K. P. Birman (Cornell)
Network Working Group
Request for Comments: 992
T. A. Joseph (Cornell)
November 1986

On Communication Support for Fault Tolerant Process Groups

K. P. Birman and T. A. Joseph
Dept. of Computer Science, Cornell University
Ithaca, N.Y. 14853
607-255-9199

1. Status of this Memo.

This memo describes a collection of multicast 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. Unlike other process group approaches, such as Cheriton's "host groups" (RFC's 966, 988, [Cheriton]), our approach provides powerful guarantees about the behavior of the communication subsystem when process group membership is changing dynamically, for example due to process or site failures, recoveries, or migration of a process from one site to another. Our approach also addresses delivery ordering issues that arise when multiple clients communicate with a process group concurrently, or a single client transmits multiple multicast messages to a group without pausing to wait until each is received. Moreover, the cost of the approach is low. An implementation is being undertaken at Cornell as part of the ISIS project.

Here, we argue that the form of "best effort" reliability provided by host groups may not address the requirements of those researchers who are building fault tolerant software. Our basic premise is that reliable handling of failures, recoveries, and dynamic process migration are important aspects of programming in distributed environments, and that communication support that provides unpredictable behavior in the presence of such events places an unacceptable burden of complexity on higher level application software. This complexity does not arise when using the fault-tolerant process group alternative.

This memo summarizes our approach and briefly contrasts it with other process group approaches. For a detailed discussion, together with figures that clarify the details of the approach, readers are referred to the papers cited below.

Distribution of this memo is unlimited.

2. Acknowledgments

This memo was adopted from a paper presented at the Asilomar workshop on fault-tolerant distributed computing, March 1986, and summarizes material from a technical report that was issued by Cornell University, Dept. of Computer Science, in August 1985, which will appear in ACM Transactions on Computer Systems in February 1987 [Birman-b]. Copies of these paper, and other relevant papers, are available on request from the author: Dept. of Computer Science, Cornell University, Ithaca, New York 14853. (birman@gvax.cs.cornell.edu). The ISIS project also maintains a mailing list. To be added to this list, contact M. Schmizzi (schiz@gvax.cs.cornell.edu).

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.

3. 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. This version of ISIS, reported in [Birman-a], supports transactional resilient objects as a basic programming abstraction. Our current work undertakes to provide a much broader range of fault-tolerant programming mechanisms, including fault-tolerant distributed bulletin boards [Birman-c] and fault-tolerant remote procedure calls on process groups [Birman-b]. The approach to communication that we report here arose as part of this new version of the ISIS system.

Unreliable communication primitives, such as the multicast group communication primitives proposed in RFC's 966 and 988 and in [Cheriton], leave some uncertainty in the delivery status of a message when failures and other exceptional events occur during communication. Instead, a form of "best effort" delivery is provided, but with the possibility that some member of a group of processes did not receive

the message if the group membership was changing just as communication took place. When we tried to use this sort of primitive in our original work on ISIS, which must behave reliably in the presence of such events, we had to address this aspect at an application level. The resulting software was complex, difficult to reason about, and filled with obscure bugs, and we were eventually forced to abandon the entire approach as infeasible.

A wide range of reliable communication primitives have been proposed in the literature, and we became convinced that by using them, the complexity of our software could be greatly reduced. These range

from reliable and atomic broadcast [Chang] [Cristian] [Schneider] to Byzantine agreement [Strong]. For several reasons, however, the existing work does not solve the problem at hand. The most obvious is that they do not provide a mechanism for sending a message to all the members of a group when the membership is changing dynamically (the "group addressing" problem). In addition, one can identify delivery ordering issues and questions regarding the detection of communication failures that should be handled within the broadcast mechanism. These motivate a careful reexamination of the entire reliable broadcast problem.

The multicast primitives we report here 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 multicast 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. Such groups are similar to the host groups of the V system and the ones described in RFC's 966 and 988, but provide guarantees of consistency in just the situations where a host group provides a "best effort" delivery which may sometimes be erroneous.

It is helpful to think of our primitives as providing a logical or "virtual" form of reliability: rather than addressing physical delivery issues, they ensure that a client will never observe a system state "inconsistent" with the assumption that reliable delivery

has occurred. Readers familiar with serializability theory may want to think of this as a weaker analog: in serializability, one allows interleaved executions of operations provided that the resulting system state is consistent with the assumption that execution was sequential. Similarly, reliable communication primitives permit deviations from the reliable delivery abstraction provided that the resulting system state is indistinguishable from one in which reliable delivery actually did occur.

Using our 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 the algorithms that are needed to maintain high availability even when failures, recoveries, or process migration occurs. More recently, we have applied the same approach to a variety of other problems in distributed computing, and even designed a consistent, fault tolerant, distributed bulletin board data structure (a generalized version of the blackboards used in artificial intelligence programs), with equally good results [Birman-c]. Thus, we feel that the approach has been shown to work in a variety of settings where unreliable primitives simply could not be used.

In the remainder of this memo we summarize 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, such as is used in the ARGUS system [Liskov], and the fault-tolerant process group mechanism mentioned above. We argue that transactional interactions are too restrictive to support the sort of mechanism needed, and then show how our primitives can be used to provide such a mechanism. We conclude by speculating on future directions in which this work might be taken.

4. Issues in fault-tolerance

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. Thus, a mechanism is needed for ordering conflicting events. Additional problems that can arise in this context include deadlock avoidance or detection, livelock avoidance, etc.

[2]Failure 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]. Regardless of how this problem is solved, some sort of reliable failure detection mechanism will be needed in any fault-tolerant distributed system.

[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 A holds, and on the basis of this interacts with process q, the state of q should not contradict the fact that p believes A to be true. This problem is closely related to notions of knowledge and consistency in distributed systems [Halpern] [Lamport]. In our context, A will often be the assertion that a multicast has been received by q, or that q saw some sequence of events occur in the

same order as did p. Thus, it is necessary to be able to specify the precise consistency constraints on a distributed software system, and system support should be available to facilitate the attainment of these constraints.

[4] Serializability. Many distributed systems are partitioned into data manager processes, which implement shared variables, and transaction manager processes, which issue requests to data managers [Bernstein]. If transaction managers can execute concurrently, it is 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. Thus, one needs a way to achieve serializability in applications where it will be needed, without imposing system-wide restrictions that would prevent the design of software subsystems for which serializability is not needed.

Jointly, these problems render the design of fault-tolerant distributed software daunting in the absence of adequate support. The correctness of any proposed design and of its implementation become serious, if not insurmountable, concerns. In Sec. 7, we will show how the primitives of Sec. 6 provide simple ways to overcome all of these issues.

5. Existing alternatives

If one rules out "unreliable" communication mechanisms, there are basically two fault-tolerant alternatives that can be pursued.

The first 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 whereas transactions are easily implemented on top of fault-tolerant process groups -- we have done so -- the converse is much harder. Moreover, transactions

represent a relatively heavy-weight solution to the problems surveyed in the previous section, and might impose an unacceptable overhead on subsystems that need to run non-transactionally, for example because a pair of concurrent processes needs to interact on a frequent basis. (We are not saying that "transactional" mechanisms such as cobegins and toplevel actions can't solve this problem, but just that they yield a solution that is awkward and costly). This sort of reasoning has lead us to focus on non-transactional interaction mechanisms, and to treat transactions as a special class of mechanisms used when processes that have been designed to employ a transactional protocol interact.

The second 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. Above, we termed this the fault tolerant 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] and in UNIX, and more recently in the ISIS system. A proposal to provide Internet support for host groups was raised in RFC's 966 and 988. However, the idea of adapting the process group approach to work reliably in an environment subject to the sorts of exception events and concurrency cited in the previous section seems to be new.

As noted earlier, existing reliable communication protocols do not address the requirements of fault-tolerant process groups. For example, in [Schneider], an implementation of a reliable multicast 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 (sent to all sites in a system) 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 multicast 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 deeply 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 on knowledge theoretic grounds

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 multicasts. For reasons discussed below, this tends to result in high latency in typical local networking environments. An additional 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 multicasts. Since this problem arises in any setting where one process monitors another, we felt it should be addressed at the same level as the communication protocol. Finally, one wants a group oriented multicast protocol, not a site oriented broadcast, and this issue must be resolved too.

6. Our multicast primitives

We now describe three multicast protocols - GBCAST, ABCAST, and CBCAST - 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. Group addressing is discussed in Sec. 6.5.

The failure model that one adopts has a 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 also assume that individual processes can crash, and that this is detected when it occurs by a monitoring mechanism present at each site. Further assumptions are sometimes made about the availability of synchronized realtime clocks. Here, we adopt the position that although reasonably accurate elapsed-time clocks may be available, closely synchronized clocks probably will not be. 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. Even with advancing technology, it seems safe to assume that clock skew will remain "large" when compared to inter-site message transmission speed. In particular, this argues against time-based protocols such as the one used in [Cristian]

6.1 The GBCAST primitive

GBCAST (group multicast) 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. 6.5).

Our GBCAST protocol ensures that if any process receives a multicast B before receiving a GBCAST G, then all overlapping destinations will receive B before G <1> This is true regardless of the type of multicast involved. Moreover, when a failure occurs, the corresponding GBCAST message is delivered after any other multicasts 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 VIEW's 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 multicasts, all members receiving a given multicast will have the same value of VIEW when they receive it.

Notice that GBCAST also provides a convenient way to change other global properties of a group "atomically". In our work, we have used GBCAST to dynamically change a ranking on the members of a group, to request that group members establish checkpoints for use if recovery is needed after all failure, and to implement process migration. In each case, the ordering of GBCAST relative

to other events that makes it possible to perform the desired action without running any additional protocol. Other uses for GBCAST will no doubt emerge as our research continues.

Members of a process group can also use the value of VIEW 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 value of VIEW that becomes defined on stable storage before using it; a simplified version of the algorithm in [Skeen-a] can then be executed when recovering from failure.

6.2 The ABCAST 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 multicast messages is not necessary. Our second primitive, ABCAST (atomic multicast), satisfies such a weaker constraint. Specifically, it is often desired that if two multicasts 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 ABCAST 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 ABCAST(msg, label, dests) provides this behavior. Two ABCAST's having the same label are delivered in the same order at all common destinations.

6.3 The CBCAST primitive

Our third primitive, CBCAST (causal multicast), is weakest in the sense that it involves less distributed synchronization then GBCAST or ABCAST. CBCAST(msg, dests) atomically delivers msg to each operational dest. The CBCAST protocol ensures that if two multicasts are potentially causally dependent on another, then the former is delivered after the latter at all overlapping destinations. A multicast B' is potentially causally dependent on a multicast B if both multicasts originate from the same process, and B' is sent after B, 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 to the process that issued B' [Lamport]. For causally independent multicasts, the delivery ordering is not constrained.

CBCAST 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 multicasts, and it is precisely this sort of order that CBCAST respects.

6.4 Other multicast primitives

A weaker multicast primitive is reliable multicast, which provides all-or-nothing delivery, but no ordering properties. The formulation of CBCAST in [Birman-b] actually includes a mechanism for performing multicasts of this sort, hence no special

primitive is needed for the purpose. Additionally, there may be situations in which ABCAST protocols that also satisfy a CBCAST ordering property would be valuable. Our ABCAST primitive could be changed to respect such a rule, and we made use of a multicast primitive that is simultaneously causal and atomic in our work on consistent shared bulletin boards ([Birman-c]). For simplicity, the presentation here assumes that ABCAST is completely orthogonal to CBCAST, but a simple way to build an efficient "causal atomic" multicast is described in our full-length paper. The cost of this protocol is only slightly higher than that of ABCAST.

6.5 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 multicast 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.

Iterative addressing is only required when the process transmitting a message has an inaccurate copy of the process group view. In the implementation we are now building, this would rarely be the case, and iteration is never needed if the view is known to be accurate. Thus, iterated delivery should be very infrequent.

6.6 Synchronous versus asynchronous multicast abstractions

Many systems employ RPC internally, as a lowest level primitive for interaction between processes. It should be evident that all of our multicast 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 multicast in which no replies, or just one reply, suffices.

In our work on ISIS, GBCAST and ABCAST 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, CBCAST, which is the most frequently used overall, is almost never invoked synchronously. Asynchronous CBCAST's are the

primary source of concurrency in ISIS: although the delivery ord-

ering 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 multicast indefinitely, the ability to defer it a little, without delaying some computation by doing so, permits load to be smoothed. Since CBCAST respects the delivery orderings on which a computation might depend, and is ordered with respect to failures, the concurrency introduced does not complicate higher level algorithms. Moreover, the protocol itself is extremely cheap.

A problem is introduced by our decision to allow asynchronous multicasts: the atomic reception property must now be extended to address causally related sequences of asynchronous messages. If a failure were to result in some multicasts 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, all messages m' that s received prior to its failure must be delivered to their remaining operational destinations. This is because the state of t may now depend on the contents of any such m', hence the system state could become inconsistent if the delivery of m' were not completed. 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 multicasts 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 multicasts have actually been delivered. For this purpose, a flush primitive is provided. Occasional calls to flush do not eliminate the benefit of using CBCAST asynchronously. Unless the system has built up a considerable backlog of undelivered multicast messages, which should be rare, flush will only pause while transmission of the last few multicasts complete.

7. Using the primitives

The reliable communication primitives described above lead to simple solutions for the problems cited in Sec. 4:

[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

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, ABCAST 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 ABCAST is used to do P() and V() operations on a distributed semaphore, the order of operations on the semaphore is set by the ABCAST, hence all the managers of the semaphore see these operations in a fixed order.

[2] Failure 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 CBCAST. 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 ABCAST primitive, as observed above, is a powerful tool for establishing an order between concurrent events, e.g. by lock acquisition. Having established such an order, CBCAST 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. If a process running a computation fails, this is detected when a failure GBCAST is received instead of the commit. Thus, executions are simple and quite deterministic.

If commit is conditional, CBCAST would be used to first interrogate participants to learn if they are prepared to commit, and then to transmit the commit or abort decision (the usual two-

phase commit). On the other hand, conditional commits can often be avoided using our approach. A method for building transactions that will roll-forward after failure after failure is discussed in more detail in [Birman-a] [Joseph-a] [Joseph-b]. Other forms of concurrency control, such as timestamp generation, can similarly be implemented using ABCAST and CBCAST. We view transactional data storage as an application-level concern, which can be handled using a version stack approach or a multi-version store, or any other appropriate mechanism.

8. Implementation

The communication primitives can be built in layers, starting with a bare network providing unreliable Internet datagrams. The software structure is, however, less mature and more complex than the one suggested in RFC's 966 and 988. For example, at this stage of our research we do not understand how to optimize our protocols to the same extent as for the unreliable host multicast approach described in those RFC's. Thus, the implementation we describe here should be understood to be a prototype. A particularly intriguing question, which we are investigating actively, concerns the use of a "best effort" ethernet or Internet multicast as a tool to optimize the implementation of our protocols.

Our basic approach is to view large area networks as a set of clusters of sites interconnected by high speed LAN devices and interconnected by slower long-haul links. We first provide protocols for use within clusters, and then extend them to run between clusters too. Network partitioning can be tolerated at all levels of the hierarchy in the sense that no incorrect actions can result after network partitioning, although our approach will sometimes block until the partition is repaired. Our protocols also tend to block within a clus-

ter while the list of operational sites for that cluster is being changed. In normal LAN's, this happens infrequently (during site failure or recovery), and would not pose a problem. (In failure intensive applications, alternative protocols might be needed to address this issue).

The lowest level of our software uses a site-to-site acknowledgement protocol to convert the unreliable packet transport this into a sequenced, error-free message abstraction, using timeouts to detect apparent failures. TCP can also be used for this purpose, provided that a "filter" is placed on the incoming message stream and certain types of messages are handled specially. 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. CBCAST has a very light-weight 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 prime, then a copy of m prime 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 a CBCAST is kept low -- often, less than one packet per destination. ABCAST employs a two-phase protocol based on one suggested to us by Skeen [Skeen-b]. This protocol has higher latency than CBCAST because delivery can only occur during the second phase; ABCAST is thus inherently synchronous. In ISIS, however, ABCAST 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 ABCAST, but with an additional mechanism that flushes messages from a failed process before delivering the GBCAST announcing the failure. Although GBCAST is slower than ABCAST or CBCAST, it is used rarely enough so that performance is probably less of an issue here -- and in any case, even GBCAST could be tuned to give very high throughput. Preliminary performance figures appear in [Birman-b].

Although satisfactory performance should be possible using an implementation that sits on top of a conventional Internet mechanism, it should be noted that to achieve really high rates of communication the layers of software described above must reside in the kernel, because they run on behalf of large numbers of clients, run frequently, and tend to execute for very brief periods before doing I/O and pausing. A non-kernel implementation will thus incur high scheduling and context switching overhead. Additionally, it is not at all clear how to use ethernet style broadcast mechanisms to optimize the performance of this sort of protocol, although it should be possible. We view this as an interesting area for research.

A forthcoming paper will describe higher level software that we are building on top of the basic fault-tolerant process group mechanism described above.

9. Conclusions

The experience of implementing a substantial fault-tolerant system left us with insights into the properties to be desired from a communication subsystem. In particular, we became convinced that to build a reliable distributed system, one must start with a reliable communication subsystem. The multicast primitives described in this memo present a simple interface, achieve a high level of concurrency, can be used in both local and wide area networks, and are applicable to software ranging from distributed database systems to the fault-tolerant objects and bulletin boards provided by ISIS. Because they are integrated with failure handling mechanisms and respect desired event orderings, they introduce a desirable form of determinism into

distributed computation without compromising efficiency. A consequence is that high-level algorithms are greatly simplified, reducing the probability of error. We believe that this is a very promising and practical approach to building large fault-tolerant distributed systems, and it is the only one we know of that leads to a rigorous form of confidence in the resulting software.

NOTES:

<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 assumption. Thus, GBCAST satisfies the sort of logical delivery property cited in the introduction.

10. References

[RFC966] Deering, S. and Cheriton, D. Host groups: A multicast extension to the internet protocol. Stanford University, December 1985.

[RFC988] Deering, S. Host extensions for IP multicasting. Stanford University, July 1986.

[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 multicast. 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 the presence of failures. Dept. of Computer Science, Cornell Univ., TR 85-694, Aug. 1985. To appear in ACM TOCS (Feb. 1987).

[Birman-c] Birman, K., Joseph, T., Stephenson, P. Programming with fault tolerant bulletin boards in asynchronous distributed systems. Dept. of Computer Science, Cornell Univ., TR 85-788, Aug. 1986.

[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 multicast 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 multicast: 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. 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 multicast 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 multicast 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.