Programming with Process Groups: Group and Multicast Semantics*

Kenneth P. Birman   Robert Cooper   Barry Gleeson

TR-91-1185
January 29, 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. **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*. 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,BSS90]. It is the semantics of these core group and multicast mechanisms that this paper explores. Process groups are a natural abstraction and have been

*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.

used in a number of distributed systems [CZ85,OSS80,KTHB89,LLS90,PBS89,AGHR89]. However, the precise characteristics of group facilities differ widely among the systems that use groups, 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.

Our analysis draws on experience with the **Isis** system, which has been distributed to more than 750 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 into the successful aspects of the technology, but also into aspects that need 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. For example, group members are able to react to external events in a coordinated way, using the synchronized group membership lists, without the need for 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 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, which predominate among current **Isis** uses, an *object* is typically an abstract data type with small state[1] that may change rapidly.

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

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.

- *Groups used for transparent fault-tolerance.* Here the components of a distributed system are systematically replaced by fault-tolerant process groups.

The numbers and uses of groups differ substantially from the original expectations dating from when **Isis** was developed. We have been forced to question many of the basic assumptions underlying the initial architecture, and to ask how the system might 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?
- 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?

2 Process groups

This section refines our terminology and confronts the first of the design questions with which the paper is concerned. In Sec. 2.1 we consider the semantics of process group membership; group communication is discussed in Sec. 2.2.

2.1 Group membership

A *process group* is a collection of *communication endpoints* that can be referenced as a single entity. A communication endpoint would be referred to by a socket address in Unix, a send-right in Mach, a 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 identify end-points with *processes*. However, although the current **Isis** implementation permits only one end-point per process, this restriction is removed in our new architecture.

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.

- *Generality.* In **Isis** applications, process groups are obviously a basic and heavily used programming construct. Assuming that a single, general mechanism can support such a diverse user community (without becoming encumbered by numerous special features), system-level support for that mechanism is justifiable.
- *Performance.* One could attempt to avoid the cost of maintaining explicit group membership lists at all times by re-computing group membership only when needed, perhaps using a caching and validation mechanism. However, if there is more than one multicast or group membership query per group change (as is usually the case), such an approach will increase costs.[2] Thus it is cheaper to maintain accurate group membership information.
- *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.

[2] Actually, we are familiar with some applications in which changes to group membership are more frequent than communication. 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. Even so, our experience suggests that these ad-hoc groups tend to be subsets of more stable enclosing groups. Useful communication structures may therefore be present, even when they are not immediate from the problem statement.

We conclude that a system-level group facility is needed, and that knowledge of group membership will be important at the application level.

2.2 Group communication

A *group multicast* occurs when a thread sends a message to a process group. Various styles of multicast are possible: asynchronous, all-reply, one-reply, *k*-reply, and so forth. **Isis** programmers find all of these styles useful. Multiple threads may initiate multicasts to the same group concurrently, hence multicast communication primitives often provide system-enforced ordering properties. Real-time systems may support message priorities or multicast delivery deadlines: messages sent between correct processors are guaranteed to be scheduled according to priority and/or to arrive within some limited delay.[3] Other potentially useful properties include *failure atomicity*, namely all-or-nothing delivery guarantees even if processes or processors fail during a multicast, and *membership atomicity*, the guarantee that group membership changes are totally ordered and synchronized with group communication. 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 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 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.

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. 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 (raising difficult fault-tolerance problems if the “agent” fails). Here, we assume that process groups will have both members and clients. In **Isis** clients identify themselves through the **pg_client** system call.

How should systems support group communication?

One might also ask why explicit system support for group communication is necessary. Most of the arguments for system management of group membership also apply here and we do not repeat them.

[3] Protocols that enforce deadlines often impose stringent timing requirements upon the operating system. Moreover, they frequently obtain determinism by introducing costly 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.

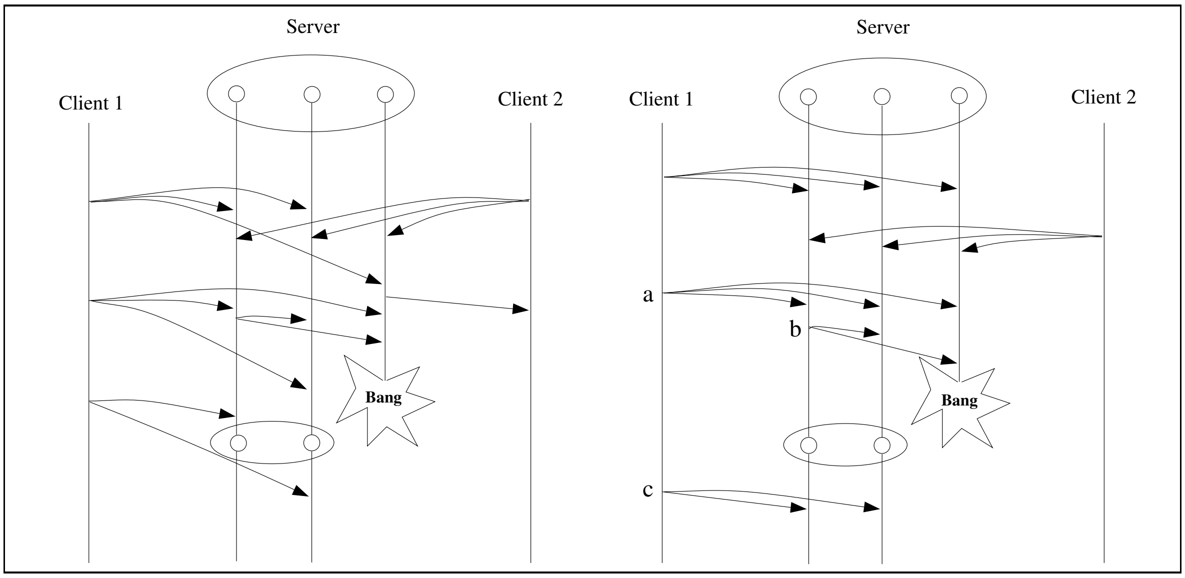

Figure 1: (a) Unordered group communication; (b) Synchronous group communication.

However, a frequently-asked question is 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. Moreover, many styles of group communication are essentially extensions of RPC, and many of the techniques used to support RPC carry over to group multicast protocols.

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 be made atomic, with parallel threads 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.

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,

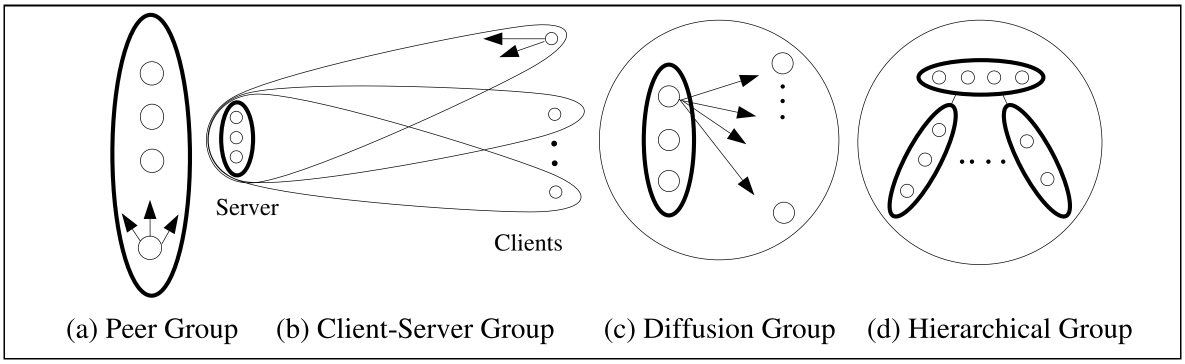

Figure 2: Common group structures

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.

We conclude that group communication should be supported directly by the system, and implemented over asynchronous message passing or transport-level multicast.

2.3 Common styles of group usage.

From experience with **Isis** users, we have identified four group structures that continually reappear in **Isis** programs (Fig. 2). Each responds to a different programming need.

Groups structures

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

For brevity, we omit detailed discussion of two degenerate cases: 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.

Multiple 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. Most communication becomes a three-way multicast: to the backup of the sender and the primary/backup pair comprising the destination. 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.

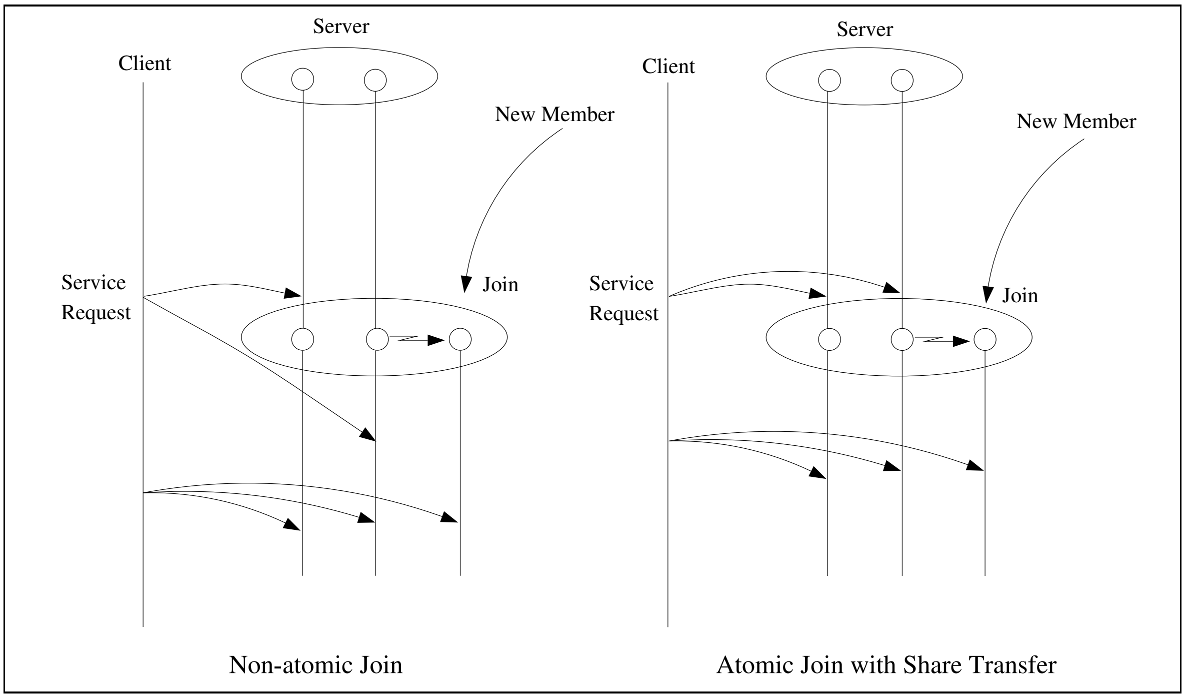

Figure 3: (a) A non-atomic join (b) Atomic join.

3 Design choices for group and multicast primitives

The goal of this section is to explore 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 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, like 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.

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.

Earlier, it was observed that 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**, atomic membership changes and atomic address expansion are both needed.

3.2 Causal and total multicast orderings

Multicast ordering raises a number of subtleties. 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 *partial* 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 abcast

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 order 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 **Isis**, **cbcast** is at least a factor of two faster than **abcast**, and even more so if an operating system context switch occurs when the **abcast** blocks.

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. 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** [BSS90].[4]

In discussing the option of building multicast over RPC, we stressed the need for asynchronous communication. 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.

[4]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**. 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 never seen an **Isis** user who actually needed **gbcast** at the application level. We now understand that the real need of application programmers is 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.

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

The use of asynchronous communication raises a 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.

To summarize:

- Asynchronous operations are a key to 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 a common file system I/O problem, and can be addressed by supporting communication system calls that have natural file system counterparts.

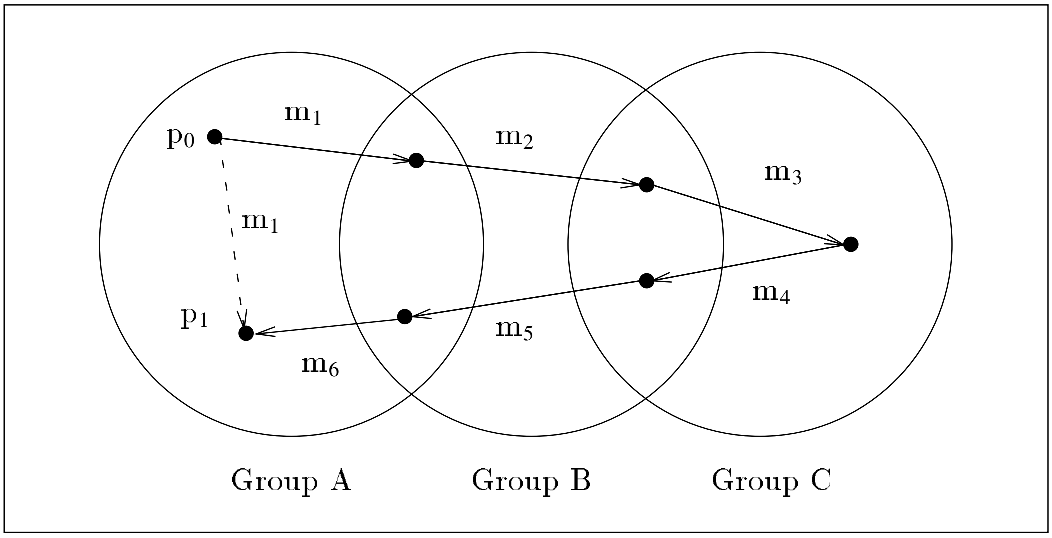

Figure 4: A causal chain spanning multiple groups

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.

Should causality be preserved between groups?

Cbcast ensures 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 happens in the example in Fig. 4, where the messages are causally ordered m_1, m_2, and so on through m_6. Message m_1 should be delivered before m_6 at process p_1. Here, the conflict arises within a *single* group, between the original operation and 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 strongly argues that causality should be preserved here. Moreover, since the user of an object may be unaware of its implementation, this guarantee should be completely automatic, requiring no overt action by the programmer.

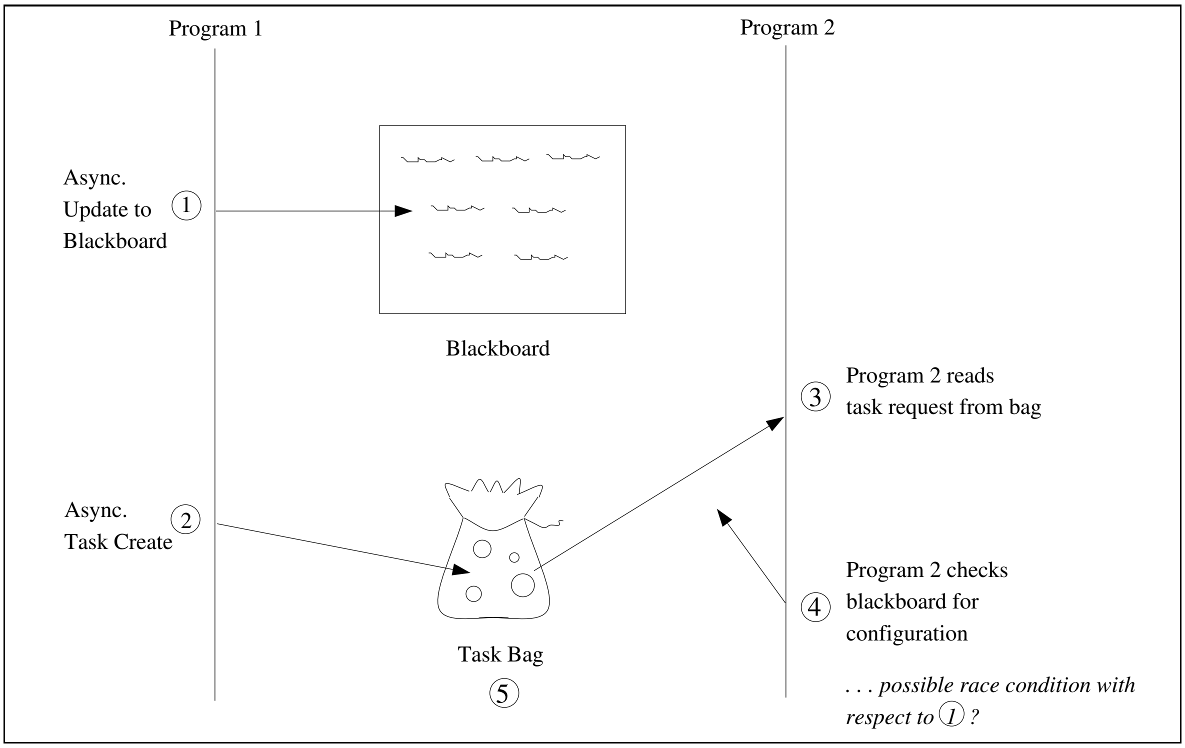

Figure 5: Application using a blackboard and a task-bag.

Should causality *always* be preserved between groups?

Consider a single program built of multiple independent subsystems. Any 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 from one another, and in some settings (e.g. when an application combines several subsystems that run at different priorities), the delays introduced by the need to enforce inter-group causality would be inappropriate.

This motivates a notion of *causality domains* which partition the groups in a system. Causality is observed only between groups in the same domain. A causality domain resembles a Psync *session* [PBS89], but contains many overlapping process groups. Following our policy of "safe" behavior for less-experienced programmers, groups which are not explicitly placed into a causality domain reside in a common, default domain. In Section 5, we propose a simple interface for defining these domains; the assumption is that typical users would simply accept the default, while real-time programmers and developers of special tools, like debugging aids, would use the mechanism to avoid undesired interference with the underlying application.

As an example, consider a user who employs a blackboard object and a task-queue object in a graphics application, both implemented to use asynchronous updates (see Figure 5). A typical execution sequence might involve 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 object for control parameters. If these two objects are placed in the same causality domain, all of the updates may be done asynchronously, without worrying about a race whereby the blackboard update might not have arrived when the computation service looks for its parameters. On the other hand, if the same application contains a monitoring or debugging operation, such as periodically reporting the length of the task queue, one would place the monitoring group in a different causality domain. By doing this, access to the instrumentation mechanism would never be delayed by activity in the base system. Moreover, the act of monitoring would not introduce new causal paths in the application, which might affect the behavior being monitored.

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 2.2) consisting of the same set of server processes, and intersecting sets of clients. One can imagine applications in which **abcast**s from the servers should be ordered totally at the overlapping client sets.

For a more 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 **abcast**s 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 **abcast**s, 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.

Implementation issues

The **Isis** system has used two quite different implementations of both **cbcast** and **abcast**. The most recent protocol suite is presented by Birman, Stephenson and Schiper [BSS90,Ste91]. Moreover, a number of other protocols exist that could be used (perhaps with modifications) in support of the abstractions proposed here. Our current preference is primarily based on engineering considerations.[5]

A related issue concerns the extent to which causality information should be hidden from users. **Isis** currently uses a scheme in which causality data is managed by the system: this simplifies the user interface, and (because causality crosses group boundaries) protects the integrity of a service from programming errors in its clients. Other researchers, such as Peterson [PBS89] and Ladin [LLS90], have proposed schemes in which users play a more direct role in maintaining, transmitting and reasoning about this information. Such approaches allow a sophisticated user—or a clever compiler—to exploit application semantics inaccessible

[5] Compared with the previous **Isis** protocol [BJ87], this new protocol suite is better suited to direct implementation in the user's address space because it does not rely on a piggybacking scheme. Piggybacking of messages intended for some process *a* into the address space of process *b* raises troubling overhead and data integrity issues; our current scheme largely avoids these. Further, in the new scheme, the work done by a process is primarily related to the number of multicasts sent and received by that process; the prior **Isis** protocol had several sorts of non-local overhead. However, none of these is a compelling argument, and there may be situations where the prior protocols, or some other solution, would be preferable.

to the runtime subsystem. Our approach, although simpler, may reduce concurrency by enforcing spurious causal orderings.

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 of these issues in the context of a distributed variant of Concurrent ML [Rep90].

5 A process group architecture

In this section, we sketch an implementation architecture on the basis of the semantics decisions reached in the preceding sections.

As seen in Fig. 6, our system architecture has three layers. The uppermost layer is a user-space library. This implements support for the user's application, and might include the **Isis** toolkit interface, a process pair implementation for transparent fault-tolerance, database transaction support, or other high-level mechanisms. An intermediate layer implements the process group mechanism, providing group multicast, membership and failure atomicity, and causality domains. This layer is intended to be implemented as a separate module that interacts with higher and lower layers through fast inter-address-space calls or local RPC. The resulting module could then be moved into the operating system if desired. Over Unix, it would probably reside in a shared library within the user's address space. The lowest layer supports a weaker notion of groups and multicast and is concerned primarily with network topology and the use of hardware multicast transport protocols.

The system-call interface for the intermediate and lower layers of the system is shown in Tables 1 and 2. These interfaces are stackable: applications that do not need virtually synchronous communication can bypass the upper-layer. Notable here is what has been omitted. Mechanisms such as the causal **abcast** and the collection of replies to multicasts are left to tools executing in the user address space. The same is true of group membership operations, such as joining a group or becoming a client.

Also omitted from the core layers are mechanisms for detecting process and machine failures, and for implementing a group namespace. These were significant sources of complexity in the initial **Isis** implementation; in our new system, they will be provided by user-space utility programs. This will make it easier to exploit pre-existing operating system or hardware services, such as hardware failure notification mechanisms. However, our scheme does place make certain assumptions about the failure detection module, as discussed by Ricciardi and Birman [RB90].

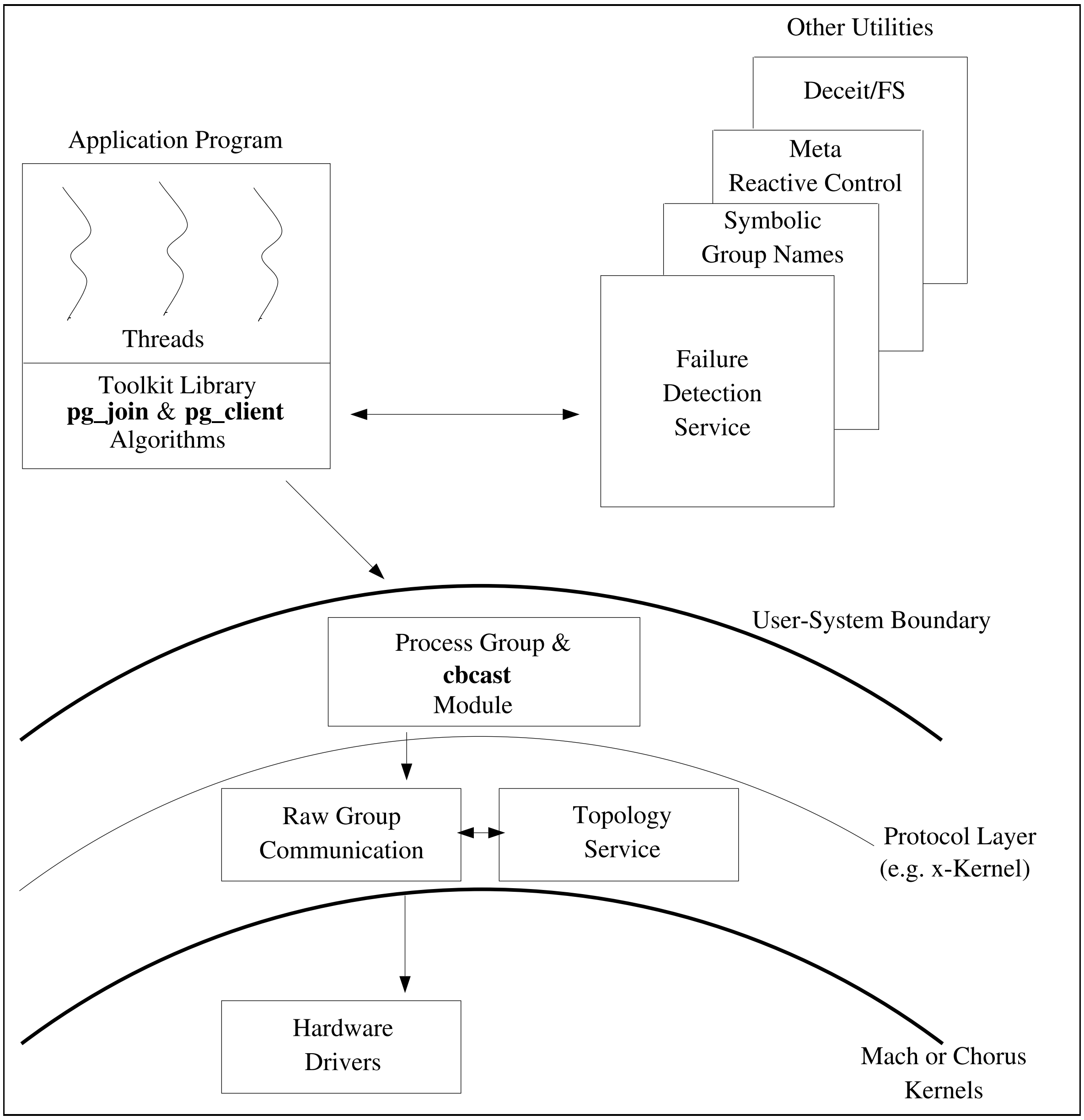

Figure 6: Proposed system architecture

Virtually synchronous process group interface	
gid = pg_create(view)	create group with initial view
pg_add(gid, pid, type)	add process ID to group member/client list
pg_del(gid, pid, type)	delete pid from member/client list
pg_monitor(gid, proc)	proc is called on group view changes
view = pg_getview(gid)	returns the current group view
cbid = cbcast(gid, msg)	message is sent to members of group gid
pg_entry(gid, proc)	proc is called when a message is delivered to group gid
pg_flush(gid)	flush outstanding messages in this group
old_id = set_dflt_domain(did)	sets default causality domain for calling thread

Table 1: Interface to proposed process group module

Group implementation interface to lower layers of system	
phys_gid = phys_create(view)	create physical group giving initial list of members
phys_add(phys_gid, pid)	add member to group
phys_del(phys_gid, pid)	delete member of group
phys_monitor(phys_gid, proc)	proc is called on member failures
phys_mcast(phys_gid, msg)	reliable multicast of msg to group phys_gid
phys_entry(phys_gid, proc)	proc is called when a message is received by group phys_gid

Table 2: Lower-level system interface

In our new scheme, operations that change the *group view* (membership or client list) are implemented using **cbcast** to inform group members of the new view. Only members need an accurate copy of this list; clients cache estimates of the membership, refreshing stale views as needed (this resembles the iterated **cbcast** algorithm [BJ87]). A distinguished member of each group initiates these calls, ensuring that only one such operation is done at a time. The effect of all this is that the group implementation module contains little more than an implementation of the causal multicast protocol [BSS90] and a mechanism for storing group views. Details of these and other higher-level algorithms appear Stephenson's thesis [Ste91].

The layer below the process groups module is lacking in current operating systems. It includes a *network topology* service which maps communication end-points (process addresses in the **Isis** implementation) to transport protocol addresses, knows the devices of which the network is composed (subnetworks, bridges, multicast ability), and knows the transport protocols that are available. Multicast operations are converted into calls to multicast and point-to-point interfaces either on raw device drivers or protocol stacks such as UDP/IP and the OSI stack. The *x*-Kernel supports this kind of dynamic layering of protocols and may form the basis for the **Isis** transport layer.

Both the atomic and raw group mechanisms include monitoring facilities. At the process group level, a callback is performed to the user-layer each time group membership changes. The user-process may also query the "current" membership of a group (in a logical or virtually-synchronous sense). The *group view* returned will list members at the time of the last group-related event received by the process.

Monitoring at the physical layer has a simpler interface and semantics. This layer retries transmissions until a destination is deemed faulty, at which point it first calls the **phys_monitor** callback routine and then reports that the message delivery failed. The **phys_mcast** routine returns only when copies of the message are known to have been delivered to all non-faulty members in the physical destination list.

The above sketch of the architecture omits a tremendous amount of detail. We have developed many of the protocols and algorithms needed to convince us that the scheme can be made to work well, and will be including these in a forthcoming specification document. We are also exploring the implementation aspects jointly with members of the Amoeba, Chorus and Mach groups. Construction of the new system will begin in early 1991.

6 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 *client*. Our approach recognizes that clients are more numerous that servers, but that their communication patterns and use of group semantics are restricted. 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 a sufficient solution to 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.

7 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.

[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.

[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.

[BSS90] Ken Birman, Andre Schiper, and Pat Stephenson. Fast causal multicast. Technical Report TR90-1105, Cornell University Computer Science Department, Ithaca, NY, April 1990. Submitted to ACM Transactions on Computer Systems.

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

[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.

[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.

[RB90] Aleta Ricciardi and Ken Birman. A formalism for fault-tolerant applications in asynchronous systems. In *Fourth SIGOPS European Workshop*, September 1990.

[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.

[Ste91] Pat Stephenson. *Fast Causal Multicast*. PhD thesis, Cornell University, January 1991. To appear.