structures global to the group. If the state has a compact representation, this can be achieved by causing the RPC used to join the group to return the state, by having each current member *reply* with a data structure encoding this information. In a virtually synchronous sense, the new member joins and the state is transferred simultaneously. If the state is large, another approach can be used. A GBCAST is used to inform the group that the new process is joining the group and to appoint a process to coordinate the state transfer. The coordinating process uses one or more CBCAST's to transfer the state, and when the entire state has been transferred, it informs all the group members of this using a CBCAST. If the coordinator fails before completing the transfer, the other members receive a failure GBCAST instead of the completion broadcast, and the state transfer can be restarted by a new coordinator. Broadcasts made to the group during the state transfer will be delivered to the new process as well, because the initial GBCAST included it in the group. These messages are buffered until the transfer is completed.

The above solution also permits the construction of software in which processes "migrate" from site to site, as in V. Using our approach, migration is "instantaneous" from the perspective of external clients communicating with the group, in the sense that communication with the group will always take place before or after migration occurs. This eliminates any need for the client to be concerned with dynamic changes to the internal state of a service, as can occur in systems like *V* if equivalently strong guarantees are needed [Theimer].

3.3. Synchronization of concurrent computations

A virtually synchronous process group can easily implement distributed versions of such synchronization constructs as semaphores, monitors, and locks. A semaphore can be constructed as follows. To initiate a P() operation, ABCAST is used to transmit the request to all processes in the group (including the one requesting the P()). Each process grants the semaphore in the order in which P() requests are received. Since ABCAST's are delivered in the same order everywhere, all members grant the semaphore to the same process. When a process wishes to perform a V(), it simply uses MBCAST to inform all processes of this. (In [Birman-c] we give a simple, low-overhead deadlock detector for this setting).

A common use for semaphores is to obtain mutual exclusion on shared state variables. If the semaphore is replicated at the same processes where the shared variables are located, updates to the state variables and V() operations can both be implemented using asynchronous CBCAST's. Every process p will always observe updates by any process q that acquired the semaphore before it, because p's P() followed q's V(), which in turn followed q's updates. Since CBCAST respects causal orderings, the messages describing q's updates must have been delivered before the one describing q's V(), and hence before p's P(). Thus, instead of delaying computations each time a replicated variable is updated, they are delayed only when a P() is done, and updates still occur correctly.

When mutual exclusion on replicated data is obtained using read and write locks, it is common to use local read locks and to replicate only write locks. The problem with non-replicated read locks, though, is that they can be broken when failures occur [Bernstein]. In a virtually synchronous environment however, one can replicate a read lock by first acquiring it locally and then using an asynchronous CBCAST to inform remote lock managers that the read lock has been acquired. If a failure occurs, these CBCAST's are delivered before the failure GBCAST; hence the remote locks are acquired before the failure is acted upon. The behavior of a replicated read lock results, although the cost is essentially that of non-replicated one [Birman-b].

3.4. Coordinator-cohort computations

One method of obtaining fault-tolerance when using replicated objects is the coordinator-cohort approach: Each time a replicated object is invoked, one of the copies is designated as the coordinator for that invocation, while the other copies (the cohorts) act as passive backups that take over in case the coordinator fails. If the object processes several requests at once, the location of coordinators can be varied to share load, or minimize latency before a request is processed. If a replicated object is implemented as a virtually synchronous process group, the coordinator begins an execution by using

CBCAST to inform the cohorts of the invocation and to pass them the arguments for the invocation. It then executes the operation and uses another CBCAST to pass on the result of the execution to the cohorts, which then update their copies of the object. Should the coordinator fail, the cohorts will receive a failure GBCAST instead of the final CBCAST, and can chose a new coordinator to reexecute the operation. Any rule that depends on the global properties of the process group and the contents of the message can be used to pick the initial and subsequent coordinators; because GBCAST is ordered relative to all other broadcasts, all the members will be in the same state when a GBCAST is received and a new coordinator will be chosen unambiguously.

3.5. Maintaining serializability in replicated databases

By combining the above replicated data mechanisms, locking mechanisms, and coordinator-cohort mechanisms, highly asynchronous transactions on replicated data can be implemented [Joseph] [Birman-b]. In this approach the cost of a replicated write is close to that of a read: The critical path for doing a write is short because it only involves initiating an asynchronous CBCAST, which runs inexpensively in the background.

3.6. Recovery manager

One of the major problems that confronts a distributed systems designer involves automated recovery from failure. We show how basing a system on process groups makes it possible to construct a *recovery manager*. At the time a process joins a process group, it informs the recovery manager of the actions to be taken if it fails, and those to take subsequently when restart becomes possible. It also uses the recovery manager when it wishes to take checkpoints, and if a failure occurs, the recovery manager restarts processes either from the last checkpoint or by initiating a state transfer from process group member(s) that survived the failure.

Observe the relationship between the actions of the recovery manager and the notion of virtual synchrony. A checkpoint, for example, is logically a snapshot of the state of a system at some instant in time, and rollback to a checkpoint is logically an action that all processes in the system should undertake simultaneously [Chandy]. Recovery from failure has a similar flavor: we would like to treat the system as if failures occurred one by one in some fixed order, reconstructible after the fact by the recovery manager. It can then determine which members of a process group failed last, and if these are restartable, execute them with arguments indicating that they should recover from their last checkpoints. Other processes would then be restarted, but with arguments causing them to join the operational members of process group (by state transfer from them). Recall that the GBCAST primitive is totally ordered relative to all communication events. Thus, if an action is taken immediately upon reception of a GBCAST message requesting it, the system state that results is logically equivalent to one in which that action was taken *simultaneously* by all the recipients of the message (provided that the action itself is not sensitive to real time), which is exactly what is required for checkpoint and rollback. Notice that although *GBCAST* is a costly protocol, we are using it here for infrequent events.

For example, to initiate a checkpoint, the recovery manager need only use GBCAST to send a checkpoint-request message to all the participants. On receiving this message, these write down their current state, numbering checkpoints in increasing order (old ones can be discarded). Subsequently, all participants can be asked to roll back to a checkpoint by sending another GBCAST, identifying the checkpoint number to use. When the participants roll back, their states will be mutually consistent. Participants that lack this checkpoint were not operational when it was made and should drop out of their respective groups; participants that were operational then, but are not now, are treated as having failed immediately after the checkpoint was made. In the absence of support for process groups, a more complicated algorithm like the one given in [Koo] must be used to ensure system correctness.

Similarly, virtual synchrony can be exploited to determine the sequence of site and process failures that occurred while a site was down. Each time the membership of a process group is changed, the members log the new list of process group members (*view*) on non-volatile storage. Since GBCAST is used to propagate membership changes, all will record the same sequence of views, but a failed process may have only a prefix of the full sequence. Thus, if all process have failed, a recovering process learns which were the last to fail by querying the other processes that were up in its last recorded view. The computation then follows the same scheme described in [Skeen], successively reducing the set of candidates until it obtains a set of processes whose last views list the others as operational. The recovery manager simply generalizes this by storing views in a single shared data structure and running this algorithm on behalf of many processes at one time.

4. The $ISIS_2$ system

The preceding sections have proposed a powerful approach to distributed computing and illustrated its application to some of the standard problems one faces in this setting. A dual problem is to *package* this approach into a facility that is elegant, easy to use, and hides correctness issues from the programmers who use it. $ISIS_2$ does this by providing several major subgroups of mechanisms: an implementation of the broadcast primitives and addressing mechanisms, a collection of tools based on algorithms like the ones in the Section 3, and two higher level interfaces supporting fault-tolerant *bulletin boards* and *resilient objects*.

4.1. Virtually synchronous process group interface

Table 1 lists the primitives used to create and manipulate process groups in $ISIS_2$. Processes and process groups are addressed by means of capabilities. The broadcast primitives take a *capability list* as an argument and translate this into the set of destinations to which the broadcast should be sent. The interface used to invoke a broadcast primitive (from the *C* programming language) is:

nresp := xbcast (dests, msg, nwanted, answ, alen);

Here, *xbcast* is one of *gbcast*, *abcast*, etc., *dests* is a list of capabilities annotated with the entry number to invoke in the processes to which each capability corresponds, *nwanted* is the number of responses desired, and *answ* is an array in which to store responses of length *alen* bytes each. A recipient replies by invoking a *reply* primitive:

reply (caller, msg, extra_dests, answ, alen);

Here, *extra_dests* lists extra destinations, if any, to which a copy of the reply should be sent. By default, replies are transmitted using *CBCAST*, although versions that use *CABCAST* and *MBCAST* are feasible.

cap := pg_create (p_name)	Create a process group containing the single process identified by *p_name*. A capability on the group is returned.
pg_join (cap, p_name)	Add process *p_name* to the process group identified by *cap*.
pg_leave (cap, p_name)	Delete process *p_name* from the process group identified by *cap*. Also occurs automatically if a member of a process group fails.
pg_migrate (cap, old, new)	Simultaneously add process *new* and delete process *old* from the process group.
pg_delete (cap)	Delete the process group corresponding to *cap*.
pg_members (cap)	Returns the "current" membership of group *cap*.
pg_kill (clist, signo)	Sends UNIX signal *signo* to the processes and groups in *clist*.

Table 1: Process group manipulation primitives

4.2. The distributed systems tool kit

Table 2 identifies a the tools provided in our first tool kit implementation. The tools implement algorithms like the ones described in the previous section and have procedural interfaces. For example, the recovery manager is divided into several subtools. One permits a process to register desired restart actions and cleanup actions in a distributed recovery database; should a failure occur, the desired action is automatically performed. Another is used to initiate a checkpoint: given a set of capabilities on the subsystems to be checkpointed, it invokes a checkpoint routine in each member process at an appropriate (logical) time. Taken as a set, the distributed systems tool kit represents a packaging of the lower level algorithms into a form that even naive application programmers will be able to cope with. Moreover, the most common actions (updates to replicated data and group RPC) rely on our cheapest protocol (asynchronous CBCAST), ensuring that the average performance of applications that use the tool kit routines will be good.

4.3. Bulletin boards

In [Birman-c], we describe a shared memory mechanism, modeled after Cheriton's problem oriented shared memory [Cheriton-b], but with rigorous consistency properties. This facility, which we

Tool	Description
Synchronization	*Distributed synchronization facility.*
Semaphores	Replicated binary and general semaphores.
Monitors	Monitor lock, condition variables and signals.
Deadlock detection	Mechanism for identifying synchronization deadlocks.
Replication	*Replicated data and state management.*
Replicated update	Asynchronous updates to replicated information.
Reconfiguration	*Assists a process group in maintaining global property information.*
Config. manager	Supports queries and updates to configuration control information.
Migration tool	Oversees migration of an activity from process to process (or site to site).
State transfer	Transfers data to a recovering or migrating process.
Failure detection	*Facility for detecting failures*
Process watcher	Monitor a process, trigger exception handler if it fails.
Recovery manager	Supervises actions after failure and upon recovery.
Checkpointing tool	Supervises checkpoint creation and rollback.
Recovery tool	Restarts a process and advises it whether to *join* or *rollback*.
Recovery blocks	Triggers cleanup actions by *other* processes if the caller fails.
Stable storage	*Disk-based stable storage, for managing logs, checkpoints, etc.*
Transactions	*Tools for building distributed transactions*
TID manager	Generates, compares transaction id's.
Version storage	Caches provisional copies of data items.
Locking	A distributed transactional locking facility.
Commit tool	Manages *commit lists*, propagates commit and abort decisions.

Table 2: The distributed computing tool kit

call the bulletin board (bboard) tool, can be used as a higher level asynchronous communication interface in programs that interact by having some processes post information that other processes read later, at their leisure. A well known instance of this paradigm is in artificial intelligence systems, where expert subsystems cooperate by interacting through a blackboard. Recent work in distributed systems like Linda [Carriero] and V suggests that this approach can also be valuable in distributed settings.

Our approach allows each process to subscribe to one or more bboards (Figure 2). A bboard contains some set of abstract data items, maintained by routines that the bboard designer supplies. The bboard functions as a *scheduler*, accepting invocations of operation from its clients and sometime later delivering them in the order they should be executed. This leaves the flexibility for clients to interpret

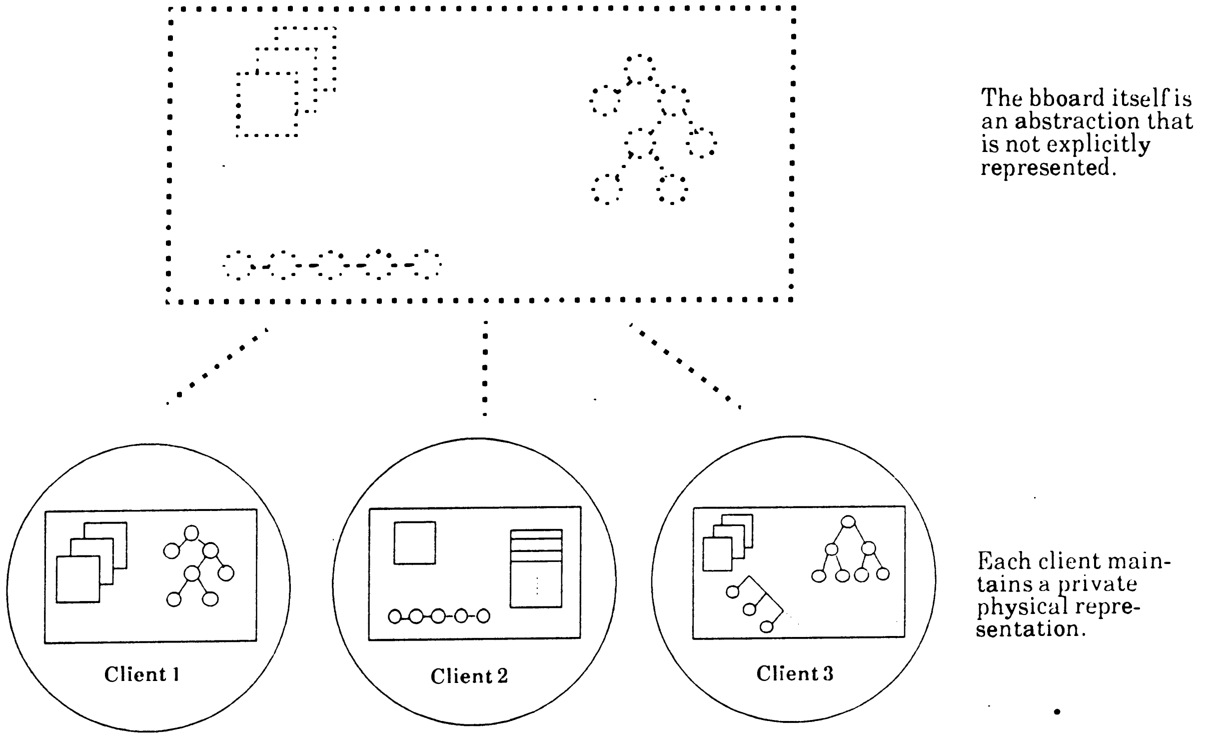


Figure 2: Clients interacting through a bboard

operations in differing ways, as illustrated in the figure.

The bulletin board gives each invocation a *request name* that can be used in guards. Thus, a process wishing to post a memo only after some pending request *rname* has completed could perform:

{ rname } post(memo);

Guards also support a pattern matching construct: [*pname*,*op*], which matches any invocation of *op* by process *pname*. Either *pname* or *op* can be Φ, matching any value, and the *op* can be *TERM* to match process termination. For example, the guard:

{ after(rname)[Φ, delete] **or** [q, TERM] } post(memo);

delays the *post* until some process does a *delete* subsequent to the execution of request *rname*, or *q* terminates. A process can refer to its own termination, as in:

async { [p,TERM] } cleanup();

which process p might use to schedule an operation to cleanup after itself.

Our approach addresses several aspects of bboard correctness. One is *consistency*: which involves ensuring that the results the bboard returns to different clients are in agreement, and can be seen as a question of enforcing orderings on the execution of operations. For example, a *causally* consistent bboard guarantees that potentially causally related operations will be executed in the same order everywhere, while *total* consistency provides a globally fixed execution ordering. Other forms of consistency are discussed in [Birman-c]. A second aspect is *synchronization,* which we address by allowing an operation to be *guarded* by an expression describing events that must be completed before the operation is scheduled. And, a third aspect is *fault tolerance,* which is handled by integrating a failure detection mechanism into the guards.

We do not see bboards as a single mechanism suited to all possible applications, but they seem to be well suited to a large class of applications. For example, a designer of a chess application might use a causally consistent bboard for interactions between a display expert, perhaps coded in *C* to take advantage of the various window packages, a move generator coded in LISP, and a move evaluator coded in PROLOG. Causal consistency ensures that if a posted data item is referenced in some subsequently posted item, a process reading the latter will also observe the former. Similarly, primitives such as the ones from the S/Net's Linda Kernel could easily be implemented using a totally consistent bboard -- here, the total ordering is needed because the Linda tuple operations conflict with one another. A number of interesting subsystems can be built by combining multiple bboards having different levels of consistency. We see bboards as the sort of tool that a naive programmer might use without learning more about distributed computing than is required to deal with the bboard interface.

4.4. Resilient objects

The resilient object support from $ISIS_1$ is being ported to $ISIS_2$ [Birman-b]. This subsystem translates fault-intolerant, non-distributed object specifications into process groups that implement those specifications so as to mimic a single fault-tolerant instance of the desired type of object. Clients interact with the object using group RPC's (Figure 3), possibly bundling multiple requests into a

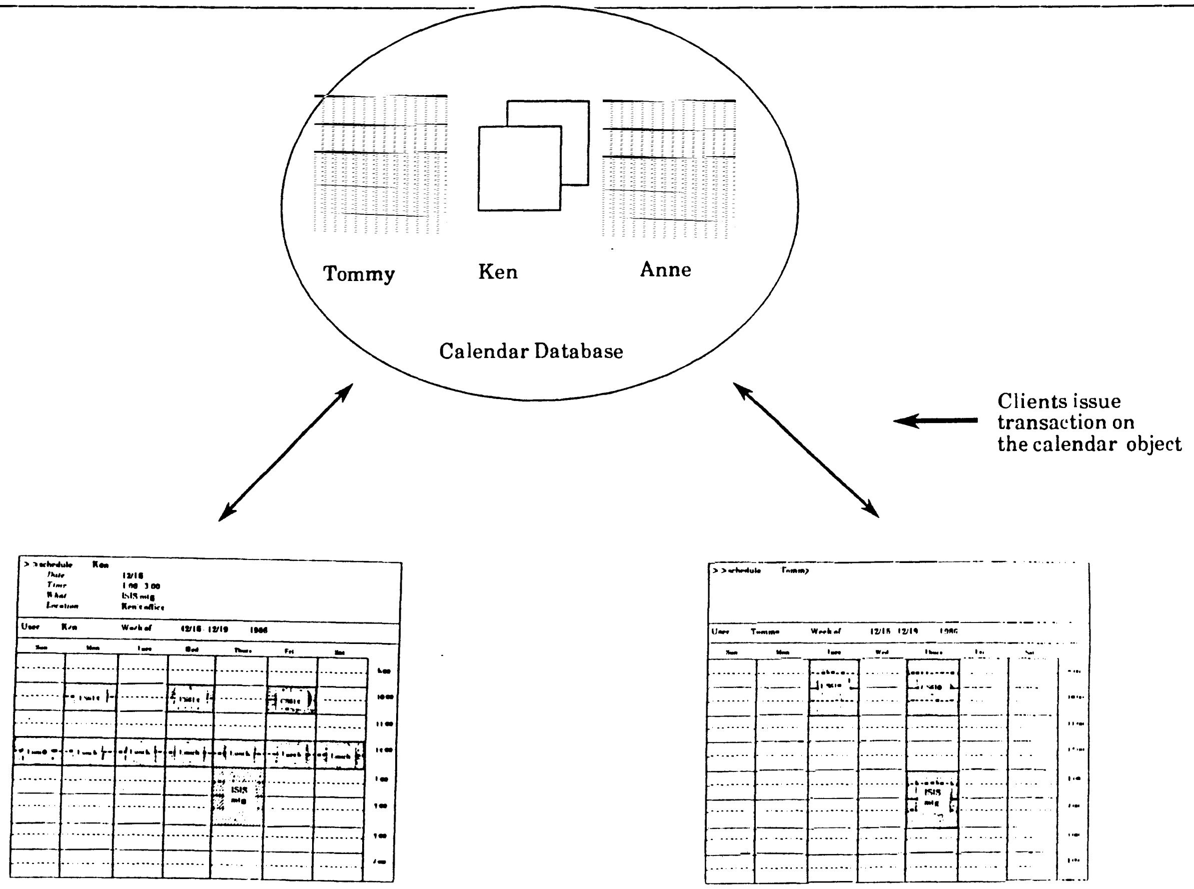


Figure 3: Clients interacting with a resilient calendar database

transaction by first issuing a BEGIN() and then terminating the sequence with a COMMIT() or ABORT(). Thus, if a calendar object supporting retrieve and insert operations has been designed, a program might execute the sequence of requests:

```
BEGIN();
        CALOBJ$insert("ken", "3/1/86", "2pm", "Meet Tommy in Upson 315");
        CALOBJ$insert("tommy", "3/1/86", "2pm", "Meet Ken in Upson 315");
COMMIT();
```

to atomically insert entries into Ken's and Tommy's calendars (Figure 3).

Resilient objects differ from bboards in several ways. First, a resilient object persists independently from its clients. Execution of requests is transactional, and since one object can issues calls to another, these are nested transactions [Moss]. Mechanisms for lock-based concurrency control, dynamic storage allocation, top-level actions, and cobegin statements are provided. Finally, a roll-forward failure recovery mechanism has been implemented.

The performance of our initial resilient object implementation was good, and is likely to improve in our new system. However, the resilient object approach is strongly oriented towards a database style of object, and the overhead associated with transactional execution is not negligible. Thus, an application that can be implemented using the tool kit or a bboard will generally achieve higher performance than if it was implemented using a resilient object.

5. Status and performance

At the time this paper is being written (Dec. 1986), our protocols have been implemented and most of the code has been debugged as a user-mode process under UNIX. We plan to move this code into the UNIX kernel to benefit from reduced context switching and scheduling overhead. At the present time, we are unable to make precise performance statements, except to say that performance will exceed that of the protocols in $ISIS_1$. *Resilient objects can easily be ported to run under the new system, and the tool kit will be easy to construct, as most routines involve just a few calls to the protocols and the more complex ones, like the version store, exist as part of* $ISIS_1$. *The bboard implementation will be simple because the algorithms are largely straightforward /Birman-b/. Thus, we are confident that all software described in this paper will be operational by the end of summer in 1987 and that by mid-May we will be able to replace this paragraph by a section giving performance figures for the protocols.*

6. Conclusions and future directions

We have described the design rationale and features of a new system based on the notion that virtually synchronous communication is the most straightforward methodology for building distributed software. This approach enables us to insulate users from the details of message based interprocess communication, but at the same time to provide programming tools that will perform well and have precisely specified behavior, even in the presence of failures. The distributed programming methodology that results is surprisingly simple but at the same time rigorous. Moreover, the overhead associated with our approach is paid primarily when a site fails or process group membership changes -- both relatively infrequent events. Thus, substantial performance benefits and design simplicity are achieved at low actual cost.

Many questions remain open. The most obvious is that we do not know how to infer the choice of protocol from context, or the level of consistency needed for a given bboard application. In practice, however, this does not seem difficult. We have largely overlooked real time issues, although these are important in many settings where fault-tolerance is needed. One possibility may be to place loose time constraints on message delivery and signal a "timing fault" when those constraints cannot be satisfied. Extremely demanding real time scheduling constraints, however, are probably incompatible with the current $ISIS_2$ design philosophy. Our handling of network partitioning is also weak, since the protocols on which this work is based tend to block rather than risk incorrect actions when partitioning occurs.

We are now convinced that the process group approach represents a conceptual breakthrough. Having tried to build robust distributed software using other methodologies and failed, we have now succeeded using this approach. The experience has turned us into believers in the sort of virtual correctness the process group provides. As this technology becomes widely available and the remaining limitations are overcome, it will enable relatively unsophisticated programmers to undertake a completely new kind of programming, perhaps contributing to the explosion of distributed computing that has been predicted so often, but up until now has failed to materialize. Moreover, it provides an intriguing glimpse into the type of services a genuinely integrated distributed operating system might provide. It seems reasonable to expect that such systems will eventually become common, and that they will will fundamentally change the way we formulate and solve distributed computing problems.

7. Acknowledgements

The work reported here draws on work done in collaboration with many others. Dale Skeen was a founder of the *ISIS* project, and the *ABCAST* and failure detection protocols arose from joint work with him. Frank Schmuck and Pat Stephenson were also co-authors of the paper on bulletin boards that was cited in Sections 3.6.3 and 4.3. Amr El-Abbadi, Wally Deitrich, and Thomas Raeuchle were all involved in the work on resilient objects reported in Section 4.4. We are also grateful to Ozalp Babaoglu, Fred Schneider, Sam Toueg, and John Warne for their many insightful comments and suggestions.

8. References

[Bernstein] Bernstein, P. and Goodman, N. The failure and recovery problem for replicated databases. *Proc. 2nd ACM SIGACT/SIGOPS Conference on the Principles of Distributed Systems*, Montreal, Canada (Aug. 1983), 114-122.

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

[Birman-c] Birman, K. and Joseph, T. Programming with shared bulletin boards in asynchronous distributed systems. Dept. of Computer Science TR-86-772, Cornell University (August 1986; Revised December 1986).

[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-a] Cheriton, D. and Zwaenepoel, W. Distributed process groups in the V kernel. *ACM Transactions on Computer Systems* **3**, 2 (May. 1985), 77-107.

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

[Cooper] Cooper, E. Replicated distributed programs. *Proc. 10th ACM SIGOPS Symposium on Operating Systems Principles.* Orcas Island, Washington, Dec. 1985, 63-78.

[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.

[Daniels] Daniels, D, Spector, A. An algorithm for replicated directories. *Proc. 2nd ACM SIGACT/SIGOPS Conference on the Principles of Distributed Systems*, Montreal, Canada (Aug. 1983), 104-113.

[Gifford] Gifford, D. Weighted voting for replicated data. *Proc. 7th ACM SIGOPS Symposium on Operating Systems Principles.* December 1979.

[Herlihy] Herlihy, M. Replication methods for abstract data types. *Ph.D. thesis*, Dept. of Computer Science, MIT (LCS 84-319), May 1984.

[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. TR 85-706, Dept. of Computer Science, Cornell University (Oct. 1985).

[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.

[Moss] Moss, E. Nested transactions: An approach to reliable, distributed computing. Ph.D. thesis, MIT Dept of EECS, TR 260, April 1981.

[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.

[Theimer] Theimer, M., Lantz, K., Cheriton, D. Preemtable remote execution facilities for the V-System. *Proc. 10th ACM SIGOPS Symposium on Operating Systems Principles.* Orcas Island, Washington, Dec. 1985, 2-12.