Communication Support for Reliable Distributed Computing*

Kenneth P. Birman
Thomas A. Joseph

TR 86-753
May 1986

Department of Computer Science
Cornell University
Ithaca, NY 14853

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

Distributed Computing, Springer Verlag,

COMMUNICATION SUPPORT FOR RELIABLE DISTRIBUTED COMPUTING*

Kenneth P. Birman and Thomas A. Joseph

Department of Computer Science
Cornell University, Ithaca, New York

ABSTRACT

We describe a collection of communication primitives integrated with a mechanism for handling process failure and recovery. These primitives facilitate the implementation of fault-tolerant process groups, which can be used to provide distributed services in an environment subject to non-malicious crash failures.

1. Introduction

At Cornell, we recently completed a prototype of the *ISIS* system, which transforms abstract type specifications into fault-tolerant distributed implementations, while insulating users from the mechanisms by which fault-tolerance is achieved [Birman-a]. A wide range of reliable communication primitives have been proposed in the literature, and we became convinced that by using such primitives when building the *ISIS* system, complexity could be avoided. Unfortunately, the existing protocols, which range from reliable and atomic broadcast [Chang] [Cristian] [Schneider] to Byzantine agreement [Strong], either do not satisfy the ordering constraints required for many fault-tolerant applications or satisfy a stronger constraint than necessary at too high a cost. In particular, these protocols have not attempted to minimize the latency (delay) incurred before message delivery can occur. In *ISIS*, latency appears to be a major factor that limits performance. Fault-tolerant distributed systems also need a way to detect failures and recoveries consistently, and we found that this could be integrated into the communication layer in a manner that reduces the synchronization burden on higher level algorithms. These observations motivated the development of a new collection of primitives, which we present below.

*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.

Our broadcast primitives are designed to respect several sorts of ordering constraints, and have cost and latency that varies depending on the nature of the constraint required [Birman-b] [Joseph-a] [Joseph-b]. Failure and recovery are integrated into the communication subsystem by treating these events as a special sort of broadcast issued on behalf of a process that has failed or recovered. The primitives are presented in the context of *fault tolerant process groups:* groups of processes that cooperate to implement some distributed algorithm or service, and which need to see consistent orderings of system events in order to achieve mutually consistent behavior. Our primitives provide flexible, inexpensive support for process groups of this sort. By using these primitives, the *ISIS* system achieved both high levels of concurrency and suprisingly good performance. Equally important, its structure was made suprisingly simple, making it feasible to reason about the correctness of our algorithms.

In the remainder of this paper we sumarize the issues and alternatives that the designer of a distributed system is presented with, focusing on two styles of support for fault-tolerant computing: remote procedure calls coupled with a transactional execution facility, and the fault-tolerant process group mechanism mentioned above. Next, our primitives are described. We conclude by speculating on future directions in which this work might be taken.

2. Goals and assumptions

The difficulty of constructing fault-tolerant distributed software can be traced to a number of interrelated issues. The list that follows is not exhaustive, but attempts to touch on the principal considerations that must be addressed in any such system:

1. *Synchronization.* Distributed systems offer the potential for large amounts of concurrency, and it is usually desirable to operate at as high a level of concurrency as possible. However, when we move from a sequential execution environment to a concurrent one, it becomes necessary to synchronize actions that may conflict in their access to shared data or entail communication with overlapping sets of processes. Additional problems that can arise in this context include deadlock avoidance or detection, livelock avoidance, etc.

2. *Fault detection*. It is usually necessary for a fault-tolerant application to have a consistent picture of which components fail, and in what order. Timeout, the most common mechanism for detecting failure, is unsatisfactory, because there are many situations in which a healthy component can timeout with respect to one component without this being detected by some another. Failure detection under more rigorous requirements requires an agreement protocol that is related to Byzantine agreement [Strong] [Hadzilacos].

3. *Consistency*. When a group of processes cooperate in a distributed system, it is necessary to ensure that the operational processes have consistent views of the state of the group as a whole. For example, if process p believes that some property P holds, and on the basis of this interacts with process q, the state of q should not contradict the fact that p believes P to be true. This problem is closely related to notions of knowledge and consistency in distributed systems [Halpern] [Lamport]. In our context, P will often be the assertion that a broadcast has been received by q, or that q saw some sequence of events occur in the same order as did p.

4. *Serializability*. Many distributed systems are partitioned into data manager processes, which implemented shared variables, and transaction manager processes, which issue series of requests to data managers [Bernstein]. If transaction managers can execute concurrently, it is often desirable to ensure that transactions produce serializable outcomes [Eswaren] [Papadimitrou]. Serializability is increasingly viewed as an important property in "object-oriented" distributed systems that package services as abstract objects with which clients communicate by remote procedure calls (RPC). On the other hand, there are systems for which serializability is either too strong a constraint, or simply inappropriate.

Jointly, these problems render the design of fault-tolerant distributed software daunting. The correctness of any proposed design and of its implementation become serious, if not insurmountable, concerns. We faced this range of problems in our work on the *ISIS* system, and rapidly became convinced that in the absence of some systematic never be constructed. In Sec. 6, we will show how the primitives of Sec. 5 provide such an

The failure model that one adopts has considerable impact on the structure of the resulting system. We adopted the model of fail-stop processors [Schneider]: when failures occur, a processor simply stops (crashes), as do all the processes executing on it. We rejected the extremely pessimistic assumptions of the malicious Byzantine failure models because they lead to slower, more redundant software, and because the probability that a system failure will be undetectably malicious seems vanishingly small in practice. Work based on Byzantine assumptions is described in [Lamport] and [Schlicting]. We also assume that the communication network is reliable but subject to unbounded delay. Although network partitioning is an important problem, we do not address it here.

Further assumptions are sometimes made about the availability of synchronized realtime clocks. Here, we adopt the position that although reasonably accurate elapsed-time clocks are normally available, closely synchronized clocks frequently are not. For example, the 60Hz "line" clocks commonly used on current workstations are only accurate to 16ms. On the other hand, 4-8ms inter-site message transit times are common and 1-2ms are reported increasingly often. Thus, it is impossible to synchronize clocks to better than 32-48ms, enough time for a pair of sites to exchange between 4 and 50 messages. Thus, we assume that clock skew is "large" compared to inter-site message latency.

3. Alternatives

Two different approaches to reliable distributed computing have become predominant. The first approach involves the provision of a communication primitive, such as *atomic broadcast*, which can be used as the framework on which higher level algorithms are designed. Such a primitive seeks to deliver messages reliably to some set of destinations, despite the possibility that failures might occur during the execution of the protocol. We term this the *process group* approach, since it lends itself to the organization of cooperating processes into groups, as described in the introduction. Process groups are an extremely flexible abstraction, and have been employed in the V Kernel [Cheriton] as well in the *ISIS* system. The idea of using process groups to address the problems raised in the previous section seems to be new.

A higher level approach is to provide mechanisms for *transactional* interactions between processes that communicate using remote procedure calls [Birrell]. This has lead to work on nested transactions (due to nested RPC's) [Moss], support for transactions at the language level [Liskov], transactions within an operating systems kernel [Spector] [Allchin] [Popek] [Lazowska], and transactional access to higher-level replicated services, such as resilient objects in *ISIS* or relations in database systems. The primitives in a transactional system provide mechanisms for distributing the request that initiates the transaction, accessing data (which may be replicated), performing concurrency control, and implementing commit or abort. Additional mechanisms are normally needed for orphan termination, deadlock detection, etc. The issue then arises of how these mechanisms should themselves be implemented. Our work in *ISIS* leads us to believe that transactions are easily implemented on top of fault-tolerant process groups; lacking such a mechanism a number of complicated protocols are needed and the associated system support can be substantial. Moreover, transactions represent a relatively heavy-weight solution to the problems surveyed in the previous section. We now believe that transactions are inappropriate for casual interactions between processes in typical distributed systems. The remainder of this paper is therefore focused on the process group approach.

4. Existing broadcast primitives

The considerations outlined above motivated us to examine reliable broadcast primitives. Previous work has been reported on this problem, under assumptions comparable with those of Sec. 2, and we begin by surveying this research. In [Schneider], an implementation of a *reliable broadcast* primitive is described. Such a primitive ensures that a designated message will be transmitted from one site to all other operational sites in a system; if a failure occurs but any site has received the message, all will eventually do so. [Chang] and [Cristian] describe implementations for *atomic broadcast*, which is a reliable broadcast with the additional property that messages are delivered in the same order at all overlapping destinations, and this order preserves the transmission order if messages originate in a single site.

Atomic broadcast is a powerful abstraction, and essentially the same behavior is provided by one of the primitives we discuss in the next section. However, it has several drawbacks which made us hesitant to adopt it as the *only* primitive in the system. Most serious is the latency that is incurred in order to satisfy the delivery ordering property. Without delving into the implementations, which are based on a token scheme in [Chang] and an acknowledgement protocol in [Schneider], we observe that the delaying of certain messages is fundamental to the establishment of a unique global delivery ordering; indeed, it is easy to prove that this must always be the case. In [Chang] a primary goal is to minimize the number of messages sent, and the protocol given performs extremely well in this regard. However, a delay occurs while waiting for tokens to arrive and the delivery latency that results may be high. [Cristian] assumes that clocks are closely synchronized and that message transit times are bounded by well-known constants, and uses this to derive atomic broadcast protocols tolerant of increasingly severe classes of failures. The protocols explicitly delay delivery to achieve the desired global ordering on broadcasts. Hence for poorly synchronized clocks (which are typical of existing workstations), latency would be high in comparison to inter-site message transit times.

Another drawback of the atomic broadcast protocols is that no mechanism is provided for ensuring that all processes observe the same sequence of failures and recoveries, or for ensuring that failures and recoveries are ordered relative to ongoing broadcasts. We decided to look more closely at these issues.

5. Our broadcast primitives

We now describe three broadcast protocols - *GBCAST*, *BCAST*, and *OBCAST* - for transmitting a message reliably from a sender process to some set of destination processes. Details of the protocols and their correctness proofs can be found in [Birman-b]. The protocols ensure "all or nothing" behavior: if any destination receives a message, then unless it fails, all destinations will receive it.

5.1. The GBCAST primitive

GBCAST (group broadcast) is the most constrained, and costly, of the three primitives. It is used to transmit information about failures and recoveries to members of a process group. A recovering member uses *GBCAST* to inform the operational ones that it has become available. Additionally, when a member fails, the system arranges for a *GBCAST* to be issued to group members on its behalf, informing them of its failure. Arguments to *GBCAST* are a message and a process group identifier, which is translated into a set of destinations as described below (Sec. 5.6).

Our *GBCAST* protocol ensures that if any process receives a broadcast B before receiving a *GBCAST* G, then all overlapping destinations will receive B before G. This is true regardless of the type of broadcast involved. Moreover, when a failure occurs, the corresponding *GBCAST* message is delivered after any other broadcasts from the failed process. Each member can therefore maintain a *view* listing the membership of the process group, updating it when a *GBCAST* is received. Although views are not updated simultaneously in real time, all members observe the same sequence of view changes. Since, *GBCAST*'s are ordered relative to all other broadcasts, all members receiving a given broadcast will have the same value of *view* when they receive it.[1] Members of a process group can use this value to pick a strategy for processing an incoming request, or to react to failure or recovery without having to run any special protocol first. Since the *GBCAST* ordering is the same everywhere, their actions will all be consistent. Notice that when all the members of a process group may have failed, *GBCAST* also provides an inexpensive way to determine the last site that failed: process group members simply log each new view that becomes defined on stable storage before using it; a simplified version of the algorithm in [Skeen-

[1]A problem arises if a process p fails without receiving some message after that message has already been delivered to some other process q: q's view when it received the message would show p to be operational; hence, q will assume that p received the message, although p is physically incapable of doing so. However, the state of the system is now equivalent to one in which p did receive the message, but failed before acting on it. In effect, there exists an interpretation of the actual system state that is consistent with q's as-

a] can then be executed when recovering from failure.

5.2. The BCAST primitive

The *GBCAST* primitive is too costly to be used for general communication between process group members. This motivates the introduction of weaker (less ordered) primitives, which might be used in situations where a total order on broadcast messages is not necessary. Our second primitive, *BCAST*, satisfies such a weaker constraint. Specifically, it is often desired that if two broadcasts are received in some order at a common destination site, they be received in that order at all other common destinations, even if this order was not predetermined. For example, if a process group is being used to maintain a replicated queue and *BCAST* is used to transmit queue operations to all copies, the operations will be done in the same order everywhere, hence the copies of the queue will remain mutually consistent. The primitive *BCAST*(*msg*, *label*, *dests*), where *msg* is the message and *label* is a string of characters, provides this behavior. Two *BCAST*'s having the same label are delivered in the same order at all common destinations. On the other hand, *BCAST*'s with different labels can be delivered in arbitrary order, and since *BCAST* is not used to propagate information about failures, no flushing mechanism is needed. The relaxed synchronization results in lower latency.

5.3. The OBCAST primitive

Our third primitive, *OBCAST* (ordered broadcast), is weakest in the sense that the it involves less distributed synchronization then *GBCAST* or *BCAST*. *OBCAST*(*msg*, *dests*) atomically delivers *msg* to each operational *dest*. If an *OBCAST potentially causally dependent* on another, then the former is delivered after the latter at all overlapping destinations. A broadcast B_2 is potentially causally dependent on a broadcast B_1 if both broadcasts originate from the same process, and B_2 is sent after B_1, or if there exists a chain of message transmissions and receptions or local events by which knowledge could have been transferred from the process that issued B_1 to the process that

---

sumption.

issued B_2 [Lamport]. For causally independent broadcasts, the deliver ordering is not constrained.

OBCAST is valuable in systems like *ISIS*, where concurrency control algorithms are used to synchronize concurrent computations. In these systems, if two processes communicate concurrently with the same process the messages are almost always independent ones that can be processed in any order: otherwise, concurrency control would have caused one to pause until the other was finished. On the other hand, order is clearly important within a causally linked series of broadcasts, and it is precisely this sort of order that *OBCAST* respects.

5.4. Other broadcast primitives

A weaker broadcast primitive is reliable broadcast, which provides all-or-nothing delivery, but no ordering properties. The formulation of *OBCAST* in [Birman-b] actually includes a mechanism for performing broadcasts of this sort, hence no special primitive is needed for the purpose. Additionally, there may be situations in which *BCAST* protocols that also satisfy an *OBCAST* ordering property would be valuable. Although our *BCAST* primitive could be changed to respect such a rule, when we considered the likely uses of the primitives it seemed that *BCAST* was better left completely orthogonal to *OBCAST*. In situations needing hybrid ordering behavior, the protocols of [Birman-b] could easily be modified to implement *BCAST* in terms of *OBCAST*, and the resulting protocol would behave as desired.

5.5. Synchronous versus asynchronous broadcast abstractions

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

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

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

A second problem arises when the user-level implications of this atomicity rule are considered. In the event of a failure, any suffix of a sequence of aysnchronous broadcasts could be lost and the system state would still be internally consistent. A process that is about to take some action that may leave an externally visible side-effect will need a way to pause until it is guaranteed that such broadcasts have actually been delivered. For this purpose, a **flush** primitive is provided. Occasional calls to **flush** do not eliminate the benefit of using *OBCAST* asynchronously. Unless the system has built up a considerable backlog of undelivered broadcast messages, which should be rare, **flush** will only pause while transmission of the last few broadcasts completes.

5.6. Group addressing protocol

Since group membership can change dynamically, it may be difficult for a process to compute a list of destinations to which a message should be sent, for example, as is needed to perform a *GBCAST*. In [Birman-b] we report on a protocol for ensuring that a given broadcast will be delivered to all members of a process group in the same view. This view is either the view that was operative when the message transmission was initiated, or a view that was defined subsequently. The algorithm is a simple iterative one that costs nothing unless the group membership changes, and permits the caching of possibly inaccurate membership information near processes that might want to communicate with a group. Using the protocol, a flexible message addressing scheme can readily be supported.

5.7. Example

Figure 1 illustrates a pair of computations interacting with a process group while its membership changes dynamically. One client issues a pair of *OBCAST*'s, then uses *BCAST* to perform a third request on the group. A second client interacts only once, using *BCAST*. Note that unless the first client invoked flush before issuing the *BCAST*, the *BCAST* might be received before the prior *OBCAST*'s at some sites. Arrows showing reply messages have been omitted to simplify the figure, but it would normally be the case that one or more group members reply to each request.

6. Using the primitives

The reliable communication primitives described above dramatically simplify the solution of the problems cited in Sec. 2:

1. *Synchronization*. Many synchronization problems are subsumed into the primitives themselves. For example, consider the use of *GBCAST* to implement recovery. A recovering process would issue a *GBCAST* to the process group members, requesting that state

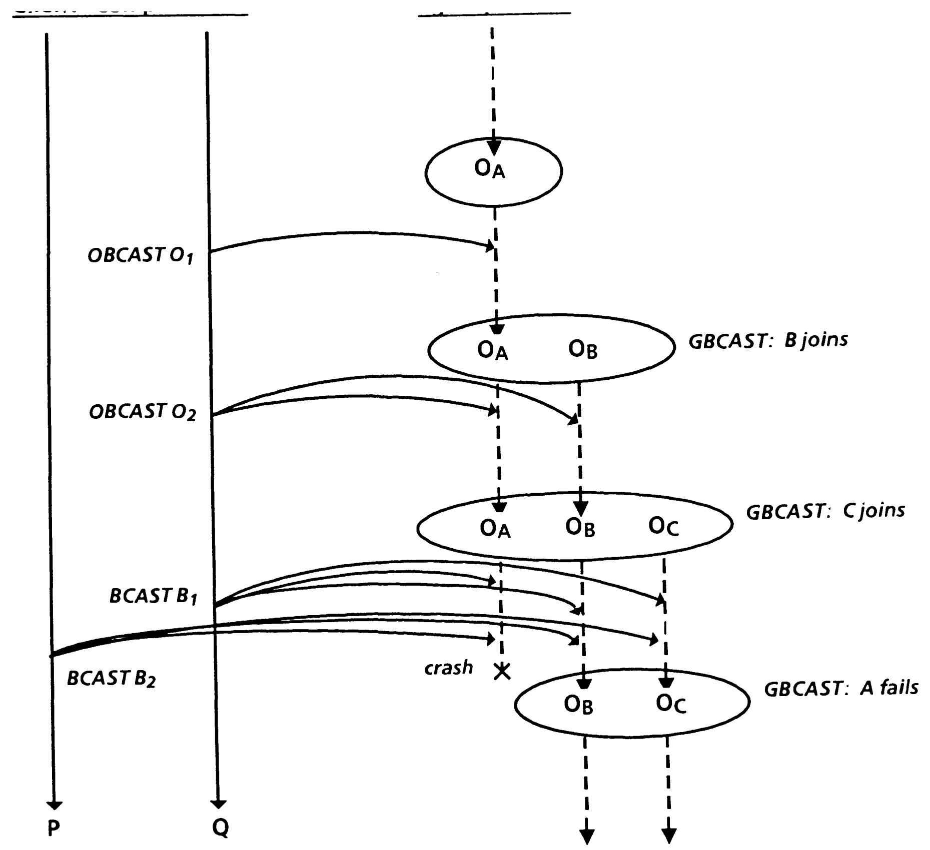

Figure 1: Client processes interacting with a process group

information be transferred to it. In addition to sending the current state of the group to the recovering process, group members update the process group view at this time. Subsequent messages to the group will be delivered to the recovered process, with all necessary synchronization being provided by the ordering properties of *GBCAST*. In situations where other forms of synchronization are needed, *BCAST* provides a simple way to ensure that several processes take actions in the same order, and this form of low-level synchronization simplifies a number of higher-level synchronization problems. For example, if *BCAST* is used to request write-locks from lock-manager processes, two write-lock requests on the same item can never deadlock by being granted in different orders by a pair of managers.

2. *Fault detection*. Consistent failure (and recovery) detection are trivial using our primitives: a process simply waits for the appropriate process group view to change. This facilitates the implementation of algorithms in which one processes monitors the status of another process. A process that acts on the basis of a process group view change does so with the assurance that other group members will (eventually) observe the same event and will take consistent actions.

3. *Consistency*. We believe that consistency is generally expressible as a set of atomicity and ordering constraints on message delivery, particularly causal ones of the sort provided by *OBCAST*. Our primitives permit a process to specify the communication properties needed to achieve a desired form of consistency. Continued research will be needed to understand precisely how to pick the weakest primitive in a designated situation.

4. *Serializability*. To achieve serializability, one implements a concurrency control algorithm and then forces computations to respect the serialization order that this algorithm choses. The *BCAST* primitive, as observed above, is a powerful tool for establishing an order between concurrent events. Having established such an order, *OBCAST* can be used to distribute information about the computation and also its termination (commit or abort). Any process that observes the commit or abort of a computation will only be able to interact with data managers that have received messages preceding the commit or abort, hence a highly asynchronous transactional execution results. This problem is discussed in more detail in [Birman-a] [Joseph-a] [Joseph-b].

7. Implementation

The communication primitives can be built in layers, starting with a bare network providing unreliable datagrams. A site-to-site acknowledgement protocol converts this into a sequenced, error-free message abstraction, using timeouts to detect apparent failures. An agreement protocol is then used to order the site-failures and recoveries consistently. If timeouts cause a failure to be detected erroneously, the protocol forces the affected site to undergo recovery.

Built on this is a layer that supports the primitives themselves. *OBCAST* has a very lightweight implementation, based on the idea of flooding the system with copies of a message: Each process buffers copies of any messages needed to ensure the consistency of its view of the system. If message m is delivered to process p, and m is potentially causally dependent on a message m', then a copy of m' is sent to p as well (duplicates are discarded). A garbage collector deletes superfluous copies after a message has reached all its destinations. By using extensive piggybacking and a simple scheduling algorithm to control message transmission, the cost of an *OBCAST* is kept low -- often, less than one packet per destination. *BCAST* employs a two-phase protocol based on one suggested to us by Skeen [Skeen-b]. This protocol has higher latency than *OBCAST* because delivery can only occur during the second phase; *BCAST* is thus inherently synchronous. In *ISIS*, however, *BCAST* is used rarely; we believe that this would be the case in other systems as well. *GBCAST* is implemented using a two-phase protocol similar to the one for *BCAST*, but with an additional mechanism that flushes messages from a failed process before delivering the *GBCAST* announcing the failure. Although *GBCAST* is slow, it is used even less often than *BCAST*. Preliminary performance figures appear in [Birman-b].

8. Applications of the approach

Our work with communication primitives has convinced us that the resilient objects provided by the *ISIS* system exist at too high a level for many sorts of distributed application. For example, consider the cognac still shown in figure 2. If independent, non-identical computer systems were used to control distillation, two aspects would have to be addressed. First, it would be necessary to design the hardware itself in a way that admits safe actions in all possible system states. Second, however, one would need to implement the control software in each processor in a way that ensures mutual consistency of the operational computing units. That is, given that the

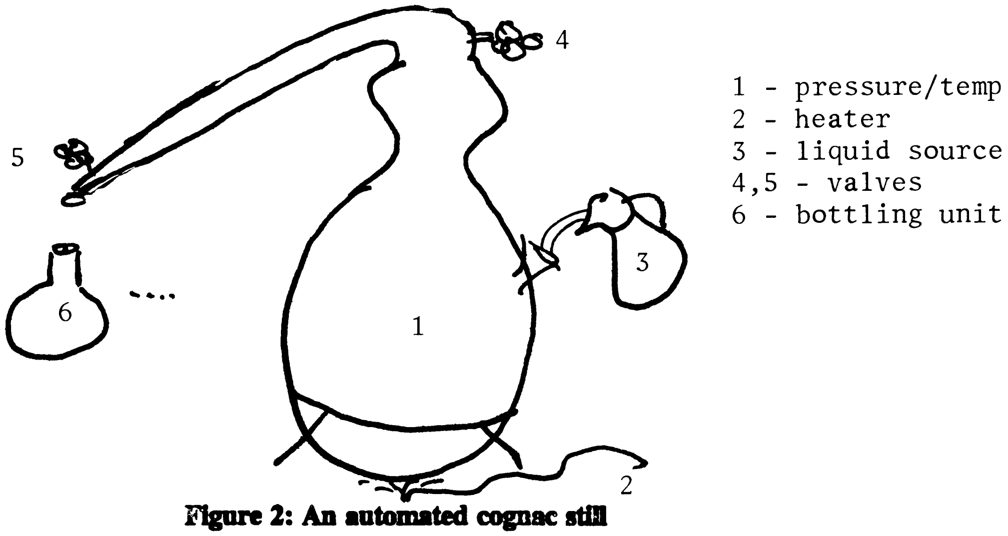

Figure 2: An automated cognac still

specification describes a sequence of actions to take in some scenario (for example, detection of excessive pressure in the distillation vessel), can we be assured that the operational processors will *jointly* act to avert a disastrous spill of cognac? We believe that fault-tolerant process groups provide a simple, elegant way to address problems such as this one. We plan to complete an implementation of the protocols by the summer of 1986, and then to develop a collection of software subsystems running on top of them.

9. Acknowledgement

The authors are grateful to Pat Stephenson and Fred Schneider for many suggestions that are reflected in the presentation of this material, and to Dale Skeen, with whom we colaborated on many aspects of the work reported here.

10. References

[Allchin] Allchin, J., McKendry, M. Synchronization and recovery of actions. *Proc. 2nd ACM SIGACT/SIGOPS Principles of Distributed Computing*, Montreal, Canada, 1983.

[Babaoglu] Babaoglu, O., Drummond, R. The streets of Byzantium: Network architectures for fast reliable broadcast. *IEEE Trans. on Software Engineering* **TSE-11**, 6 (June 1985).

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

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

[Birman-b] Birman, K., Joseph, T. Reliable communication in an unreliable environment. Dept. of Computer Science, Cornell Univ., TR 85-694, Aug. 1985.

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

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

[Cheriton] Cheriton, D. The V Kernel: A software base for distributed systems. *IEEE Software* 1 12, (1984), 19-43.

[Cooper] Cooper, E. Replicated procedure call. *Proc. 3rd ACM Symposium on Principles of Distributed Computing.*, August 1984, 220-232. (May 1985).

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

[Eswaren] Eswaren, K.P., *et al* The notion of consistency and predicate locks in a database system.

Comm. ACM **19**, 11 (Nov. 1976), 624-633.

[Hadzilacos] Hadzilacos, V. Hadzilacos, V. Byzantine agreement under restricted types of failures (not telling the truth is different from telling of lies). Tech. ARep. TR-19-83, Aiken Comp. Lab., Harvard University (June 1983).

[Halpern] Halpern, J., and Moses, Y. Knowledge and common knowledge in a distributed environment. Tech. Report RJ-4421, IBM San Jose Research Laboratory, 1984.

[Joseph-a] Joseph, T. Low cost management of replicated data. Ph.D. dissertation, Dept. of Computer Science, Cornell Univ., Ithaca (Dec. 1985).

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

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

[Lazowska] Lazowska, E. *et al* The architecture of the EDEN system. *Proc. 8th Symposium on Operating Systems Principles*, Dec. 1981, 148-159.

[Liskov] Liskov, B., Scheifler, R. Guardians and actions: Linguistic support for robust, distributed programs. *ACM TOPLAS* **5**, 3 (July 1983), 381-404.

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

[Papadimitrou] Papadimitrou, C. The serializability of concurrent database updates. *JACM* **26**, 4 (Oct. 1979), 631-653.

[Popek] Popek, G. *et al.* Locus: A network transparent, high reliability distributed system. *Proc. 8th Symposium on Operating Systems Principles*, Dec. 1981, 169-177.

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

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

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

[Skeen-b] Skeen, D. A reliable broadcast protocol. *Unpublished.*

[Spector] Spector, A., *et al* Distributed transactions for reliable systems. *Proc. 10th ACM SIGOPS Symposium on Operating Systems Principles*, Dec. 1985, 127-146.

[Strong] Strong, H.R., Dolev, D. Byzantine agreement. *Digest of papers, Spring Compcon 83*, San Francisco, CA, March 1983, 77-81.

REPORT DOCUMENTATION PAGE

Exp. Date: Jun 30, 1986

REPORT SECURITY CLASSIFICATION Unclassified	1b. RESTRICTIVE MARKINGS
SECURITY CLASSIFICATION AUTHORITY	3. DISTRIBUTION/AVAILABILITY OF REPORT Approved for Public Release Distribution Unlimited
DECLASSIFICATION/DOWNGRADING SCHEDULE	
PERFORMING ORGANIZATION REPORT NUMBER(S) TR86-753	5. MONITORING ORGANIZATION REPORT NUMBER(S)
NAME OF PERFORMING ORGANIZATION Kenneth Birman, Dept. of CS Cornell University | 6b. OFFICE SYMBOL (If applicable)	7a. NAME OF MONITORING ORGANIZATION Defense Advanced Research Projects Agency/IPT
ADDRESS (City, State, and ZIP Code) Dept. of Computer Science, 405 Upson Hall Cornell University Ithaca, NY 14853	7b. ADDRESS (City, State, and ZIP Code) Defense Advanced Research, Project Agency Attn: TIO/Admin, 1400 Wilson Blvd. Arlington, VA 22209
NAME OF FUNDING/SPONSORING ORGANIZATION DARPA/IPTO | 8b. OFFICE SYMBOL (If applicable)	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER ARPA order 5378 Contract MDA-903-85-C-0124
ADDRESS (City, State, and ZIP Code) Defense Advanced Research, Project Agency Arlington, VA 22209	10. SOURCE OF FUNDING NUMBERS PROGRAM ELEMENT NO. | PROJECT NO. | TASK NO. | WORK UNIT ACCESSION NO

TITLE (Include Security Classification)

Communication Support for Reliable Distributed Computing

Approved for Public Rele
Distributed Unlimited

PERSONAL AUTHOR(S)

Kenneth P. Birman and Thomas A. Joseph

TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT (Year, Month, Day)	15. PAGE COUNT
Special Technical	FROM ____ TO ____	May 1986	16

SUPPLEMENTARY NOTATION

COSATI CODES			18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
FIELD	GROUP	SUB-GROUP	

ABSTRACT (Continue on reverse if necessary and identify by block number)

We describe a collection of communication primitives integrated with a mechanism or handling process failure and recovery. These primitives facilitate the implementation f fault-tolerant process groups, which can be used to provide distributed services in an nvironment subject to non-malicious crash failures.

DISTRIBUTION/AVAILABILITY OF ABSTRACT ☒ UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT. ☐ DTIC USERS	21. ABSTRACT SECURITY CLASSIFICATION	
NAME OF RESPONSIBLE INDIVIDUAL	22b. TELEPHONE (Include Area Code)	22c. OFFICE SYMBOL