Design Alternatives for Process Group Membership and Multicast*

Kenneth P. Birman**
Robert Cooper
Barry Gleeson

TR 91-1257
(replaces 91-1185)
December 1991

P - 33

Department of Computer Science
Cornell University
Ithaca, NY 14853-7501

*This paper is a revision of TR91-1185 (Jan. 1991)
**The first two authors are in the Dept. of Computer Science, Cornell University, and were supported under DARPA/NASA grant NAG 2-593. The third author is with the UNISYS Corporation, San Jose, CA.

Design Alternatives for Process Group Membership and Multicast[†]

Kenneth P. Birman   Robert Cooper   Barry Gleeson

December 18, 1991

Abstract

Process groups are a natural tool for distributed programming, and are increasingly important in distributed computing environments. However, there is little agreement on the most appropriate semantics for process group membership and group communication. These issues are of special importance in the Isis system, a toolkit for distributed programming [Bir91]. Isis supports several styles of process group, and a collection of group communication protocols spanning a range of atomicity and ordering properties. This flexibility makes Isis adaptable to a variety of applications, but is also a source of complexity that limits performance. This paper reports on a new architecture that arose from an effort to simplify Isis process group semantics. Our findings include a refined notion of how the *clients* of a group should be treated, what the properties of a multicast primitive should be when systems contain large numbers of overlapping groups, and a new construct called the *causality domain*. As an illustration, we apply the architecture to the problem of converting processes into fault-tolerant process groups in a manner that is "transparent" to other processes in the system. A system based on this architecture is now being implemented in collaboration with the Chorus and Mach projects.

Keywords: distributed computing, fault-tolerance, Isis, process groups, virtual synchrony, causal multicast, atomic broadcast.

1 Introduction

Isis is a toolkit for distributed programming that provides a set of problem-oriented tools built around process groups and reliable group multicast [BJ87, BSS91]. Process groups are a natural abstraction and have been used in a number of distributed systems [CZ85, OSS80, KTHB89.

[†]This paper is a revision of TR 91-1185 (Jan. 1991)

[*]The first two authors are in the Dept. of Computer Science, Cornell University, and were supported under DARPA/NASA grant NAG-2-593. The third author is with the UNISYS Corporation, San Jose, Ca.

LLS90, PBS89, AGHR89]. However, the precise characteristics of group facilities differ among these systems, as do the protocols employed to implement them. The primary goal of this paper is to sort through the design choices at this level, arriving at a process group architecture that is simple, powerful and appropriate. A secondary goal is that the architecture should admit elegant solutions to classical problems in this area, such as transforming a program into an equivalent fault-tolerant one, without sacrificing efficiency.

As evidence in support of our arguments we show how the architecture can be used to derive a simple fault-tolerance transformation. Consistent with our goals, the solution would (theoretically) perform as well as the best known solutions to this problem. Despite the fact that it would achieve high levels of concurrency, the solution is fully described at a high level and is surprisingly easy to understand.

Our analysis draws on experience with the Isis system, which has been distributed to hundreds of sites since the first public software release in 1987. Isis is presently used in diverse settings such as brokerage and banking applications, value-added telecommunications systems, wide-area seismic data collection and analysis, factory floor automation, document flow, distributed simulation, scientific computing, high-availability file management, reactive control, database integration, education and research [BC90]. Through participation in the design of a number of these distributed systems, we have gained insight both into the successful aspects of the technology, and those in need of further work.

Successful Isis applications often share two characteristics:

- *They depend on consistent, distributed process group state.* Isis provides tools for reading and writing replicated data, adapting to failures, transferring group data to new members, and viewing group membership. Many Isis applications using these tools rely on the guarantee that group members see *mutually consistent* sequences of updates for replicated information, and that a process can join the group and obtain its "current" state without possibly missing an update or seeing one twice. This property is useful for more than just replication of data. For example, group members are able to react to external events in a coordinated way, treating the group membership list as a form of data replicated among the members without running an additional agreement protocol.

- *They employ large numbers of groups.* Isis was designed assuming that typical applications would be organized into some (small) number of fault-tolerant distributed servers, each implemented using a single process group. However, many Isis users seized upon groups as a fine-grained structuring construct, building applications with large numbers of overlapping groups. This trend motivates several of the architectural changes discussed below.

Groups are used in a variety of ways in Isis applications:

- *Groups used for fault-tolerance.* Here, some of the components of a system are transparently replaced by fault-tolerant process groups that mimic the original components. As we demonstrate in Section 5, our architecture permits this to be done without changing programs that interact with the modified components.

- *Groups as services with clients.* In this case, group members provide services to *client* programs, either in a request-reply style, or through a registration interface with repeated callbacks (e.g. a broker's workstation might subscribe to a stock price publication service, receiving callbacks each time the price changes). Multi-level servers are common, with the processes that implement one service registering as clients of other services.

- *Process groups for distributed or replicated objects.* In these applications, an object is typically an abstract data type with small state[1] that may change rapidly. Reasons for replicating objects include improved fault-tolerance, and increased performance through concurrency or coherently replicated data.

- *Groups used for parallel programming.* Several scientific computing projects have employed Isis to obtain coarse grained parallelism and fault-tolerance in simulations and graphics applications, running on networks of high-performance workstations.

- *Groups used for fault-tolerant, distributed system management.* Isis has been used in application-oriented monitoring and control software for high-reliability, autonomous, distributed systems. The underlying application will often make no explicit use of Isis, although hooks may be included to permit the monitoring system to intervene when necessary.

The numbers and uses of groups differ substantially from our original expectations, dating to when Isis was first developed. This has brought into question several of the basic assumptions underlying the initial architecture, leading us to ask how the system might need to be re-designed to simplify future development, improve performance and exploit emerging operating systems and hardware technologies, such as communication devices supporting high-speed multicast.

This paper focuses upon the following questions:

- Why is explicit system support for process groups and group communication necessary?

[1]Larger database-style objects would normally be managed using conventional database packages. Isis tools can be combined with such packages, and a mechanism for dealing with databases is included within the toolkit.

- What types of groups are needed in distributed systems, and what patterns of client-server interactions should be supported?
- What should be the semantics of communication and membership in a single process group?
- How should these semantics be extended to multiple, overlapping groups?
- How can a process group system take advantage of the emerging generation of modular operating systems?

We note that although the paper is intended to be self-contained and to define the terminology used, the issues considered here arise from the many, often contradictory, approaches to process groups and group communication that have been advanced. This results in a somewhat abrieviated presentation of some of the alternatives, and may make the paper difficult to read without some prior knowledge of the field.

2 Process groups

This section refines our terminology and confronts the first of the design questions: at what level groups and multicast should implemented.

2.1 Group membership

A *process group* is a collection of *communication endpoints* that can be referenced as a single entity. Communication endpoints can be implemented in a number of ways. In Unix, each Isis process creates a socket which can be referenced by its internet address. A communication endpoint would correspond to a send-right in Mach, an entity-ID in the V-System, a port UI in Chorus, or a capability in Amoeba. We assume multiple threads sharing an address space (i.e. a process in Unix or Amoeba, a task in Mach, or an actor in Chorus). This permits an address space to own several communication end-points, thus decoupling us from any specific model of processes or memory. Following the conventions of other group-based projects and the original Isis implementation, we will continue with the term *process group*, in this paper, rather than *port group*. However, our new architecture does allow multiple end-points per process.

2.2 Why provide support for process groups?

The process group membership mechanism comprises the algorithms used to support joining and leaving groups, and to query the current membership list. One might ask whether these operations are more appropriately realized at the application level, or in a shared software subsystem such as Isis. Three issues arise: the importance and generality of the group mechanism, the performance implications of an application-level implementation, and the complexity of the solution.

- *Standardization.* In Isis applications, process groups are a basic and heavily used programming construct. Assuming that a single, general mechanism can support a diverse user community without becoming encumbered by numerous special features – and we will argue that this is so – standardization has obvious benefits.
- *Complexity.* The protocols required to support process groups are subtle and difficult to implement correctly. If non-experts are to use group-based programming structures, such as replicated data, there may be no choice but to implement the group mechanism in a shared subsystem.
- *Performance.* The complexity of the protocols implies that it will be difficult to make all the necessary engineering decisions and performance trade-offs correctly. For example, it is by no means clear *a-priori* whether membership lists should be replicated at all group members, or cached at some smaller set of sites. In fact, we believe that there are strong technical and performance arguments in favor of a direct replication approach, but these arguments come down to engineering considerations that a typical user of a system might not be knowledgeable enough to make.[2]

In a shared software subsystem, these issues would be addressed by the implementor of the subsystem – not by the authors of the applications that use the subsystem. This is desirable because it permits the largest possible set of users to benefit from the insight of a small, expert group of designers.

Our preference for a system-supplied mechanism that explicitly manages group membership and replicates this information directly among the members may seem unreasonably biased in favor of making communication cheap at the expense of a more costly group membership facility. One might question this choice. As a matter of fact, we are familiar with applications in which changes to group membership are more frequent than communication. Fortunately, it is generally possible to convert "membership intensive" applications into communication intensive ones. For example, consider an application in which messages are sent to *the set of idle servers* in a compute service. If servers perform short tasks, membership in this group could vary rapidly. On the other hand, the full set

[2]Our work on Isis employs protocols in which replicating membership information has important performance advantages. But, it has taken us years of protocol design, implementation, and experimentation to arrive at this conclusion, and it is unlikely that a typical programmer would employ the best known solution if this is at all complex.

of servers probably changes slowly. Our experience suggests rapidly changing ad-hoc groups are almost always subsets of more stable enclosing groups. Given a system in which group membership changes are relatively costly but communication is cheap, a a cost-effective solution would be to have the server group treat the "subset of lightly loaded servers" as a form of dynamically updated replicated data. Changes to the subset will now be cheap.

We conclude that a system-level group facility is needed, and that accurate knowledge of group membership should be available to processes that commonly initiate multicasts to the group.

2.3 Which processes should be allowed to send to a group?

In some systems [LLS90, PBS89] only members of a group may multicast to it. This simplifies group management but does not reflect the way programmers use groups, at least in Isis. In such an approach, *client* programs that wish to communicate with a service implemented by a group must either join the group (which does not scale well), or use point-to-point communication with individual group members (requiring the application programmer to implement a non-trivial protocol, and in particular to solve a difficult fault-tolerance problem in the case where the "agent" fails).

We believe that process groups will often have both members and clients, and hence that this issue will be commonly encountered in any system supporting group programming. For example, a common use of groups in future distributed systems will be to make a system component fault-tolerant using replication (we give a protocol for this in Section 5). Here, the fault-tolerant program will be the group, and the programs that interact with it will be clients. Moreover, one would not wish to require that such clients be aware that they are interacting with a group, as opposed to a single entity.

We conclude that a client-server model should be supported, in which clients can communicate reliably and transparently with groups. Implications of supporting such a notion of "clients" will be examined in depth in Section 3.1.

2.4 Should group multicast provide "strong guarantees"?

Early work on process groups, such as the work in the V-system [CZ85], provided best-effort communication guarantees. Given a process group with stable membership, and assuming that nothing fails and that the communication subsystem is working reasonably well, the V multicast delivers a message to all group members. If any of these guarantees is not satisfied, some members might not receive a message. Moreover, the order in which messages are delivered can differ from member to member.

Isis differs from the V-system in adopting a multicast layer with very strong semantics: a program that uses Isis multicast knows exactly what to expect. We believe that this is one of the major reasons that Isis has turned out to be so easy to program in comparison with V, where the group multicast facility was used primarily to locate resources. However, it is not enough to simply accept that a multicast should provide strong guarantees. Multicast can be presented in many ways, and with many sorts of guarantees. What options exist at this level of a system?

Various models of multicast interaction have been proposed: asynchronous, all-reply, one-reply, k-reply, and so forth. Isis supports all of these and our users have found them all important. Moreover, a group may receive multiple multicasts concurrently, or a stream of multicasts from a single sender. For this reason, communication primitives often provide system-enforced ordering properties.[3] Other potentially important properties include *failure atomicity*, namely all-or-nothing delivery guarantees even if processes or processors fail during a multicast, and *membership atomicity*, namely the guarantee that group membership changes are totally ordered and synchronized with group communication.

Figure 1 illustrates two extremes for group communication. In an *unordered execution* no atomicity guarantees are provided. In a *closely synchronous* execution, one event occurs at a time, and multicast messages are delivered atomically to the full membership of the group at a single logical instant, during which both address expansion[4] and delivery occurs. The *virtually synchronous* execution model supported by Isis is indistinguishable from a closely synchronous execution for a correct program, but relaxes synchronization to improve performance. Multicast ordering and atomicity issues are discussed more fully in Sections 3.2 and 3.3

In section 3, we will discuss these options in some detail. To anticipate the conclusion of this discussion, we will argue that strong guarantees are important in most process-group based software. Lacking them, a system will be incapable of supporting important classes of applications. On the other hand, we will also suggest that unsophisticated users can (and should) be presented with a default form of multicast with very simple semantics. The idea is that naive users should employ a multicast primitive that is likely to behave as they would expect, while sophisticated users and

[3]In this paper, we consider only asynchronous systems, in which any timing constraints or deadlines are weak with respect to communication performance. Realtime communication protocols, such as the ones described in [CASD85], impose stringent timing requirements upon the operating system and frequently obtain determinism by introducing delays and idle periods. Few current Isis applications need deadlines or priorities, hence we have chosen to concentrate on "logical" properties, such as delivery ordering and atomicity, in this paper.

[4]We use the term *address expansion* to refer to the phase of a multicast during which the system determines the group members to which a message will be delivered.

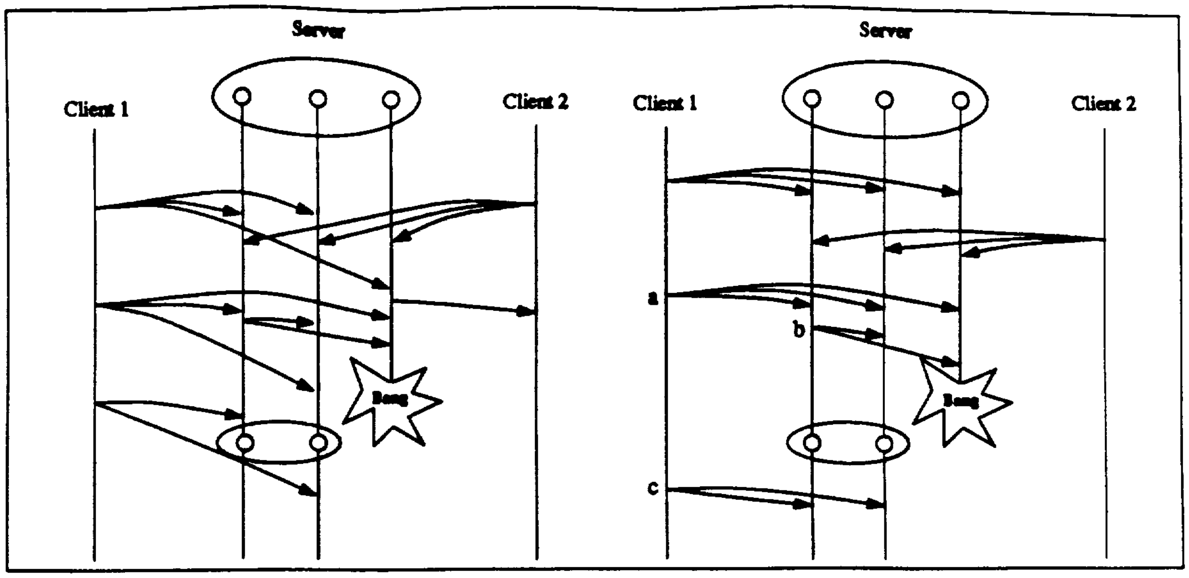

Figure 1: (a) Unordered group communication; (b) Synchronous group communication.

subsystems will need flexibility to achieve the highest possible performance.

2.5 Why not layer group communication over RPC?

A frequently-asked question concerns whether group communication should be implemented over RPC. Many current operating systems are RPC-based, and this protocol is often highly optimized and well supported. For this reason, if a user were to implement a multicast protocol at the application layer, this would have to be done over either RPC, a stream protocol such as TCP (which makes little sense),[5] or a datagram protocol such as UDP which, because it is unreliable and infrequently used, poses many practical obstacles.[6] Moreover, many styles of group communication are essentially generalizations of RPC, and many of the techniques used to support RPC carry over to group multicast protocols. Thus, it may seem natural to layer protocols such as group multicast over RPC, and to "grease the skids" so that RPC will be as fast as possible.

In principle, one could build a reliable multicast protocol over an RPC transport, and a group mechanism over this multicast. Given transactional RPC [LS83, Spe85], such a multicast could

[5]The problem with implementing multicast over TCP is that TCP is optimized for continuous, stream-style transmission of large quantities of data from one source to one destination. The protocol is mismatched with a burst, one-to-many communication pattern – a criticism that would not apply to RPC. The same comments apply to X.25, the OSI stream protocol.

[6]RPC protocols automatically deal with message loss and retransmission, fragmentation of large packets into small ones, etc. All of these problems would have to be addressed by hand in a protocol layered over UDP, or the equivalent OSI datagram protocol.

be made atomic, with parallel threads (lightweight processes) doing RPCs to deliver the messages, and using a two-phase commit to ensure atomicity. Of course, such a solution would also need to address the concerns of the remainder of this paper: multicast ordering, synchronization of multicast address expansion with group membership changes, etc. A protocol with predictable behavior in all of these respects would be no simpler over RPC than any other technology. The question, therefore, is one of performance.

Of special interest to us are applications that use *asynchronous* group communication to achieve high performance. Communication is *synchronous* if it follows a request-reply style, whereby the thread that sends a message blocks waiting for the reply. Asynchronous communication arises when the sending thread does not block and no reply message is sent. Although underlying message transport layers still need to exchange acknowledgement and flow-control messages, these impose little overhead and do not delay the higher-level protocols, or require further synchronization in the application.

Asynchronous communication has an obvious performance benefit if no replies are needed from the destination processes. This benefit becomes a necessity when the number of destinations grows large, because of the cost of collecting superfluous replies at the requester. Implementing an asynchronous multicast communication protocol over an RPC layer would cause severe congestion at the sender. A second factor is that multicast hardware would be very difficult to exploit from an RPC-based implementation. A third concern would be the potentially large amount of memory needed for the stacks of the threads associated with pending multicasts on the sender side: as many as one thread per destination per multicast.

Thus we conclude:

- Group membership management and group communication are commonly used services that should be implemented once, in a common shared subsystem.
- Multicast should be implemented over asynchronous message passing or transport-level multicast.

2.6 Does multicast belong inside the operating system?

There remains the question of whether multicast support should exist in the operating system or in a shared user-space library. The key issue, again, is performance. For good performance multicast should implemented "near the wire"; in other words, the latency of network device interrupts should be minimized.

To fully justify this claim we would need to review the protocols that have been offered in support of group multicast, an exercise that would exceed the scope of this paper. Briefly, though, any protocol for group multicast will involve delaying some messages and exchanging background messages of one sort or another. It follows that if all protocol messages must reach the user's address space, an expensive cross-address space call will have to be done (perhaps even a scheduling action and several context switches) just to deliver a message that might not trigger execution of any application-related code. The cost savings of putting at least the core functionality of the multicast mechanism in the operating system can thus be substantial.

Experimental work that has placed some form of multicast directly in the operating system shows that startling performance gains are attainable using this approach [DC90, KT91, PBS89]. These systems are as much as 25 times faster than the current UNIX-based Isis implementation, despite the fact that multicasts in this version of Isis substantially outperform other UNIX-based multicast protocols with which we are familiar [BSS91].

On the other hand, multicast will not be needed by *every* operating system user, so we should not require nor expect every operating system to provide it. Thus we are attracted by modular operating systems [AGHR89, Ras86] in which a small kernel and a collection of operating system modules communicate using fast inter-module calls. In this way, group and multicast support can be provided in a separate, optional operating system module.

We conclude that where possible (and notably in modular operating systems) group and multicast should be provided in a separate operating system module.

3 Detailed design choices for a single group

The goal of this section is to explore, in detail, the choices for group and multicast semantics within a single group. Section 4 explores issues raised when multiple groups co-exist in a single application.

3.1 Group structure: Members and clients

In Section 2.3 it was suggested that processes outside a group will often need to interact with the group as a single entity. From experience with Isis users, we have identified four group "structures" that frequently arise in Isis programs (Fig. 2). Each responds to a different programming need.

A *peer group* is composed of a set of members that cooperate closely. Fault-tolerance and load-sharing are dominant considerations in these groups, which are typically small. In a *client-server*

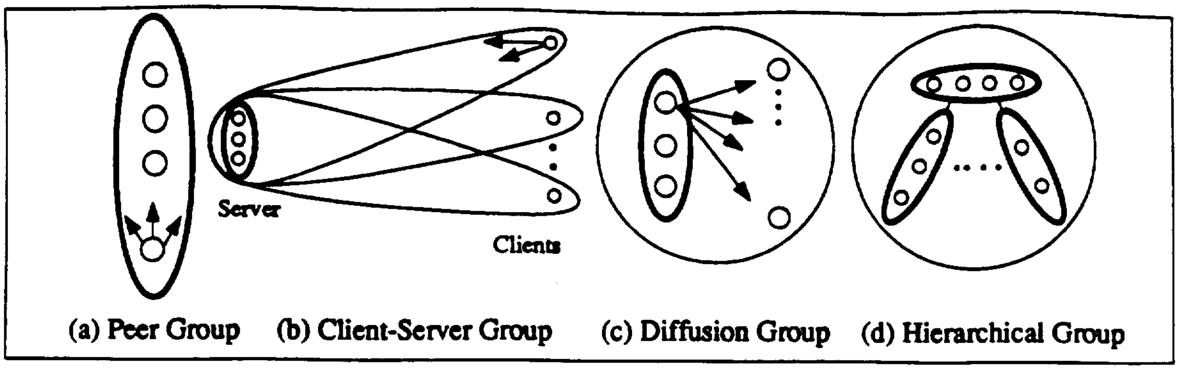

Figure 2: Common group structures

group, a potentially large number of clients interacts with a peer group of servers. Requests may be multicast or issued as RPCs to some favored server after an initial setup. The servers either respond to requests using point-to-point messages, or use multicast to atomically reply to the client while also sending copies to one-another. The latter approach is useful for fault-tolerance: if a primary server fails, multicast atomicity implies that a backup server will receive a copy if (and only if) the client did. Thus, a backup server will know which requests are still pending.

A special case of client-server communication arises in the *diffusion group*, which supports *diffusion multicasts*. Here, a single message is sent by a server to the full set of clients and servers. In current Isis applications, diffusion groups are the only situations in which a typical multicast has a large number of destinations. The use of multicast hardware to optimize this case is thus attractive.

These three cases are easily distinguished at runtime in Isis. The only explicit actions by the programmer are to register as a member (using the **pg_join** system call) or client (**pg_client**), and to designate diffusion multicasts using an option to the Isis multicast system call. A single group may operate in both client-server modes simultaneously.

The last common group structure is the *hierarchical group*. In large applications with a need for sharing, it is important to localize interactions within smaller clusters of components. This leads to an approach in which a conceptually large group is implemented as a collection of subgroups. In client-server applications with hierarchical server groups, the client is bound, transparently, to a subgroup that accepts requests on its behalf. A root group is responsible for performing this mapping, which is done using a *stub* linked into the client's address space that routes messages to the appropriate subgroup. The root group sets up this binding when a process becomes a group client, and may later re-bind the client to a different subgroup. Group data is partitioned so that only one subgroup holds the primary copy of any data item, with others either directing operations to the appropriate subgroup or maintaining cached copies. Multicast to the full set of group members is supported, but but its use is discouraged in this architecture.

For brevity, we omit detailed discussion of one-time client-server interactions, and groups used only to monitor membership, but never for communication. Both merit special treatment in an implementation. For example, a large membership-only group should be supported as a client-server structure, minimizing the number of processes informed on each membership change. The servers would be informed of monitoring requests and would only communicate with a client when a monitor is triggered.

Explicit support for these group structures is important for performance and scaling. Clients are more numerous than members, but clients of a group never communicate with each other via that group. This fact can be exploited to reduce the amount of information maintained per-client, and permits clients to be omitted from most group coordination protocols. If clients are treated as fully fledged group members (as required in most group-based systems) then groups may not provide sufficient performance for many applications.

3.2 Atomicity

In Section 2, we suggested a need for multicast primitives supporting strong semantics. In this section, we begin a more detailed examination of the options by looking at the question of atomicity. As stated earlier, a process group system may support two forms of atomicity: *membership atomicity* and *failure atomicity*. The first provides the illusion of group membership that changes instantaneously as members join, leave or fail. The second ensures that multicasts interrupted by a crash will be transparently terminated. Isis supports both properties, and these have proved important to users of the system.

Consider first the atomicity of group join/leave/fail. It is difficult to program with process groups in which the expansion of a multicast address from a group address to a list of members is not atomic (i.e. there is no guarantee concerning exactly which processes received a particular multicast, as illustrated in Fig. 3.a). In Isis, this is guaranteed to be the complete membership of the group. defined at a logical instant when delivery occurs (Fig. 3.b).

Similar comments apply to failure atomicity. Process group algorithms are greatly simplified by the ability to send a multicast without the concern that an unlikely event, such as a crash, will result in partial delivery. When a group member fails, Isis further guarantees that other processes will receive the failure notification only after having received all outstanding messages from the failed process, and that failures leave no gaps in the causal message history. These properties eliminate bizarre failure sequences, such as the delivery of a message from a process after system state maintained for that process has been garbage collected.

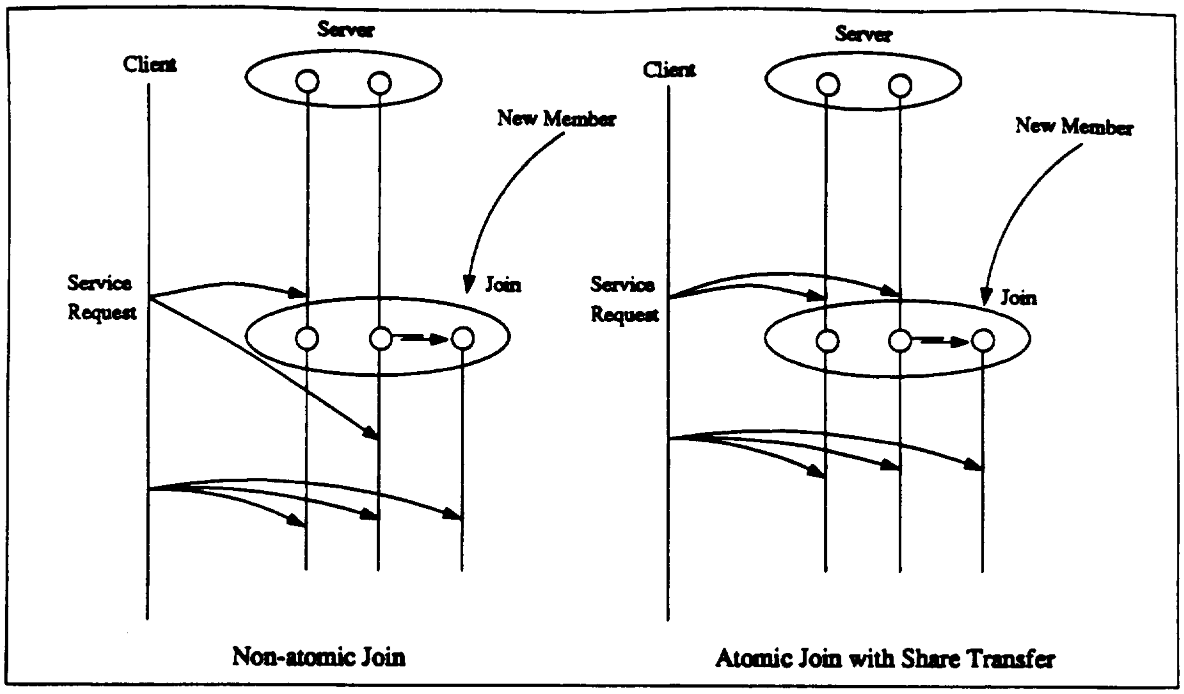

Figure 3: (a) A non-atomic join (b) Atomic join.

Although some systems, notably the V-System, have developed applications using non-atomic group semantics, the primary use seems to be in name services that use multicast for service location. In this context, the consequences of a missed reply or an inaccurate membership protocol are simply an occasional loss of performance.

Isis tools and applications build other forms of atomicity on top of the membership and failure atomicity semantics of groups. For example, the Isis *state transfer* tool copies data from an existing group member to a joining process. (The application designer determines what the state should include.) State transfer is a key to supporting groups with consistent distributed state. However, it is important that the state transferred correspond to the programmer's notion of group state at the (logical) instant of the join. Obtaining this property requires that state transfer be synchronized with the reception of messages that might change the state. Specifically, all messages sent to the group before the new member was added must be delivered before the state is sent. Messages delivered to the group after this event must include the new member. Finally, the event by which the old and new members are informed of the membership change (through a callback) must be coordinated to occur at the same point in the execution of each. We believe that, in the absence of strong atomicity properties, it would be impossible to define (much less implement) state transfer.

Membership atomicity is useful for another reason: it gives process group members *implicit knowledge* about one-another's states. This permits each group member to use the same deterministic function for choosing the primary site in a data replication algorithm, or for subdividing work in a parallel computation, for example. Because of membership atomicity, this function operates only on local data (the synchronized group membership list) but achieves group-wide consistency. Several Isis tools are driven by atomic group membership changes, making no use of any other communication between group members.

We conclude that in systems like Isis, membership atomicity and failure atomicity are both needed.

3.3 Causal and total multicast orderings

In Section 2.4, we observed that there are many possible multicast delivery ordering guarantees. This section focuses on the choice between causal and total ordering in a single group, while the following sections examine multicast ordering in systems with large numbers of possibly overlapping process groups.

Although Isis supports a number of multicast ordering alternatives, application builders are primarily concerned with two of these, **cbcast** and **abcast**. The **cbcast** protocol delivers messages in the order they were sent, (the *causal* or *happens before* order that is natural in distributed systems [Lam78]). For example, in Fig. 1.b, multicast a causally precedes multicasts b and c, but b and c are concurrent. **Cbcast** would therefore deliver a before b or c, at all destinations but the relative delivery order used for b and c would be unconstrained and might vary from process to process.

That **cbcast** does not order concurrent multicasts is not necessarily a drawback. Often, application-level synchronization or scheduling mechanisms are used to serialize conflicting operations: further serialization of multicasts is superfluous. **Cbcast** is attractive in such cases, because there is no built-in delay associated with the algorithm. In fact **cbcast** never delays a message unless it arrives out of order.

The **abcast** protocol delivers messages to group members in a single mutually observed order. Referring to Fig. 1.b, this implies that processes s_1, s_2 and s_3 would receive multicasts a, b and c in the same order. This extra ordering comes at a significant cost: *any* **abcast** protocol delays some (or all) messages during the period when this order is being determined. For example, in one common implementation of **abcast**, recipients of a message wait for an *ordering message* from a distinguished *sequencer* process. The nature of the delay varies from protocol to protocol, but the presence of a delay of this sort is intrinsic to the **abcast** ordering property.

The performance implications of using abcast instead of cbcast

The extra delay with **abcast** can lengthen the critical path of a distributed computation. In a common usage of multicast, a process multicasts an operation to a group that includes itself, and upon receiving its own multicast performs the operation. By acting on the operation after it has received its own multicast the process is certain that it is performing the operation in an order consistent with the other members of its group, and that the other members are guaranteed to receive the multicast and could take over the operation should this process fail (because of failure atomicity). Where **abcast** is used, the sending process may not act on the message until a total ordering for delivering it has been decided. Unless the sender is also the sequencer (which is not generally the case) this delay will involve a remote communication. In contrast a **cbcast** implementation need never delay delivery of the message at the sending process, and in general delivery at one destination is never delayed because of slow response at another destination. In this sense, a **cbcast** implementation can be optimal.

Schmuck has shown that distributed algorithms can be built primarily from **cbcast** [Sch88, BJ89]. This is done by demonstrating that most algorithms can be recoded in a style that enforces mutual exclusion between conflicting operations, for which **cbcast** suffices.

In Isis, this transformation is used extensively for performance reasons: the **abcast**-based algorithms may be simpler to understand, but are often much slower. In particular, the latency between transmission and delivery of an **cbcast** is at least a factor of two smaller than for **abcast**. Moreover, at the *sender*, the difference can be a factor of one hundred or more. The problem is that if the sender needs a copy of its own message, in the same order as the other group members will see it (i.e. for a replicated update), **abcast** will block while **cbcast** can be used without blocking. This is because **abcast** has to deal with the case where two senders concurrently communicate to the same group. Even if this is uncommon, **abcast** cannot deliver the message to any destination until it is known to be the "next" one, and this requires some communication with other potential senders. In contrast, **cbcast** can be delivered immediately at the sender.

Distributed systems, and indeed computing systems of all sorts, are notoriously bursty: often there will be very few active threads. By blocking the sender of a multicast, **abcast** may delay one of the only things going on in the entire system! Thus, in applications where the sender of a multicast is also a destination, the benefit of using **cbcast** instead of **abcast** can be dramatic.

To summarize, we have identified a two-level issue. First, asynchronous systems are likely to outperform synchronous systems by a substantial factor (in the current version of Isis, as much as one to two orders of magnitude). Second, given a system that uses multicast communication, the **cbcast** delivery ordering property will be substantially cheaper to provide then the **abcast** property, and this is true regardless of whether the sender uses the protocol synchronously or asynchronously.

The pervasiveness of causality obligations

Abcast may seem strictly stronger (more ordered) than **cbcast**, since concurrent multicasts are ordered. However, **abcast**, in most definitions, is actually not required to use an order consistent with causality. Consider a process that sends two asynchronous **abcast** messages. It would be normal to expect that these be delivered in the order sent, and most **abcast** protocols have this property in the absence of failures. However such a non-causal (or "mostly causal") **abcast** should not be used asynchronously because it does not *guarantee* this property. For these reasons we believe that **abcast** should support both a total and a causal order. Such a *causal* **abcast** protocol can be built over **cbcast** [BSS91].[7]

In discussing the option of building multicast over RPC, we stressed the need for asynchronous communication, and in the discussion of the previous section reiterated this issue. Indeed, delay is often the most serious threat to performance in distributed systems. Delays are especially apparent in applications that maintain replicated data using read and write operations, with a locking or token passing scheme used to avoid conflicts. Any delay when doing a read or write operation may be visible to the user of such an application. On the other hand, the latency before all replicas are updated is invisible unless it impacts on read or write response times, or on availability. Using a causally consistent communication protocol, one can code completely asynchronous replicated data management algorithms—regardless of whether that protocol is **abcast** or **cbcast**. The user programs as if updates were synchronous, and the causal ordering property, combined with failure atomicity, ensure that the execution respects this logical property [BJ87, BJ89, Sch88, LLS90]. Equally, a protocol that might violate causality is unsafe for asynchronous use, even if it still provides a total order. Unless causal obligations are observed, the initiator of an operation must wait until completion of the operation is acknowledged before proceeding. Otherwise the total order might enforce an arbitrary serialization that violates causality.

By the same reasoning, it must be possible for point-to-point communication in a process group

[7]Those familiar with the previous Isis work will wonder where the gbcast protocol fits into this. In the original versions of Isis, **abcast** and **cbcast** were completely unordered with respect to each other. Gbcast was totally ordered with respect to both **abcast** and **cbcast**, and was needed to implement group membership atomicity. However, some applications also used the protocol. The equivalent of gbcast is still present within the group join mechanism, and is implemented using a cbcast that triggers a group flush prior to deliver. However, we have determined that Isis users who employed gbcast at the application level generally could have obtained the same effect using a causally ordered abcast, and that given this primitive, gbcast can be viewed as a purely internal mechanism. This simplifies groups as seen by users.

setting to convey the causality obligations. For instance in a computation spanning two processes, one process may initiate an asynchronous multicast, and then send an RPC to the other process, which initiates a second asynchronous multicast. The second multicast should causally follow the first. In Isis a point-to-point **cbcast** achieves this effect.

Message stability

The use of asynchronous communication raises an additional problem of message stability. A message is said to be *k-stable* if its delivery is assured provided that no more than k failures occur, and is *stable* (where k is omitted) if delivery is certain to occur. For example, suppose that a process, p_1, sends multicast a to processes p_2 and p_3. Process p_2 receives a and sends multicast b to p_3. If a was not stable at the time of its delivery to p_2, the failure of p_1 might prevent a from (ever) being delivered to p_3. This represents a form of communication deadlock, since messages from p_2 to p_3 will now be delayed indefinitely. A related issue arises if process p_2 takes an externally visible action based on the reception of a. Here, it may be that p_2 should delay the action until a and its causal predecessors are stable, since failures might otherwise create a situation in which an irreversible action was taken but no operational process in the system realizes this.

Although these problems can be avoided by delaying delivery of a message until it and all of its causal antecedents are stable, this introduces a tradeoff between the levels of performance and safety needed in the application. We favor allowing messages to be delivered before they become stable, and providing a per-group **pg_flush** operation that delays the caller until stability is achieved for any asynchronous messages pending in the group, and for their causal predecessors. We are also considering a system call to specify the stability parameter k for a given group. An analogous problem arises in file systems, when output to a disk is cached or buffered, and is typically solved in a similar way by providing a system call such as the Unix **fsync** operation.

Summary

To summarize the arguments in this section:

- Asynchronous operations are the single most important factor in obtaining good performance in distributed systems, regardless of the underlying communication primitive.
- Asynchronous operations create causal delivery obligations, hence group communication should respect causality.

- **Cbcast** is used to implement causal **abcast**, hence it should be the core communication protocol in our process group architecture.
- (Causal) **abcast** is slower than **cbcast** and should be avoided by sophisticated users. Less sophisticated users find **abcast** easier to understand and should avoid **cbcast**.
- The message stability problem closely resembles the problem of flushing file system I/O buffers, and is readily addressed by providing a user-callable flush primitive.

4 Ordering properties that span group boundaries

The **Isis** system is notable for enforcing multicast ordering properties across group boundaries. Here we re-evaluate the usefulness of these semantics, while considering their cost and complexity.

4.1 Who uses overlapping groups?

Many Isis applications employ multiple, overlapping groups. In object-oriented applications group overlap is often carried to an extreme. Here, each program is typically composed of some set of objects, and any object that maintains distributed state is implemented by a group. A single process may thus belong to many groups. Large numbers of groups also arise when Isis is used for transparent fault-tolerance in the process pair style [Bar81], with a shadow process backing up each real process. Here, each communication entity in the system is represented by a group containing two members: a primary and a backup. Some Isis applications superimpose multiple groups on the same set of processes. For example, in a stock trading application, a service that computes bid/offered prices for a stock (a diffusion group) might also provide historical information on demand (a request-reply interaction). Moreover, individual processes within the server set may well subscribe to other services.

4.2 Should causality be preserved between groups?

Consider a graphics application that uses a blackboard object, containing the scene model, and a task-queue object, specifying views to be rendered (see Figure 4). Both objects allow asynchronous updates. A typical execution sequence involves posting data about a problem on the blackboard and then adding new tasks to the task list. Idle servers remove these tasks and consult the blackboard for scene data.

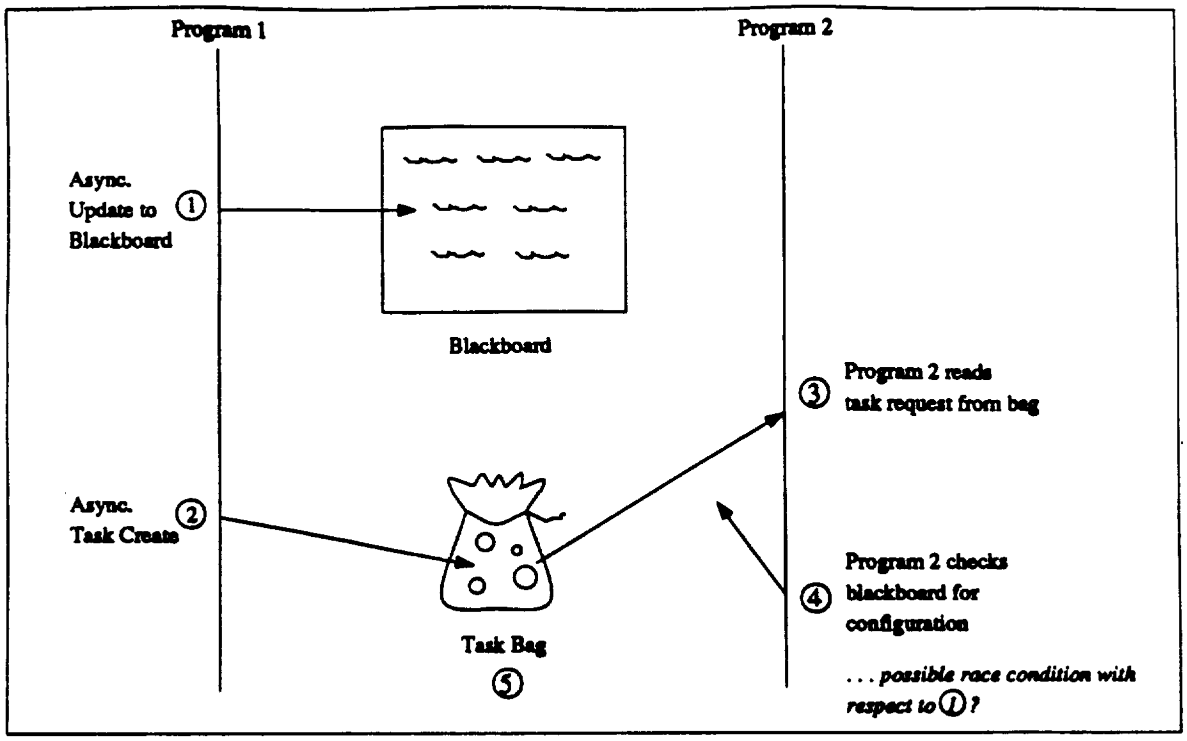
Program 1
Program 2
Async. Update to ① Blackboard
Blackboard
③ Program 2 reads task request from bag
Async. Task Create ②
Program 2 checks ④ blackboard for configuration
Task Bag ⑤
... possible race condition with respect to ①?

Figure 4: Application using a blackboard and a task-bag.

For fault tolerance or performance reasons, the blackboard and the task bag might both be implemented as process groups. Let us call the blackboard group B and the task bag group T. Group B has some number of members, and at least two clients: Program 1 (*p1*) and Program 2 (*p2*). Similarly, group T has *p1* and *p2* as clients. Thus, these two groups overlap at *p1* and *p2*.

For correct behavior, it is essential that when server *p2* consults the blackboard (step 4 in Figure 4), it finds the data that *p1* posted before putting *p2*'s task in the task bag. There are two ways this could be accomplished:

- Make *p1* wait at step 1 until it receives an acknowledgement from group B, indicating that the parameters have been posted, before adding the task to the task bag, or
- Make *p2* wait at step 4 if the blackboard update from step 1 is not yet complete.

These two solutions perform *very* differently. It is highly unlikely that the blackboard update will not be complete at the time *p2* executes step 4. The first solution delays *p1* *every* time it posts data to the blackboard, just to cover the unlikely case. The second solution only delays execution of *p2* when absolutely necessary, and never delays *p1* (except possibly for flow control reasons).

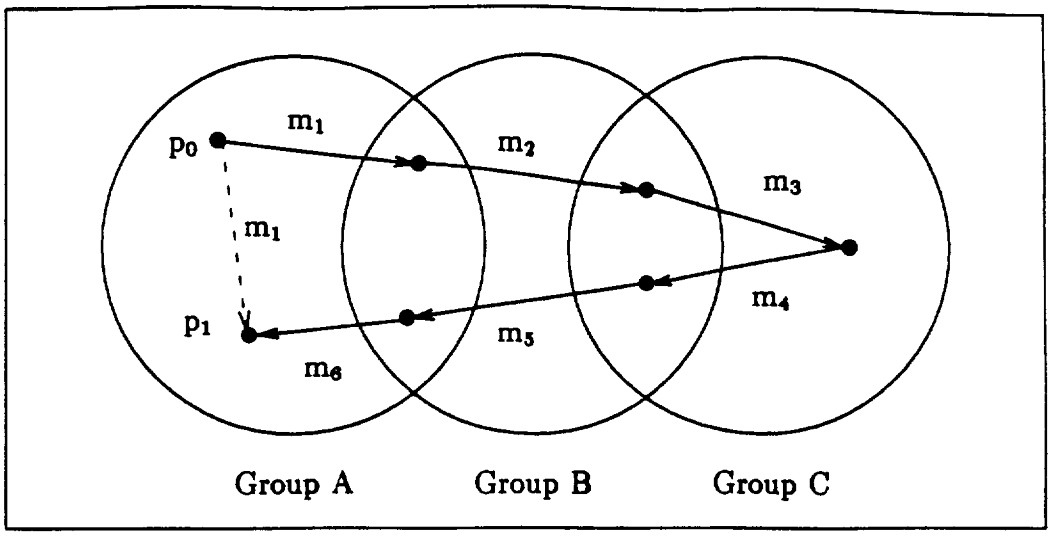

Figure 5: A causal chain spanning multiple groups

Of course, to implement the second solution, there must be some way to recognize that the message sent by *p2* at step 4 causally follows the message sent by *p1* at step 1. This causality obligation in group *B* must somehow be propagated through the task bag (group *T*).

In general, **cbcast** is used to ensure that sequences of causally related message events are processed in order. Where overlapping groups are concerned, the question is whether causal ordering should be enforced when a chain of events leaves some group, spans other groups, and then some operation re-enters the original group. This situation is schematically depicted in Figure 5. Here, the conflict arises within a *single* group, between the original operation and a later, causally dependent one. In a sense, each chain of causally related events represents an execution sequence, similar to a thread of control, that must be honored. Our belief in an asynchronous style of computation argues that causality should be preserved here.

4.3 Should causality *always* be preserved between groups?

Suppose that the author of our graphics task bag and blackboard application decides to include a debugging facility. This debugger should be able to halt execution of the entire application, then provide the user with facilities for probing the state of each of the application's processes. Suspending execution could be done with an asynchronous multicast to a group containing all the processes to be debugged. The debugging process could then communicate with the various application processes via RPC—invoking special state reporting code in each process.

Suppose that the new debugger is invoked in the situation described in the previous section. Execution is halted just before server *p2* consults the blackboard (step 4 in Figure 4.) Suppose further that server *p1*'s asynchronous blackboard update (step 1 in Figure 4) has not yet been posted.

If interactions with the debugger respect causality, the debugger is now in a very difficult position. If it interrogates the state of server *p1* or *p2*, or the task bag, it will lose its ability to interact with the blackboard. The problem is that when the debugger receives a message (e.g. an RPC reply) from *p1, p2* or the task bag, the *debugger's* execution becomes causally dependent on *p1*'s asynchronous blackboard update (step 1 in Figure 5). Thus, messages from the debugger to the blackboard are constrained to be delivered *after* the message in step 1 is delivered. Since normal execution of the blackboard has been halted, neither message can be delivered.

(Note that it was possible for the debugger to halt execution as described because the debugger's first message is not causally related to any activity in the debugged processes. Thus, the debugger's halt message can be delivered to the blackboard before *p1*'s update message, without violating causality.)

Clearly, it would be preferable if messages between the debugger and the debugged processes were completely unrelated to the messages between the debugged processes.

Other circumstances where it seems inappropriate to preserve causality between groups include:

- Performance Monitoring. The issues here are identical to those associated with debugging.
- Out-of-band Communication. **cbcast** is a generalization of FIFO message ordering: it *prevents* "out-of-band" communication.
- Background "bookkeeping" algorithms, such as garbage collection, deadlock detection, and orphan detection.

There is a more general way to look at these examples. Consider a program built of multiple independent subsystems. Each of these subsystems might be composed of several objects, represented by process groups, between which causality should be preserved. Yet, the subsystems may be completely independent of each other, and in some settings (e.g. when an applications combines several subsystems that run at different priorities), the potential delays introduced by the need to enforce inter-group causality would be inappropriate.

In the examples above, the debugging and monitoring parts of the application are subsystems that must run at higher priority than the basic graphics application, while bookkeeping operations typically run at a lower priority.

4.4 How visible should causality information be?

The examples above argue that the application programmer must have some control over the propagation of causality information. What form should this control take? What granularity of control is required?

Other researchers, such as Peterson [PBS89] and Ladin [LLS90], have proposed schemes in which users play a direct role in maintaining, transmitting and reasoning about causality information. Such approaches allow a sophisticated user—or a clever compiler—to exploit application semantics inaccessible to the runtime subsystem.

Our approach, in contrast, is motivated by the observation that naive programmers expect causal order to be respected as a matter of course. Indeed, some Isis users employ asynchronous communication without really understanding the causality issue at all. The decision to respect causal order means that such users will be able to develop correct code; a decision not to respect causality would have exposed subtle race conditions. Further, we have observed that although requirements for breaking causal order do arise, they are often related to the existence of sophisticated, independently developed, subsystems.

This leads us to favor a declarative approach in which explicit action must be taken to prevent the system from enforcing causal ordering. Our proposal is that groups be created in a specific *causality domain*. If the domain is not specified, a standard "default" domain would be used. Causality is observed only between groups in the same domain. A causality domain resembles a Psync *session* [PBS89], but may contain multiple, overlapping process groups. Naive developers would accept this default; thus placing the groups in their applications into a single causality domain. Sophisticated users—such as the author of the debugging package for our task and blackboard graphics application above—would take explicit action to ensure that debugging communication occurs in a separate causality domain.

Our emphasis is thus on simplicity of use—at the possible expense of concurrency. We prefer to enforce the occasional spurious causal ordering, rather than requiring that all programmers decide which causal information should be propagated where.

The presentation of causality information points to the broader question of how process groups should be presented within programming languages and object oriented environments. Systematic study of these issues will be needed if process groups are to become a common and widely used programming tool. One of us (Cooper) is currently examining these issues in the context of a distributed variant of Concurrent ML [Rep90].

4.5 Should abcast be ordered between groups?

The total order achieved by **abcast** is used to serialize independent requests to a process group, providing a simple form of mutual exclusion or concurrency control. When groups represent distinct *objects*, there is generally no need for **abcast** ordering to be observed at group overlaps (i.e. when two or more objects reside at the same process). Rather, each object is responsible for its own concurrency control (e.g. to maintain one-copy semantics for replicated data), and the object implementations are usually separate and non-interfering. In these cases a single-group **abcast** will ensure serializability, while the causality semantics of **abcast** will ensure that the relative ordering of requests at different objects is observed.

However these assumptions, while common, do not always hold. An object could be known by more than one group address, or there may be no direct mapping between groups and objects. One example would be overlapping diffusion groups (see Section 2.4) consisting of the same set of server processes, and intersecting sets of clients. One can imagine applications in which **abcasts** from the servers should be ordered totally at the overlapping client sets.

For an abstract example, consider a distributed form of the dining philosopher's problem. For each philosopher there is a process group that includes the pair of forks to use. One might use **abcasts** to atomically claim or release the forks for a given philosopher. Notice that no two processes (forks) receive the same pair of multicasts. Yet, **abcast** ordering is important here, because if **abcast** is not *globally* ordered, a cyclic request ordering could arise that would cause a deadlock. This example highlights a subtlety with multiple group **abcast** semantics. There are two reasonable generalizations of single group ordering. In the first, two concurrent **abcasts**, one to each of two overlapping groups, are ordered totally, but only at the processes in the intersection of the groups. In the second, stronger, definition **abcast** delivery is globally ordered. The first definition permits cycles in **abcast** delivery orderings; the second does not [GT90].

While we can create abstract examples to motivate multiple group **abcast** ordering, we have yet to see *practical* situations where this kind of ordering is necessary. Further, protocols that provide global order are more costly than protocols that are ordered only within a single group: in the current Isis protocols, a causal, locally ordered **abcast** is more than twice as fast as the best causal, globally ordered **abcast** protocol we could devise. This perhaps argues for a notion of *ordering domains*, analogous to causality domains. For example, one might provide a global **abcast** order within the subgroups of a hierarchical group, but not between two "unrelated" groups. However, we are unconvinced that ordering domains would see much use. For the moment, we are implementing single group **abcast** semantics and will re-evaluate this decision in the light of further experience.

To summarize:

- In most cases, causality should be preserved when a communication chain leaves and re-enters a group.
- Causality domains allow the scope of causality obligations to be restricted, in particular for applications with subsystems that must not interfere with one another.
- The **abcast** ordering is normally not needed when multicasts to two different groups happen to overlap. An exception arises when the two groups arise in a single object. Were this common, it would argue for a notion of *ordering domain* similar to the one for causality.

5 Extended example: Causal process pairs

To better justify our assertions, we now present a process-pair scheme for fault tolerance designed to be as efficient as possible within our architecture. The example illustrates several points. First, essentially all the issues discussed above arise, and the choices favored in the previous sections lead to simple solutions. Second, the performance of the overall fault-tolerance solution would be quite good – theoretically, as good or better than any previously known solution (we recognize that until we complete a full implementation and compare it directly to an implementation of some other method, this claim lacks the force of an experimental result). Finally, the example demonstrates that our architecture permits an obvious and important problem to be addressed in an elegant way using general primitives, suggesting that hand-crafted solutions to these sorts of problems are not necessarily preferable to solutions layered over a more standard subsystem.

We are given a system consisting of processes $\{P, Q, ...\}$ that communicate by sending point-to-point messages, and we wish to make some of these processes tolerant to single crash failures in a manner that is as transparent as possible to the programmer. This problem has been explored by many researchers and companies [Bar81, BBG+89, SY85, JZ87].

The basic idea of process pairs is to maintain a *backup process* for each *primary process* that we wish to make fault tolerant. The backup process keeps itself synchronized with the primary by keeping a checkpoint of the state of the primary, duplicates of any requests sent to the primary subsequent to the checkpoint, and enough supplemental data to overcome non-determinism in the execution.

For each primary process, P, let P' denote its backup. As illustrated in Figure 6, a process P will send a request r to the process pair (Q, Q') by first sending a *trace* message m to its backup, and then sending the request, r, using a multicast to the pair (Q, Q'). More specifically:

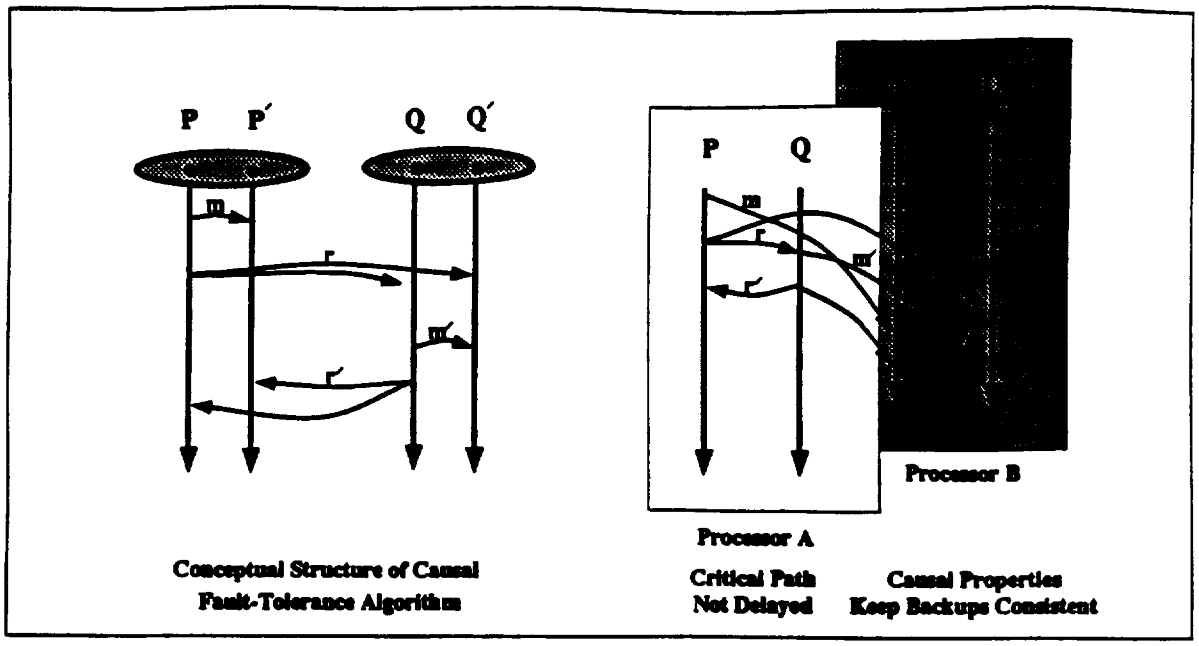
P
P′
Q
Q′
m
r
m′
r′
P
Q
m
r
m′
r′
Processor B
Processor A
Conceptual Structure of Causal Fault-Tolerance Algorithm
Critical Path Not Delayed
Causal Properties Keep Backups Consistent

Figure 6: Transparent fault-tolerance using causal process-groups.

1. Message m will be sent by P to P', and will contain sufficient *trace* information to enable P' to reproduce the execution of P up the this point. In the case where P is completely deterministic, m might be empty (in which case the action of sending it can be omitted). Otherwise, it would contain information about the order in which P received and processed requests, the order in which its threads were scheduled, and other sorts of information needed to resolve non-determinism in the execution. If desired, this message can also contain a complete checkpoint of the state of P, and indeed it may be desirable to periodically make such a checkpoint to ensure that recovery from failure will incur little delay.

2. Message r, which is causally ordered after m, contains the request that P is issuing to Q. P will send r using an atomic causal multicast to the group (Q, Q').

The reply from Q to P is treated in the same manner: the scheme is completely symmetrical with respect to clients and servers.

The trace message, m, from P to P' indicates the order in which P removed requests from its input queue because P may receive multiple concurrent requests, say from R and S. Although these messages will also be sent to P', unless the order of delivery is the same at P and P', P' will not know the order in which P processed them. This information can be omitted from the trace message if a totally ordered multicast is used for all requests. The same discussion applies to the trace message m' sent by Q to Q'.

To recover from the failure of P, process P' will, upon observing the failure event, reconstruct the state that P was in by loading the most recent checkpoint and simulating the computation performed by P. This may cause P to send duplicate messages to Q, which should detect and discard them (since P' behaves exactly the way that P behaved prior to failing, this can be done by numbering messages consecutively).

This scheme introduces two kinds of overhead not present in the original computation: extra messages (between P and P' and between Q and Q'), and delay along the critical path of the computation—when failure is rare, this would be the interactions between primaries.

The arguments made in Section 2.4, favoring asynchronous, causally ordered communication, apply here. By using causal communication throughout, there will *never be any need to delay a message along the critical path* (the transmission of r from P to Q) because of the messages sent to the backup processes (the transmission of m to P' and the copy of r sent to Q'; messages m' and the copy of r' sent to P' are not on the critical path). Given adequate background capacity to send these trace messages and remote (or "backup") messages, the fault-tolerant version of a computation might actually execute at the same speed as the original one! Moreover, although trace messages and messages to the backup processes do consume bandwidth, they can be delayed and sent in batches, thus pipelining communication and achieving higher efficiency. Although the details will depend on the protocol used, in many situations, the extra messages sent will not impact the performance of the application, provided of course that transmission of messages to backups does not cause congestion at the communication interface.

The stability property explained earlier is important, because it defines the limits of allowable asynchrony beyond which safety could be compromised. Specifically, if multicast a causally precedes multicast b and some process that receives b remains operational, a system that implements causal ordering must ensure that a is eventually delivered to all of its destinations (except those that fail). In our application, there is no real limit to the extent to which primaries can run "ahead" of the backups, except for the requirement that this safety condition be maintained.

If we represent each process pair (Q, Q') as a process group with two members, this example illustrates the need to preserve causality across the boundaries of process groups. To see this, consider when P sends a trace message m to P' and then sends some request r to (Q, Q'). Message m causally precedes r, but they are not sent in a single process group. But, if P now fails, we need to know that if r does get delivered, m will also be delivered. Thus, causality across group boundaries prevents a serious potential bug. The example also illustrates the need to communicate to a group from outside it; here, in fact, most communication is originated by external "clients" of the group. Finally, notice that the synchronization of group membership with respect to communication would be needed if one wished to create a new backup after failure of the primary.

Although we will not develop the details here, it is interesting to note that the the scheme described above is nearly identical to the Tandem process-pair implementation, with the exception of the asynchrony afforded by the causal ordering property. However, our description is more general; for example, it extends without modification to the case of k backups, while the Tandem work is very much tied to the assumption that $k = 1$.

Our scheme is also similar to that used in Targon/32 [BBG+89], but uses only a two-way causal multicast, rather than the three-way totally ordered multicast (**abcast**) they require. Using **abcast** rather than **cbcast** for transmission of requests, would eliminate many of the trace messages, but has the potentially serious disadvantage of delaying delivery of messages to the primary, introducing latency on the critical path but simplifying recovery after a failure. This argues in favor of **cbcast** for transmitting requests to a process pair.[8]

6 An implementation

Many of the foregoing observations and conclusions have been driven not just by usage of the Isis system, but by lessons learned from its implementation. So, while this paper is primarily about the semantics of group-based systems, it is clearly important that the methods we propose correspond to an efficiently implementable system architecture. In fact, our group is presently engaged on the design and implementation of a successor to Isis, called Horus, that will employ the experience gained from the initial system and the observations made above to achieve substantially increased flexibility and performance. Our basic approach is to separate Isis into two parts, one of which would be linked into the application address space, and one residing in the operating system. The operating system module can be made extremely spare, implementing a bare minimum of functionality: virtually synchronous process groups, causal domains, **cbcast** and **abcast** – the core functions identified in the discussion above. The remainder of the Isis model and the Toolkit itself would be realized at the library level. It would be beyond the scope of this paper, and somewhat premature, to discuss the design of Horus in greater detail. Completion of a prototype is expected in late 1992, at which time we plan to follow up on the present paper with one giving details and performance.

[8]Readers familiar with the algorithm in [BSS91] will realise that, under this algorithm, the approaches might actually have identical costs. The implementation of (causal) **abcast** in that paper uses a token holder to decide delivery ordering, and messages are never delayed at the token holder. If the primary member of a process pair is always used as token holder, as would be likely in an implementation of the approach under Isis, the flow of messages resulting from transmission of requests to the pair using **abcast** is the same as would result when using **cbcast** with a trace message that informs the backup of the order that was used.

7 Conclusions

Experience with real users can reshape one's perspective on a computer system. This has been the case with the Isis system, which entered into wide academic and commercial use with generally positive but sometimes surprising results. Our experiences support the belief that distributed systems should implement process groups at a basic level.

The mechanisms underlying this support need not be as exhaustive as in the present Isis system, which provides a bewildering variety of group membership and multicast ordering options to its users. Our understanding of the system and its users has now reached a point where we can argue that these be reduced to two mechanisms (atomic group membership and causal multicast) over which the virtually synchronous toolkit can be rebuilt.

Our paper makes two types of contributions. The first of these is at the level of group structures, particularly by refinement of the notion of group to address issues raised by having multiple groups, groups with external clients, and groups of groups. Our approach recognizes that clients are more numerous than servers, but that their communication patterns and use of group semantics are restricted, and it organizes groups into causal domains. We expect these styles of client-server groups to be durable because they are directly based on uses observed in practice. Although new group and multicast protocols are to be expected, these group structures should continue to present programmers with the interface they actually need.

Our second major contribution is the argument that *asynchronous* communication, combined with *failure atomicity* and *causal ordering*, is faster than synchronous request-response communication, and is sufficient for most communication needs. Although a total ordering is sometimes necessary, such ordering imposes unavoidable delays and should be implemented on top of a causal communication primitive.

Our new virtual synchrony architecture retains some of the complexity for which the original Isis system can be criticized. We believe that this is acceptable for two reasons. First, we see no way to further simplify the system without breaking important properties. Additionally, the elegance of the fault-tolerance transformation stands as evidence that the approach *does* result in simple solutions to important distributed computing problems. Thus, although the rationale of the architecture and the details of its implementation may continue to mystify non-experts, users of the system will find these concerns unimportant because it substantially simplifies their work. Just as the obscure details of register scheduling in an optimizing compiler or concurrency control in a database system do not prevent us from using these technologies, we believe that programmers of the next generation of distributed applications will leave the details of communication to the operating system – and will be far more productive for having done so.

8 Acknowledgements

The material presented here was arrived at through discussions with many others. We thank Micah Beck, Tushar Chandra, Rich Draves (CMU), Brad Glade, Keith Marzullo, Doug Orr (Chorus), Franklin Reynolds (OSF), Mark Rozier (Chorus), Fred Schneider, Pat Stephenson, Robbert Van Renesse, and Mark Wood. Our architecture was also influenced by the work of Franz Kaashok (Vrije), Paulo Veríssimo (INESC), and by the ANSA project. And we thank Maureen Robinson for producing the figures.

References

[AGHR89] François Armand, Michel Gien, Frédéric Herrmann, and Marc Rozier. Revolution 89 or Distributing UNIX brings it back to its original virtues. Technical Report CS/TR-89-36.1, Chorus systèmes, 6 Avenue Gustave Eiffel, F-78182, Saint-Quentin-en-Yvelines, France, August 1989.

[Bar81] Joel F. Bartlett. A NonStop kernel. In *Proceedings of the Eighth ACM Symposium on Operating Systems Principles*, pages 22–29, Pacific Grove, California, December 1981. ACM SIGOPS.

[BBG+89] A. Borg, W. Blau, W. Gretsch, F. Herrmann, and W. Oberle. Fault tolerance under Unix. *ACM Transactions on Computer Systems*, 7(1):1–23, February 1989.

[BC90] Kenneth Birman and Robert Cooper. The ISIS project: Real experience with a fault tolerant programming system. Technical Report TR90-1138, Cornell University Computer Science Department, Ithaca, NY, July 1990.

[Bir91] Kenneth Birman. The process-group approach to reliable distributed computing. Technical Report TR91-1216, Cornell University Computer Science Department, Ithaca, NY, July 1991. Submitted to Comm. ACM.

[BJ87] Kenneth P. Birman and Thomas A. Joseph. Reliable communication in the presence of failures. *ACM Transactions on Computer Systems*, 5(1):47–76, February 1987.

[BJ89] Ken Birman and Thomas Joseph. Exploiting replication in distributed systems. In Sape Mullender, editor, *Distributed Systems*, pages 319–368, New York, 1989. ACM Press, Addison-Wesley.

[BSS91] Ken Birman, Andre Schiper, and Pat Stephenson. Lightweight causal and atomic group multicast. *ACM Transactions on Computer Systems*, 9(3):272–314, August 1991.

[CASD85] Flaviu Cristian, Houtan Aghili, H. Ray Strong, and Danny Dolev. Atomic broadcast: From simple message diffusion to Byzantine agreement. In *Proceedings of the Fifteenth International Symposium on Fault-Tolerant Computing*, pages 200–206, Ann Arbor, Michigan, June 1985. Institution of Electrical and Electronic Engineers. A revised version appears as IBM Technical Report RJ5244.

[CZ85] David Cheriton and Willy Zwaenepoel. Distributed process groups in the V kernel. *ACM Transactions on Computer Systems*, 3(2):77–107, May 1985.

[DC90] Stephen E. Deering and David R. Cheriton. Multicast routing in datagram internetworks and extended LANs. *ACM Transactions on Computer Systems*, 8(2):85–110, May 1990.

[GT90] Ajei Gopal and Sam Toueg. On the specification of broadcast. In *Proceedings of the Second IEEE International Workshop on Future Trends of Distributed Computing Systems*, pages 54–56, Cairo, Egypt, October 1990. IEEE Computer Society.

[JZ87] David B. Johnson and Willy Zwaenepoel. Sender-based message logging. In The Seventeenth Annual International Symposium on Fault-Tolerant Computing: Digest of Papers, pages 14–19. Institution of Electrical and Electronic Engineers, June 1987.

[KT91] M.F. Kaashoek and A.S. Tanenbaum. Group communication in the amoeba distributed operating system. In *Proc. The 11th Internatinal Conference on Distributed Computer Systems.*, pages 222–230, Arlington, VA, May 1991. Institution of Electrical and Electronic Engineers.

[KTHB89] M. Frans Kaashoek, Andrew S. Tanenbaum, Susan Flynn Hummel, and Henri E. Bal. An efficient reliable broadcast protocol. *Operating Systems Review*, 23(4):5–19, October 1989.

[Lam78] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, July 1978.

[LLS90] Rivka Ladin, Barbara Liskov, and Liuba Shrira. Lazy replication: Exploting the semantics of distributed services. In *Proceedings of the Tenth ACM Symposium on Principles of Distributed Computing*, pages 43–58, Qeubec City, Quebec, August 1990. ACM SIGOPS-SIGACT.

[LS83] Barbara Liskov and R. Scheifler. Guardians and actions: Linguistic support for robust, distributed programs. *ACM Transactions on Programming Languages and Systems*, 5(3):381–404, July 1983.

[OSS80] John Ousterhout, D. A. Scelza, and P. S. Sindhu. Medusa: an experiment in distributed operating structure. *Communications of the ACM*, 23(2):92–105, February 1980.

[PBS89] Larry L. Peterson, Nick C. Bucholz, and Richard Schlichting. Preserving and using context information in interprocess communication. *ACM Transactions on Computer Systems*, 7(3):217–246, August 1989.

[Ras86] R. F. Rashid. Threads of a new system. *Unix Review*, 4:37–49, August 1986.

[Rep90] John H. Reppy. *Concurrent Programming with Events—The Concurrent ML Manual (version 0.9)*. Department of Computer Science, Cornell University, Upson Hall, Ithaca, NY 14853, November 1990.

[Sch88] Frank Schmuck. *The use of Efficient Broadcast Primitives in Asynchronous Distributed Systems*. PhD thesis, Cornell University, 1988.

[Spe85] Alfred Spector. Distributed transactions for reliable systems. In *Proceedings of the Tenth ACM Symposium on Operating Systems Principles*, pages 127–146, Orcas Island, Washington, December 1985. ACM SIGOPS.

[SY85] Ray Strom and S. Yemeni. Optimistic recovery in distributed systems. *ACM Transactions on Computer Systems*, 3(3):204–226, April 1985.