The Process Group Approach to Reliable Distributed Computing

Kenneth P. Birman*

TR 91-1216
July 1991

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

*The author is in the Department of Computer Science, Cornell University, and was supported under DARPA/NASA grant NAG-2-593, and by grants from IBM, Hewlett Packard, Siemens, GTE and Hitachi.

The Process Group Approach to Reliable Distributed Computing *

Kenneth P. Birman

July 3, 1991

Abstract

The difficulty of developing reliable distributed software is an impediment to applying distributed computing technology in many settings. Experience with the ISIS system suggests that a structured approach based on *virtually synchronous process groups* yields systems which are substantially easier to develop, fault-tolerant, and self-managing. This paper reviews six years of research on ISIS, describing the model, the types of applications to which ISIS has been applied, and some of the reasoning that underlies a recent effort to redesign and reimplement ISIS as a much smaller, lightweight system.

1 Introduction

As distributed computing systems have become prevalent, the development of reliable distributed software has emerged as a major challenge. Even in non-distributed systems, reliability is a complex property, spanning issues such as correctness, fault-tolerance, self-management, real-time responsiveness, protection and security. Distributed systems take these issues further: a distributed system consists of multiple processes that must cooperate, hence one must be concerned not just with the behavior of individual components, but also with their joint behavior in the context of the overall application.

One might expect confidence in the correctness of a distributed system to follow easily from the correctness of its constituents, but this is not always the case. The mechanisms used to structure a distributed system and to implement communication between components play a vital role in determining how a system will behave. We argue that contemporary distributed operating systems have placed excessive emphasis on communication performance, overlooking the need for tools to support the development of complex systems. Further, communication primitives often give generally reliable behavior, but exhibit weak or ill-defined semantics when uncommon events such as failures or system configuration changes occur. The

*The author is in the Department of Computer Science, Cornell University, and was supported under DARPA/NASA grant NAG-2-593, and by grants from IBM, HP, Siemens, GTE and Hitachi.

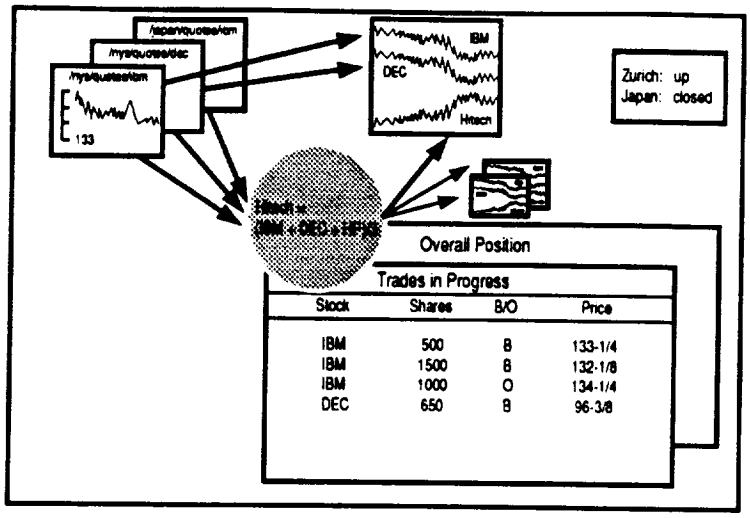

Figure 1: Broker's trading system

resulting building blocks are useful in developing fast distributed software, but unsuitable where reliability is important.

This paper reviews six years of research on the ISIS system, which provides a tools in support of reliable distributed computing. The basic idea is that the development of reliable distributed software can be simplified using *process groups* and *group programming tools*. Our goal in this paper is not to present new material, but rather to motivate the approach taken in ISIS, to survey the system, and to discuss experience with some real applications.

It will be helpful to consider these issues in a more concrete context. Contemporary brokerage and trading systems are highly distributed. It is not uncommon for brokers to cooperate by coordinating trading activities across multiple markets. Trading strategies rely heavily on accurate pricing and market volatility data, dynamically changing databases giving the firm's holdings in various equities, news and analysis data, and elaborate financial and economic models based on relationships between the prices of sets of financial instruments. A distributed system in support of this application must serve multiple communities: the firm as a whole, where reliability and security are key considerations; the brokers, who depend on speed and the ability to customize the trading environment; and the system administrators, who seek uniformity, ease of monitoring and control. Notice that these goals compete: support for customization of the interface increases the flexibility of the system, but could make it harder to administer and less reliable. A theme of the paper will be that one overcomes this intrinsic problem by standardizing the methods used to "glue the system together", and by endowing the corresponding mechanisms with predictable, fault-tolerant behavior.

Figure 1 shows a possible interface to a trading system. The display is centered around the current position of the account being traded, showing purchases and sales as they occur. A broker typically authorizes

purchases or sales of shares in a stock, specifying limits on the price and the number of shares. These instructions are communicated to the trading floor, where agents of the brokerage or bank trade as many shares as possible, remaining within this authorized window. In this discussion, we don't consider the issues raised by verification of the limits, although in many systems one would need to confirm that the account has adequate funds to cover the trade and that the broker has authorization to trade the account.

The display in the figure is composed of multiple small *widgets*, of the sort one might find in a general purpose graphical toolkit. Each should be thought of as having some set of input ports, which the broker binds to information sources, and some number of output ports, which can be used as inputs to other widgets. For example, the broker has introduced an analysis relevant to the stocks being traded (shaded circle), named its output, and used it as input to the central graph. The figure names these communication channels using standard file-system pathnames. However, communication channels would not be treated like files: programs that monitor a channel must react rapidly to each new event that occurs, and it would not be useful to store the detailed pricing of a stock on an event-by-event basis.

A stock like IBM may be traded in multiple exchanges, such as New York and Tokyo. When this occurs, a broker will potentially need simultaneous access to multiple markets. Although such a broker would prefer to treat the trading system as a seamless whole, the physical architecture of any large system is hierarchical, consisting of local area networks interconnected by wide-area communication lines. These will have very different performance characteristics than local-area lines, and will generally be less reliable. Thus, a Wall Street broker who monitors the market in Japan and uses this to control trades in Zurich depends upon a sophisticated distributed communication infrastructure. With regard to reliability, the broker will need assurances that all the programs which should see a piece of information will do so, and that if an error arises, the system will automatically correct it (or will notify the broker). In a wide-area trading application, large numbers of system components (both hardware and software) will be involved in solving these problems, and many would need to be monitored and restarted automatically if a failure occurs.

The central display of Figure 1 illustrates a further point. As noted earlier, the trader has plotted a computed index of technology stocks against the price of IBM. It is important that brokers and bankers be able to introduce these sorts of analysis services without engaging in sophisticated programming. It should be possible to share the output of such services with colleagues – whether down the hall or in Zurich. The system must be flexible enough to accommodate the introduction or modification of services at runtime, and still maintain its reliability guarantees.

The reliability of introduced services may be as critical as that of the base services built into the overall system. In Figure 2, the computational widget is "shadowed" by additional copies, to indicate that it has been made fault-tolerant (i.e. it would remain available even if the broker's workstation failed). A broker is unlikely to be a sophisticated programmer, so fault-tolerance would have to be introduced by the system itself. This involves replicating or checkpointing the basic computation, placing the replicas on processors

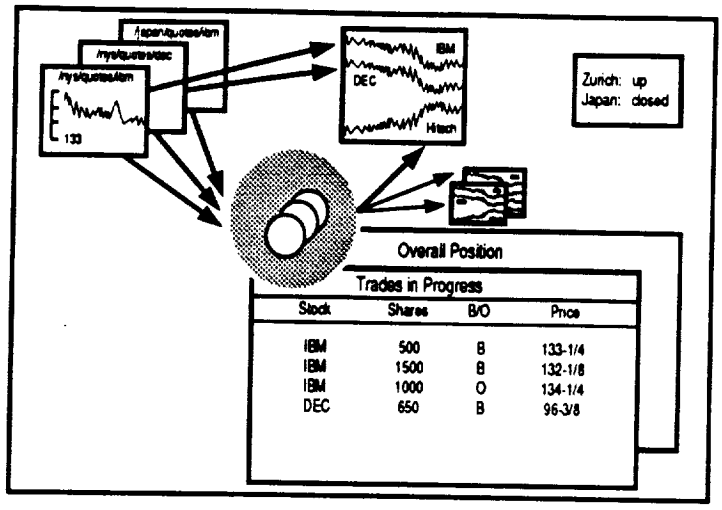

Figure 2: Making an analytic service fault-tolerant

that fail independently from the broker's workstation, and automatically activating a backup if the primary fails.

The requirements seen above are common in modern trading environments. However, they are not unique to the application. It is easy to rephrase this example in terms of the issues confronted by a team of seismologists cooperating to interpret the results of a seismic survey, a doctor reviewing the status of patients in a hospital from a workstation at home, a design group collaborating to develop a new product, or application programs cooperating in a factory-floor process control setting. To build applications for the networked environments of the future, a technology is needed that will make it possible to solve these sorts of distributed computing problems as easily as we build graphical interfaces today.

A central premise of the ISIS project, shared with several other efforts [LL86,CD90,Pet87,KTHB89], is that support for programming with *groups of cooperating programs* is the key to solving problems such as the ones seen above. For example, underlying a fault-tolerant data analysis service will be a group of programs that jointly provide continuous service, adapting transparently to failures and recoveries. The publication/subscription style of interaction involves an *implicit* use of process groups: here, the group has a dynamically varying set of publishers and subscribers. Although the processes publishing to or subscribing to a topic do not cooperate explicitly, the reliability of the overall application may well depend on the reliability of group communication. It is easy to see how problems could arise if two brokers monitoring the same stock saw different pricing information, or if a doctor and nurse were presented with inconsistent patient status information.

Process groups of various kinds arise throughout the broker's application: in fault-tolerant management of the resources available in the network, distributing quotes and other forms of system data, detecting

failures and reconfiguring to maintain availability, building the directory servers needed to track down other servers, and replicating databases containing trading and system status information. Yet, outside of a small set of research systems, current distributed computing environments have provided little support for group communication patterns and programming. These issues have been left to the application programmer, and application programmers have been largely unable to respond to the challenge. As a consequence, contemporary distributed computing environments have prevented users from realizing the potential of the distributed computing infrastructure on which their applications run.

The remainder of the paper is organized into three parts. The first focuses on group programming, defining the problems that need to be solved more carefully and discussing the algorithmic issues underlying their solutions. This leads into the ISIS computational model, called *virtual synchrony*. The next part of the paper discusses how ISIS is presented to users: not the specific interfaces, but the basic tools from which ISIS users construct applications. The last part reviews some of the applications that have been built over ISIS. The paper concludes with a brief discussion of future directions for the project.

2 Process groups

Two styles of group usage are seen in the broker's application, and in most ISIS applications:

- *Anonymous groups*: Anonymous groups arise when an application publishes data under some "topic," to which other processes subscribe. For an application to operate automatically and reliably, anonymous groups should provide certain properties. Informally,

  1. It should be possible to send messages to the group using a *group address*. The high-level programmer should not be involved in expanding the group address into a list of destinations.
  2. Messages should be delivered exactly once. The application programmer should not need to worry about message loss or duplication.
  3. Messages should be delivered in order. As we will see below, there are several ways to interpret this requirement. At a minimum, one would expect that messages be delivered in the order they were published.
  4. It should be possible to maintain the history events seen by the group. This is subtle, because programs might join the group when it has been operational for a while, and a new subscriber will often need accurate historical background. If n messages are posted and the first message seen by the subscriber is message m_i, one would expect messages $m_1 ... m_{i-1}$ to be in the history and messages $m_i ... m_n$ to all be delivered to the new process.

Of course, one can imagine applications that wouldn't require all of these properties, but each is important in many settings.

- *Coordinated action by sets of programs*: Sets of programs might cooperate explicitly for a number of reasons: fault-tolerance, load sharing, parallel database searches, replication of data, sharing a secret, and so forth. The resulting *explicit groups* share the communications requirements of an anonymous group, but have additional needs stemming from the use of group membership information in the application. For example, a fault-tolerant service might have a primary member that takes some action and an ordered set of backups that take over, one by one, if the current primary fails. Here, membership changes (failure of the primary) trigger actions by group members. Unless the same changes are seen in the same order by all members, situations could arise in which there are no primaries, or several. Similarly, a parallel database search might be done by dividing the database into n parts, where n is the number of group members; each member would do $1/n'th$ of the work. The members need consistent views of the group membership to perform such a search correctly.

At a glance, one might think that global correctness of a distributed system would follow from the local correctness of its components. That is, given a process group composed of correct processes, one might expect the service implemented by the group to also be correct. However, this overlooks the need to synchronize the actions taken by the group members. If the group maintains replicated data, partitions a database using dynamically varying criteria, or uses the composition of the group as an input to the algorithm executed by the components, it will be necessary to synchronize events that change these attributes of the group.

One thus sees that a number of more more technical problems must be considered in developing software based on distributed process groups:

- *Support for group communication*, including addressing, failure atomicity, and message delivery ordering.
- *Use of group membership as an input.* It should be possible to use the group membership or changes in membership as input to a distributed algorithm (one run concurrently by multiple group members).
- *Synchronization.* To obtain globally correct behavior from group applications, it is necessary to synchronize the order in which actions are taken, particularly when group members will act independently on the basis of dynamically changing, shared information.

3 Building distributed services using message passing

This section explores solutions to these problems that use the sorts of tools provided by conventional (non-transactional) distributed operating systems.

3.1 Conventional message passing technologies

Most contemporary operating systems offer three classes of communication services [Tan88]:

- *Unreliable datagrams*: These facilities automatically discard corrupted messages, but do little additional processing. As seen by a user, most messages will get through, but under some conditions messages might be lost in transmission, duplicated, or delivered out of order.
- *Remote procedure call*: In this approach, communication is presented as a procedure invocation that returns a result. RPC is a relatively reliable service, but when a failure does occur, the sender is unable to distinguish between four possible cases: the destination may have failed before or after receiving the request, or the network may have prevented or delayed delivery of the request or the reply.
- *Reliable data streams*: Here, communication is performed over channels that provide flow control and reliable, sequenced message delivery. Because of pipelining, data streams outperform RPC when an application sends large volumes of data. However, if a stream breaks, the situation is the same as for a failed RPC.

3.2 Building groups over conventional technologies

How might one solve the group communication problems identified in Sec. 2 using these sorts of technologies? Obviously, this is possible: after all, ISIS does so, and many distributed systems solve at least some of them. However, it is not straightforward.

Group addressing

Consider first the problem of mapping a group address to a membership list, in an application where the membership could change dynamically due to processes joining the group or leaving. The obvious way to approach this problem involves a *membership service*. Such a service maintains a map from group names to membership lists. Ignoring server fault-tolerance issues, one could implement this with a simple program that supports remotely callable procedures to register a new group or group member, obtain the membership

of a group, and perhaps to forward a message to the group. A process could then transmit a message either by forwarding it via the naming service, or by looking up the membership information, caching it, and transmitting messages directly. In the latter case, one would also need a mechanism for invalidating cached addressing information when the group membership changes (this is not a trivial problem, but the need for brevity precludes discussing it in detail). The first approach will perform better for one-time interactions; the second would be preferable in an application that sends a stream of messages to the group.

Message delivery ordering

Although either approach to the group addressing problem would get messages to the members of a group, important issues have been overlooked. Several such issues concern the order in which messages are delivered.

Consider Figure 3-a. Messages m_1 and m_2 are sent concurrently and happen to be seen in different orders by servers s_1 and s_3. In many applications, s_1 and s_3 would behave in an uncoordinated or *inconsistent* manner if this occurred. For example, one program might see the market volatility fall and then rise, while another sees it rise and then fall. Market volatility is a parameter to many financial computations, and such a sequence could easily leave the servers in inconsistent states.

A designer would have to anticipate possible inconsistent message ordering, and either design the application to tolerate such mixups, or explicitly prevent them from occurring, perhaps by delaying the processing of m_1 and m_3 within the server until an ordering has been established. The real danger is that the designer will overlook the whole issue – after all, two simultaneous messages to the server that arrive in different orders may seem like an improbable scenario – yielding an application that usually is correct, but may exhibit abnormal behavior under periods of particularly heavy load.

Unfortunately, this is only one of several delivery ordering problems illustrated in the figure. Consider the situation when s_3 receives message m_3. Message m_3 was sent by s_1 after receiving m_1, and might even refer to or depend upon m_1. For example, m_1 might authorize a certain broker to trade a particular IBM account, and m_3 could be a trade that the broker has initiated on behalf of that account. Our execution is such that s_3 has not yet received m_1 when m_3 is delivered. Perhaps m_1 was discarded by the operating system due to a lack of buffering space. It will be retransmitted, but only after a brief delay during which m_3 might be received.

Why might this matter? Imagine that s_3 is displaying buy/sell orders on the trading floor. s_3 will consider m_3 invalid, since it will not be able to confirm that the trade was authorized. An application with this problem might fail to carry out valid trading requests. Again, although the problem is solvable, the question is whether the application designer will have anticipated the problem and programmed a correct mechanism

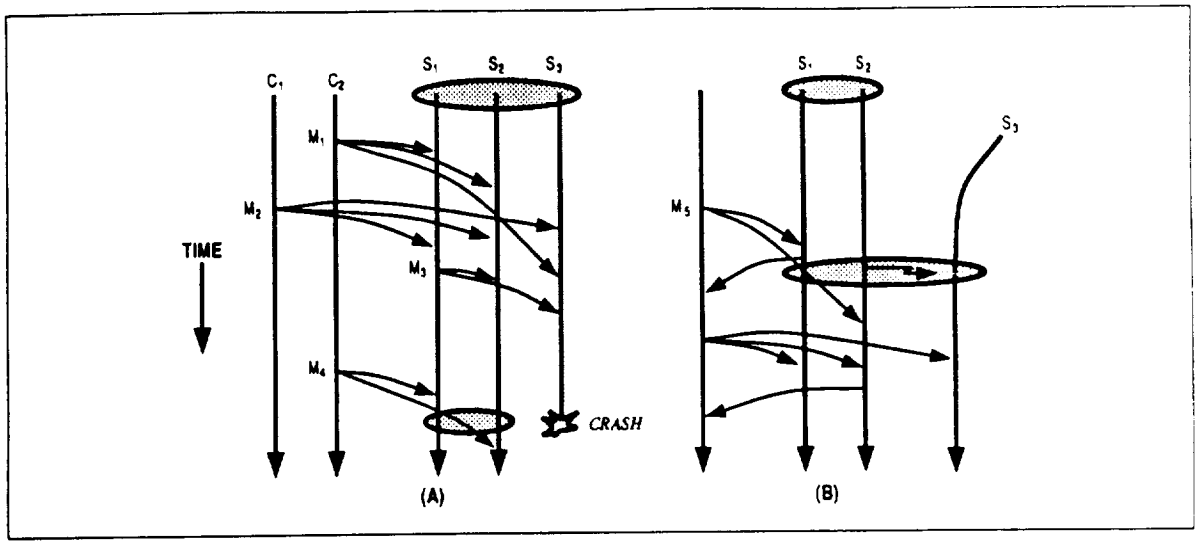

Figure 3: Message ordering problems

to compensate when it occurs. The solution involves tagging messages with enough context information to recognize when they arrive out of order, and to delay such messages appropriately.

Message m_4 exhibits an additional ordering problem. Here, s_1 believes that the service contains three members at the time m_4 arrives. Suppose that m_4 triggers a database search, and that process s_i searches the i/n'th part of the database. Process s_1 will search the first third. However, process s_2 receives m_4 after observing the failure of s_3, so it will search the second *half*. One sixth of the database will not have been searched, and the two responses will be inconsistent.

Thus, we see a whole range of ordering issues. Each is solvable, but in each case non-trivial application-level code would be required. Unfortunately, as seen in the subsections that follow, other issues raised in the figure are much harder to solve.

State transfer

Figure 3-b illustrates a slightly different problem. Here, we wish to transfer the "state" of the service to a new member, perhaps a program that has restarted after a failure (having lost prior state), or a server that has been added to redistribute load. Intuitively, the state of the server will be a data structure reflecting the data managed by the service, as modified by the updates that were done prior to when the new member joined the group. However, in the execution shown, a message has been sent to the server concurrent with the membership change. A consequence is that the new member, s_3, receives a state that does not reflect message m_3. This would leave s_3 inconsistent with s_1 and s_2. Solving this problem involves a

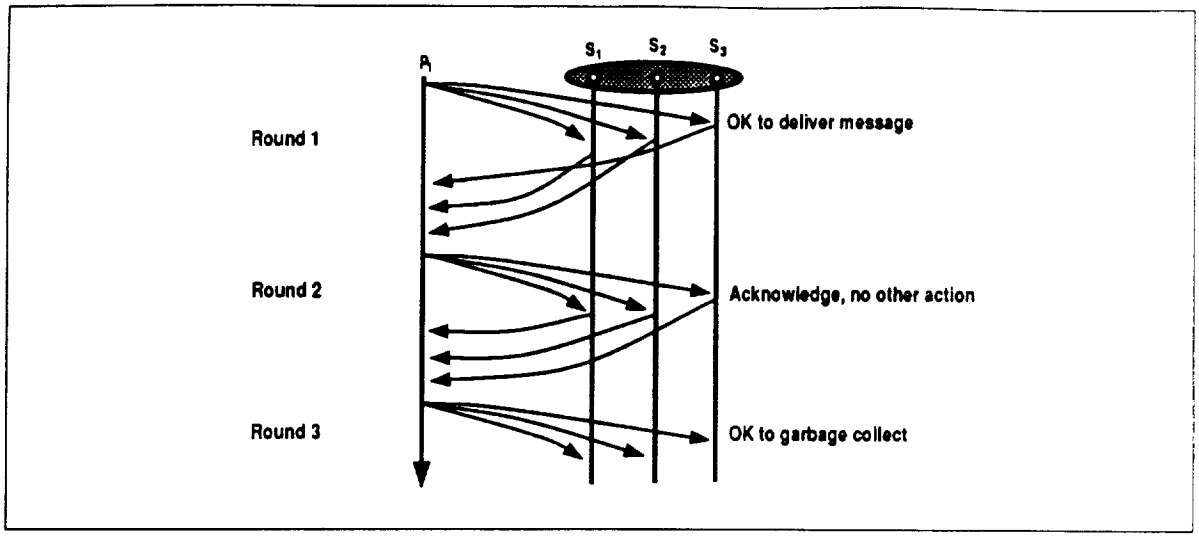

Figure 4: Three-round reliable multicast

complex synchronization algorithm (we won't present it here), and would be beyond the ability of a typical distributed applications programmer.

Fault tolerance

So far, our discussion has ignored failures. Failure raises many issues; here, we consider just one. Suppose that the sender of a message were to crash after some, but not all, destinations receive it. The destinations that do have a copy will need to complete the transmission. The protocol used should achieve exactly once delivery of each message, with bounded space overhead (the recipients must be able to garbage collect any information generated while the protocol is running).

Protocols to solve this problem can be complex, but a fairly simple solution can be based on Skeen's non-blocking three-phase atomic commit protocol [Ske82]. The protocol uses a three round sequence of RPC's to the destinations, as illustrated in Figure 4. During the first round, the sender sends the message to the destinations, which acknowledge receipt. Although the destinations can deliver the message at this point, they need to keep a copy: should the sender fail during the first round, the destination processes that have received copies will need to finish the protocol on its behalf. If no failure occurs, the sender tells all destinations that the first round has finished. They acknowledge this message and make a note that the sender is entering the third round, but take no other action. During the third round, each destination discards all information about the message – it deletes the saved copy of the message and any other data it was maintaining.

To handle failures, assume first that the destinations have a way to *detect* the failure of the sender. When a failure occurs, a process that has received a first or second round message can terminate the protocol. The basic idea is to have some member of the destination set take over the round that the sender was running when it failed; processes that have already received messages in that round detect the duplicates and respond to them as they responded after the original reception. The protocol is straightforward, and we leave the details to the interested reader.

Unfortunately, failure detection is not trivial. Many systems use timeout as a failure detection scheme, but such an approach would not be correct in the protocol sketched above. Suppose that process p starts sending first-round messages to processes $s_1 ... s_n$, but is delayed after sending the message to s_1. Upon receiving this message, s_1 will begin to monitor p, and eventually a timeout will occur. Process s_1 will now take over and run the protocol to completion, i.e. all the processes will receive and execute the message and forget completely about the interaction. Now, consider the situation if p was experiencing a transient problem that corrects itself. It will resume transmission by sending messages to $s_2 ... s_n$. None of these destinations will recognize that these messages are duplicates, so each will accept the message a second time!

One way to solve this problem would be for each recipient to save some information after delivering a message, for use in recognizing duplicates. But, a trading system may generate 1000 messages per second. If 16 bytes were retained for each message, a process might consume a megabyte of storage every minute. A better approach is to substitute a *reliable failure agreement mechanism* for the failure-detection timer. Such a protocol is described in [RB91]; among other functions, it filters messages to prevent a faulty process from interacting with operational processes without first executing a recovery protocol.

As noted at the start of this subsection, this is a relatively simple fault-tolerant multicast[1] protocol. In particular, this protocol fails to obtain any form of pipelined or asynchronous data flow when invoked many times in succession, and the use of RPC limits the degree of communication concurrency during each round (it would be better to send all the messages at once, and to collect the replies in parallel). Much better multicast protocols have been described in the literature, but improved performance often comes at the cost of increased complexity. Moreover, process group programming raises additional fault-tolerance issues, such as the fault-tolerance of the group addressing mechanism. None of these problems are intractable, but they result in a complex collection of mechanisms that must all work in concert.

[1] In this paper we use the term *multicast* to refer to sending a single message to the members of a process group. The term *broadcast* is more common in the literature, but is sometimes confused with the hardware broadcast capabilities of devices like ethernet. While a multicast might make use of hardware broadcast to reduce the number of messages used in the protocol, this would simply represent one possible implementation strategy, and for some situations, alternative approaches, such as an implementation over point-to-point messages, might perform better.

Summary of issues

The above discussion pointed to some of the potential pitfalls that confront a programmer who might undertake to solve the problem over a conventional operating system, such as UNIX: (1) group address expansion, (2) delivery ordering for concurrent messages, (3) delivery ordering for sequences of related messages, (4) state transfer, and (5) failure atomicity. This list is not exhaustive: we have overlooked questions involving real-time delivery guarantees, and persistent databases and files. However, our work on ISIS treats process group issues under the assumption that any real-time deadlines are weak, and although ISIS includes mechanisms for managing persistent data, they are optional. The list does cover the major issues that arise in this more restrictive domain [BC90]

At the start of this section, we asserted that modern operating systems lack the tools needed to develop group-based software. A basic premise of the ISIS project is that, although all of these problems can be solved, the complexity associated with working out the solutions and integrating them in a single system will be unmanageable for non-expert programmers. The only practical approach is to solve these problems in the distributed computing environment itself, or even the operating system. This permits the system to be engineered in a way that will give good, predictable performance and that takes full advantage of hardware and operating systems features. Furthermore, providing process groups as an underlying tool permits the programmer to concentrate on the problem at hand, as in the case of support for building graphical interfaces. If the implementation of process groups is left to the application designer, non-experts are unlikely to use the approach. The simple brokerage application of the introduction would be extremely difficult to build using the tools provided by a conventional operating system.

4 Virtual synchrony

ISIS simplifies process-group programming, and solves the issues raised in the preceding section, using a method motivated by database concurrency control. We will present the approach in two stages. First, we discuss an execution model called *close synchrony*. This model is then relaxed to arrive at the *virtually synchronous* model that ISIS implements. The relationship between our work and database serializability will be discussed in Sec. 7.

ISIS encourages programmers to assume a *closely synchronized* style of distributed execution [BJ89,Sch88], in which one event happens at a time. Here, the term "event" is used loosely, connoting not just a single message, but rather any single event that multiple members of a group might observe. More precisely:

- The execution of a process consists of a sequence of events, which may be internal computation, message transmissions, message deliveries, and changes to the membership of groups which it creates or joins.

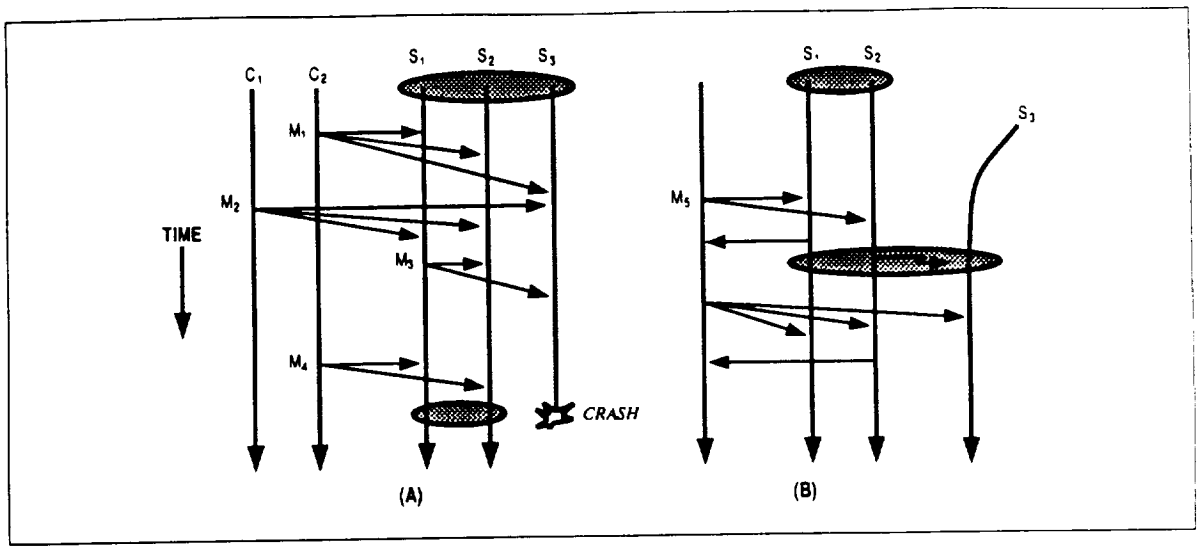

Figure 5: Closely synchronous execution

- A global execution of the system consists of a set of process executions. At the global level, one can talk about messages sent as *multicasts* to process groups.
- Any two processes that observe the same global event (i.e. by receiving messages from the same multicast, or by participating in the same group) see the corresponding local events in the same order.
- A multicast to a process group is delivered to its full membership, interpreted at the time when the delivery will be scheduled by the system. Here, we assume that senders specify a process group destination using an address that is expanded by the multicast protocol to comprise the actual set of destination processes.

Close synchrony represents a powerful guarantee. In fact, as seen in Fig. 5, it eliminates all the problems identified in the preceding section:

(1) *Group address expansion*: In a closely synchronous execution, the membership of a process group is fixed at the logical instant when a multicast is delivered. A system implementing closely synchronous group address expansion would need to synchronize communication events with group membership changes. For example, ISIS delays membership changes until "prior" multicasts have all been delivered to their destinations, and delays new multicasts until after any pending membership change has completed. This scheduling is invisible to the application programmer, although it sometimes introduces slight delays in communication.

(2) *Delivery ordering for concurrent messages*: In a closely synchronous execution, concurrently issued multicasts would be treated as distinct events. They would therefore be seen in the same order by any

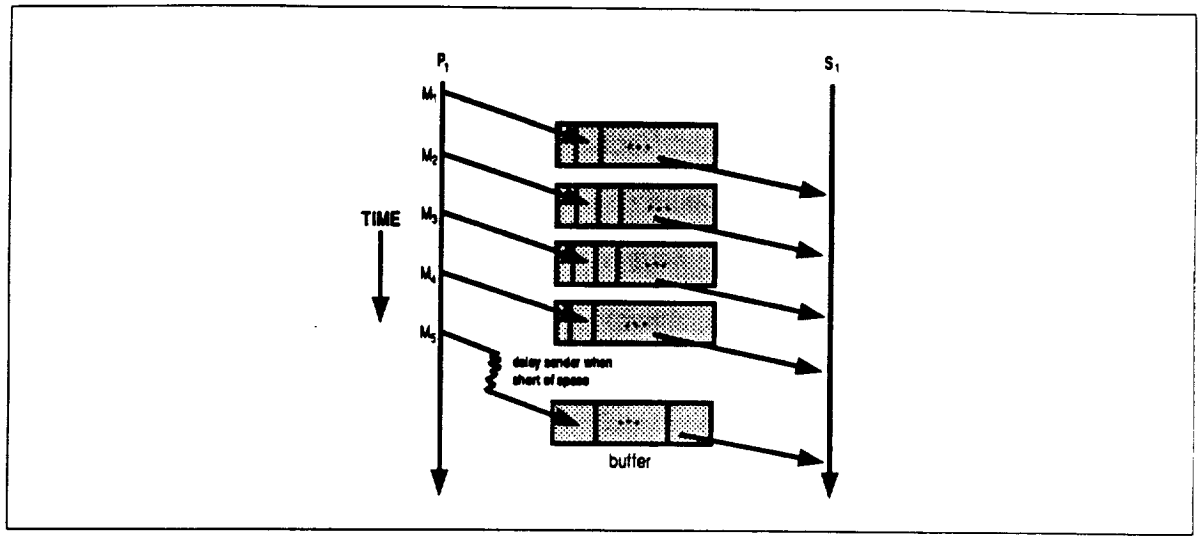

Figure 6: Asynchronous pipelining

destinations that they have in common. In practice, this means that a system supporting the model would transmit messages using a protocol that picks a delivery order and enforces it.

(3) *Delivery ordering for sequences of related messages*: In Figure 5a, process s_1 sent message m_3 after receiving m_1. We could say that $send(m_1)$ *happens before* $send(m_3)$. Processes executing in a closely synchronous world will never see anything to contradict the *happens before* relation. In practical terms, a system will need to add enough extra information to m_3 so that it can be delayed on reception if m_1 has not yet been received.

(4) *State transfer*: State transfer occurs at a well defined instant in time in the model. If a group member checkpoints the group state at the instant when a new member is added, or sends something based on the state to the new member, the state will be well defined and complete.

(5) *Failure atomicity*: The close synchrony model implicitly provides failure atomicity, by treating a multicast as a single logical event. Systems supporting close synchrony would have to implement multicast using a fault-tolerant protocols.

Unfortunately, although closely synchronous execution simplifies distributed application design, it would be too costly to employ in a practical setting. The most serious problem originates in the coupling between the sending of a message and delivery. According to the first rule, a multicast to a group address will be delivered to the full membership of a process group, interpreted at the time of delivery. Even if a process doesn't need responses from the destinations of a multicast, the model will block the sender of a multicast until the deliveries take place, so that the initiation and delivery of the multicast can be presented as a single, indivisible event.

In distributed systems, high performance comes from *asynchronous* interactions: patterns of execution in which the sender of a message is permitted to continue executing without waiting for delivery. An asynchronous approach treats the communications system like a bounded buffer, blocking the sender only when the rate of data generation exceeds the rate of consumption, or when the sender needs to wait for a reply or some other input (Figure 6). The advantage of this approach is that the latency (delay) between the sender and the destination does not affect the data transmission rate – the system operates in a *pipelined* manner, permitting both the sender and destination to remain continuously active. A closely synchronous execution would preclude such pipelining, delaying the execution of the process that sends a message until its delivery.

When we built ISIS, we wanted to benefit from close synchrony, but we didn't want to pay this price. Consequently, the system implements an approximation to close synchrony. The idea is that for each application, events are synchronized only to the degree that the application is sensitive to event ordering. In some situations, this approach will be identical to close synchronization: for example, when the group state is transferred to a new member. Here, it is important that the state seen by the new member correspond to the one seen by the old members at the logical instant of the join. Messages prior to the join must be flushed through, and the sequence of events seen by each process rigidly controlled. But, in other situations, it may be possible to deliver messages in different orders at different processes, without the application noticing. This permits a more asynchronous execution. Intuition into the idea can be obtained by considering how database systems implement serializability using two-phase locking: data servers sometimes process requests "out of order", but the resulting execution is indistinguishable from a serial one [BHG87].

Order sensitivity in distributed systems.

To better understand the ways that a process group can be sensitive to event orderings, we consider a simple example. Suppose that we wish to develop a service to manage the trading history for a set of stocks. A set of tickerplants[2] monitor the prices of futures contracts for soybeans, pork-bellies, and other commodities. Each significant price change causes a multicast by the tickerplant to the database server, which appends the new event to a list indexed by stock name. A query interface allows programs to obtain the previous n quotes for a specified commodity.

One can imagine two styles of tickerplant. In the first, pork-belly quotes might originate in any of several tickerplants, hence two different quotes (perhaps, one for Tokyo and one for New York) could be multicast concurrently by two different processes. In a second plausible design, only one tickerplant would actively multicast quotes for a given future at a time. Other tickerplants might buffer recent quotes to enable recovery from the failure of the primary server, but would never multicast them unless the primary fails.

[2] A tickerplant is a program or device that receives telemetry input directly from a stock exchange or some similar source.

Now, suppose that a key correctness constraint on the system is that the database servers behave like a single, highly reliable service. In particular, regardless of which process handles a query, the outcome should be the same. Close synchrony would yield such a service.

How sensitive are the servers to event ordering in this example? Using the first tickerplant protocol, one needs a multicast primitive capable of delivering concurrent messages in the same order at all overlapping destinations. This is normally called an *atomic delivery ordering*, and corresponds to an ISIS primitive called ABCAST.

The second style of system has a simpler ordering requirement. Here, as long as the primary tickerplant for a given commodity is not changed, it suffices to deliver messages in the order they were sent: messages sent concurrently concern different commodity. Since the query interface only returns data for a single commodity at a time, the order in which updates are done for *different* commodity is not seen by users.[3] The ordering requirement here is first in, first out (FIFO).

Now, suppose that "primaryness" *could* change dynamically, in response to a failure or to balance load. For example, perhaps one tickerplant is handling both soybeans and pork-bellies in a heated market, while another is monitoring a slow day in petroleum products. The latency on reporting quotes could be reduced by sharing the load more evenly. However, even during the reconfiguration, it remains important to deliver messages in the order they were sent, and this ordering might span multiple processes. If tickerplant t_1 sends quote q_1, and then sends a message to tickerplant t_2 telling it to take over, and tickerplant t_2 might send quote q_2 (figure 7). Logically, q_2 follows q_1, but the delivery order is seen along a thread of computation that spans multiple processes, whereas a FIFO order would normally be concerned only with the order in which messages are sent by a specific process.

Lamport [Lam78] calls the relationship between events in a thread of computation such as this a *causal ordering*, and would say that the transmission of q_1 *causally precedes* that of q_2 because these two events are related by a chain of message transmissions and receptions. We can write this as $send(q_1) \rightarrow send(q_2)$. Causal ordering is partial (concurrent events are not causally related), and is always consistent with the actual wall-clock times that events occur. A sufficient ordering property for the second style of system is that if $send(q_1) \rightarrow send(q_2)$ then $rcv(q_1)$ occurs before $rcv(q_2)$ at any destinations shared by both multicasts. This is called a *causal delivery ordering*, and is available in ISIS through a multicast primitive called CBCAST. Notice that CBCAST is weaker than ABCAST, because it permits messages that were sent concurrently to be delivered to overlapping destinations in different orders.[4]

[3] One could change the query interface so this would not be true. For example, suppose that two brokers in the same firm use a financial model that relates port-belly prices to soybean prices. One broker trades pork and bean futures in Chicago while another trades matching lots of pork-bellies and soybeans in Tokyo. To ensure that the market analysis programs these brokers run makes consistent recommendations, they should operate on consistent data streams, and hence the ABCAST ordering would be needed.

[4] The statement that CBCAST is "weaker" than ABCAST may seem imprecise: as we have stated the problem, the two protocols simply provide different forms of ordering. However, the ISIS version of ABCAST actually extends the partial CBCAST ordering into

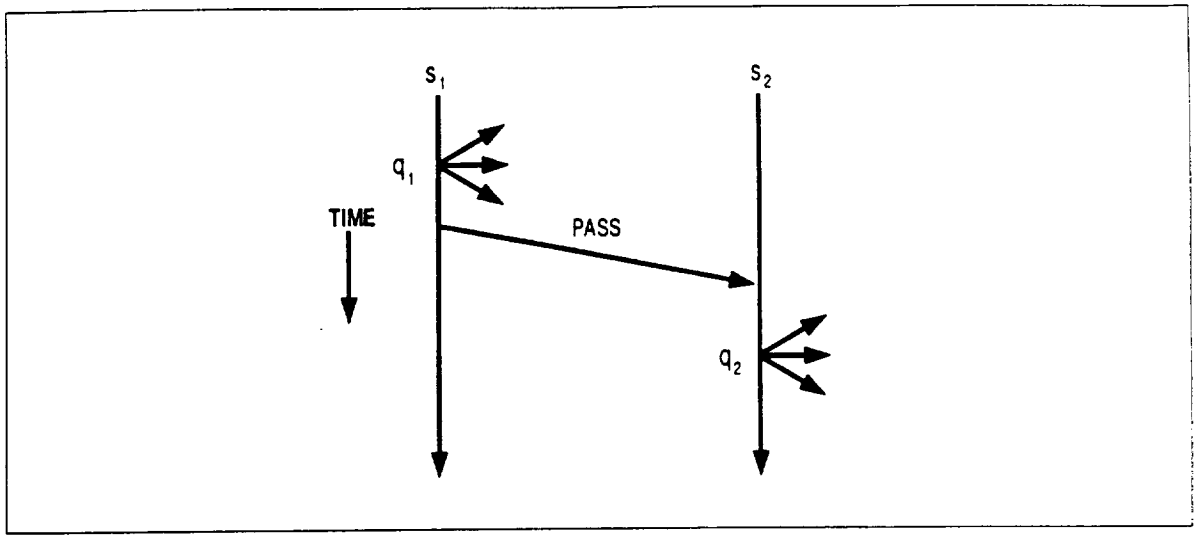

Figure 7: Causal ordering

Efficient recovery from a surge of activity in the pork-bellies pit may not seem like a compelling reason to employ causal multicast. However, the same communication pattern also arises in a more common setting: a process group that manages replicated (or cohently cached) data. Processes that update such data typically obtain a lock or mutual exclusion, then issue a stream of asynchronous updates, and then release the lock. By using CBCAST for this communication, an efficient, pipelined data flow is achieved. A process will only block if it requests a lock that it was not the last process to hold, or when communication buffering capacity is exceeded [JB89,BJ89].

The distinction between causal and total event orderings (CBCAST and ABCAST) has parallels in other settings. Although ISIS was the first distributed system to enforce a causal delivery ordering as part of a communication subsystem [Bir85], Lamport's had shown much earlier that the causal ordering is the fundamental form of time in a distributed setting. His insight was motivated by the physical theory of information and time [Lam78]. Moreover, close synchrony is related to Lamport's *state machine approach* to developing distributed software [Sch86]. Work on parallel processor architectures has yielded a memory update model called *weak consistency* [DSB86,TH90], which uses a similar principle to increase parallelism in the cache of a parallel processor. And, a causal correctness property has been used in work on *lazy update* in shared memory multiprocessors [ABHN91] and distributed database systems [JB89,LLS90]. A more detailed discussion of this issue appears in [Sch88,BJ89].

a total one: it is a *causal atomic* multicast primitive.

4.1 Chosing the right multicast primitive

One might wonder how a virtually synchronous system could make use of CBCAST. Recall that the basic approach encourages developers to specify a system in terms of a closely synchronous execution, which entails using ABCAST for all communication. Fortunately, ISIS users rarely code directly in terms of process groups and group multicast. Instead, they generally make heavy use of the higher-level software tools available with the system. If these tools execute asynchronously most applications will, too.

The ISIS toolkit has been designed to use CBCAST wherever possible. This enables a pipelined style of execution in which messages are emitted by a sender that need not delay after each send request. Thus, in addition to scheduling the delivery of messages to conform to the virtual synchrony model, ISIS plays the role of a producer-consumer buffering system. In our experimental work, CBCAST is between twice as fast and twenty-five times faster than ABCAST, depending on the degree to which the application is successful in pipelining communication; the actual data transmission rates are as good as for UNIX or TCP streams [BSS91]. Slower speeds are seen in applications that require responses from the servers on each message, as in a database query, while higher performance is seen in applications that publish data streams, like the analysis and tickerplant components of the brokerage system.

4.2 Summary of benefits due to virtual synchrony

The need for brevity precludes a more detailed discussion of virtual synchrony, or how it is used in developing distributed algorithms within ISIS. However, it may be useful to summarize the benefits of the approach:

- The ability to develop code using a simplified, closely synchronous execution model.
- A meaningful notion of group state and state transfer, both when groups manage replicated data, and when a dynamically changing rule is used to partition computation among group members.
- Efficient, pipelined communication.
- Treatment of communication, process group membership changes and failures through a single, event-oriented execution model.

Although other approaches offer some of the same properties, the virtual synchrony model is unusual in combining them within a single framework. Our experience solving problems using ISIS leaves us convinced that these issues are encountered in even the simplest distributed applications.

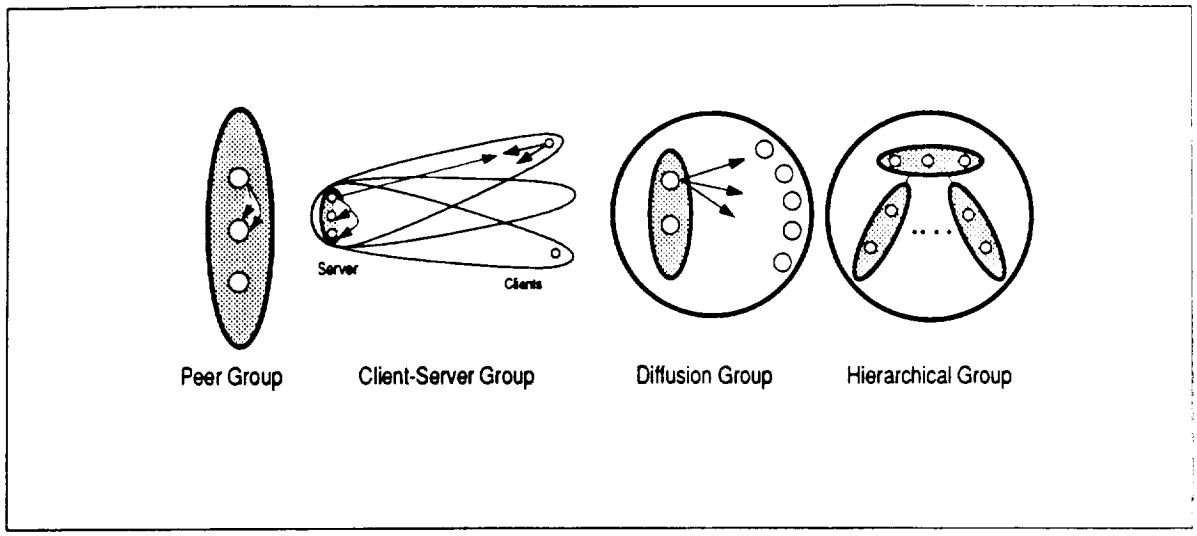

Figure 8: Styles of groups

5 The application interface

The ISIS application interface is concerned with presenting higher-level mechanisms for forming and managing process groups and implementing group-based software. This section illustrates the general approach by discussing the styles of process group supported by the system and giving a simple example of a distributed database application.

5.1 Styles of groups

The efficiency of a distributed system is limited by the information available to the protocols employed for communication. This issue arose as an consideration in developing the ISIS process group interface, where a tradeoff had to be made between simplicity of the interface and the availability of accurate information about group membership for use in multicast address expansion. As a consequence, the application interface introduces four styles of process groups that differ in how the processes involve typically interact with the group, illustrated in Fig. 8 (anonymous groups are not distinguished from explicit groups at this level of the system).

Peer groups

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 and are limited to

at most 64 members. Peer groups support the full range of ISIS facilities, and also implement a particularly efficient communication protocol. A process joins a peer group using the `pg_join` system call, which creates the group if it is not already active, and adds the process to the group otherwise. Options exist for logging/checkpointing the state of the group and reloading the logged state each time the group is restarted, for transferring data (state) from the active members of a group to a joining member, for checking the permissions of the new member.

Client-server groups

In *client-server* groups, a potentially large number of clients interacts with a peer group of servers. Requests may be multicast or issued as RPC's to a favored server, after an initial setup. Servers either respond using point-to-point messages or use multicast to reply atomically 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. A backup server will then know which requests were pending.

The clients of a group can only multicast to it and received replies; they have no direct way to monitor message passing within the group, or to learn the addresses of other clients. ISIS supports two classes of clients. A one-time user of a group can interact with it by looking up its group address, via the system name server, and sending a message. Such an *ad-hoc* interaction employs a slightly inefficient protocol, but since the cost of the whole sequence would still be measured in the tens of milliseconds, this may not be a concern if the client will not interact with the group again. On the other hand, some client programs interact repeatedly with a group. In such applications, it is desirable to reduce the cost of client-group communication to a minimum. Accordingly, ISIS also supports an interface with which the client can connect to the group for an extended period, called `pg_client`. The effect is to improve performance: a client registered through `pg_client` obtains performance close to that seen between the members. There is no limit to the number of clients that a group can support.

Diffusion groups

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 both to the full set of clients (those registered via `pg_client`), as well as to the other members of the service. This pattern of communication is used by applications to publish information for a varying set of subscribers. In current ISIS applications, diffusion groups are the only situations in which a typical multicast has a large number of destinations, and hence where ISIS would obtain a significant speedup from hardware multicast.

- *Process groups:* create, delete, join (transferring state).
- *Group multicast:* CBCAST, ABCAST, collecting 0, 1 QUORUM or ALL replies (0 replies gives an asynchronous multicast).
- *Synchronization:* Locking, with symbolic strings to represent locks. Token passing.
- *Replicated data:* Implemented by broadcasting updates to group having copies. Transfer values to processes that join using state transfer facility. Dynamic system reconfiguration using replicated configuration data. Checkpoint/update logging, spooling for state recovery after failure.
- *Monitoring facilities:* Watch a process or site, trigger actions after failures and recoveries. Monitor changes to process group membership, site failures, etc.
- *Distributed execution facilities:* Redundant computation (all take same action). Subdivided among multiple servers. Coordinator-cohort (primary/backup).
- *Automated recovery:* When site recovers, program automatically restarted. If first to recover, state loaded from logs (or initialized by software). Else, atomically join active process group and transfer state.
- *WAN communication:* Reliable long-haul message passing and file transfer facility.

Figure 9: ISIS tools at process group level

Hierarchical groups

The last group structure supported by ISIS is the *hierarchical group*. In large applications, 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 subgroup performs this mapping and can change it dynamically. 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.

5.2 The toolkit interface

As noted earlier, the performance of a distributed system is often limited by the degree of pipelining (asynchronous communication) achieved. The development of asynchronous solutions to distributed problems can be tricky, and many ISIS users would employ less efficient solutions rather than risk errors. For this reason, the toolkit includes asynchronous implementations of the more important distributed programming paradigms. These include a synchronization tool that supports a form of locking (based on distributed tokens), a replication tool for managing replicated data (even the updates are performed completely without blocking), a tool for fault-tolerant performance of requests using a primary-backup programming style, and so forth (a partial list appears in Figure 9). Using these tools, and following programming examples in the ISIS manual, even non-experts have been successful in developing fault-tolerant, highly asynchronous distributed software.

Figures 10 and 11 show a complete, fault-tolerant database server for maintaining a mapping from names (ascii strings) to salaries (integers). The example is in standard C, although ISIS is also callable from C++, FORTRAN, and Common Lisp, and interfaces to Ada and Modula-3 are now under development. The server initializes ISIS and declares the procedures that will handle update and inquiry requests. The `isis_mainloop` dispatches incoming messages to these procedures as needed (other styles of main loop are also supported). Notice the formatted-I/O style of message generation and scanning. ISIS does not actually send data in ascii format, of course, but the interface mimics the usual UNIX formatted I/O interface because most ISIS users are comfortable with this approach.

The "state transfer" routines are concerned with sending the current contents of the database to a server that has just been started and is joining the group. In this situation, ISIS arbitrarily selects an existing server to do a state transfer, invoking its state sending procedure. Each call that this procedure makes to `xfer_out` will cause to an invocation of `rcv_state` on the receiving side; in our example, the latter simply passes the message to the update procedure (the same message format is used by `send_state` and `update`). Of course, there are many variants on this basic scheme; for example, it is possible to indicate to the system that only certain servers should be allowed to handle state transfer requests, to refuse to allow certain processes to join, and so forth.

The client program should be largely self-explanatory. At startup, it does a `pg_lookup` to find the server. Subsequently, calls to its query and update procedures are mapped into messages to the server. The BCAST calls are mapped to the appropriate default for the group – ABCAST in this case.

The database server of Figure 10 uses a redundant style of execution in which the client broadcasts each request and will receive multiple, identical replies from all copies. In practice, the client will wait for the first reply, ignoring the others. Such an approach provides the fastest possible reaction to a failure, but has the disadvantage of consuming n times the resources of a fault-intolerant solution, where n is the size of

```
#include  "isis.h"
#define UPDATE      1
#define QUERY       2
main()
  {
        isis_init(0);
        isis_entry(UPDATE, update, "update");
        isis_entry(QUERY, query, "query");
        pg_join("/demos/salaries", PG_XFER, send_state, rcv_state, 0);
        isis_mainloop(0);
  }
update(mp)
  register message *mp;
  {
        char name[32];
        int  salary;
        msg_get(mp, "%s,%d", name, &salary);
        set_salary(name, salary);
  }
query(mp)
  register message *mp;
  {
        char name[32];
        int  salary;
        msg_get(mp, "%s,%d", name);
        salary = get_salary(name);
        reply(mp, "%d", salary);
  }
send_state()
  {
        struct sdb_entry *sp;
        for(sp = sdb_head; sp != sdb_tail; sp = sp->s_next)
            xfer_out("%s,%d", sp->s_name, sp->s_salary);
  }
rcv_state(mp)
  register message *mp;
  {
        update(mp);
  }
```

Figure 10: A simple database server

```
#include  "isis.h"
#define UPDATE      1
#define QUERY       2
address *server;
main()
  {
        isis_init(0);
        server = pg_lookup("/demos/salaries");
        ...
  }
update(name, salary)
  char *name;
  {
        bcast(server, UPDATE, "%s,%d", name, salary, 0);
  }
get_salary(name)
  char *name;
  {
        int salary;
        bcast(server, QUERY, "%s", name, 1, "%d", &salary);
        return(salary);
  }
```

Figure 11: A simple database client

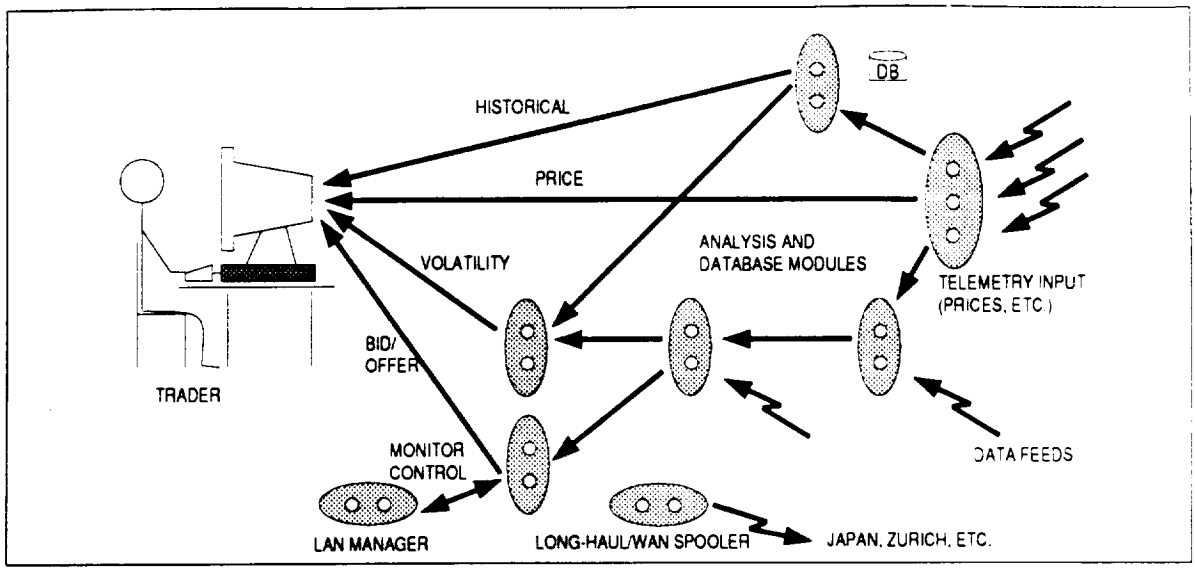

Figure 12: Architecture of brokerage system

the process group. An alternative would have been to subdivide the search so that each server performs $1/n$'th of the work. Here, the client would combine responses from all the servers, repeating the request if a server fails instead of replying (this is readily detected in ISIS).

6 Who uses Isis, and how?

The example of the previous section reveals the general nature of the ISIS interface, but may leave the reader with little sense of the broader picture. This section briefly reviews some substantial ISIS applications, looking at the roles that ISIS plays in real-world situations.

6.1 Brokerage

A number of ISIS users are concerned with financial computing systems such as the one cited in the introduction. Figure 12 illustrates such a system. The architecture is a client-server one, in which the services filter and analyze streams of data. Fault-tolerance here refers to two very different aspects of the application. First, financial systems must rapidly restart failed components and reorganize themselves so that service will not be interrupted by software or hardware failures. Second, there are specific system functions that require fault-tolerance at the level of files or database, such as a guarantee that after rebooting a file or database manager will be able to restore the data it manages into a consistent form at low cost. ISIS was designed to address the first sort of problem, but includes several tools for solving the latter one.

Generally, the approach taken is to represent key services using process groups, replicating service state information so that even if one server process fails the other can respond to requests on its behalf. During periods when n service programs are operational, one can often exploit the redundancy to improve response time; thus, rather than asking how much such an application must pay for fault-tolerance, more appropriate questions concern the level of replication at which the overhead begins to outweigh the benefits of concurrency, and the minimum acceptable performance assuming k component failures. Fault-tolerance is something of a side-effect of the replication approach.

A second attribute of financial computing is use of a subscription/publication style of computing. The basic ISIS communication primitives do not spool messages for future replay, hence an application running over the system, the NEWS facility, has been developed to support this functionality.

A final attribute of brokerage systems is that they require a dynamically varying collection of services. A firm may work with dozens or hundreds of financial models, predicting market behavior for the financial instruments being traded under varying market conditions. Only a small subset of these services will be needed at any time. Thus, systems of this sort generally consist of a processor pool on which services can be started as necessary, and this creates a need to support an automatic remote execution and load balancing mechanism. The heterogeneity of typical networks complicates this problem, by introducing a pattern matching aspect (i.e., certain programs may be subject to licensing restrictions, or require special processors, or may simply have been compiled for some specific hardware configuration). This problem is solved using the ISIS network resource manager, an application described later in this section.

6.2 NMRD example

Several ISIS applications combine local area and wide-area networking functions. A good example of this arises in the *Nuclear Monitoring Research and Development System*, or NMRD, being developed by Science Applications International Corporation.[5] NMRD includes several knowledge-based applications which collect, analyze and archive seismic data from a geographically dispersed network of seismic sensors, and a rich set of tools for selecting and analyzing data in the archive to address seismological issues. The system is extensively automated with rule-based AI techniques.

A typical NMRD application is the *Intelligent Monitoring System* (IMS), which detects, locates and classifies seismic events occurring in Eurasia. IMS is structured like a wheel. A central "hub" in Washington, DC performs most of the automated data interpretation functions, and a set of "spokes" connects this hub to free-standing LANs, where data acquisition, signal processing, and archiving is done. The spokes comprise the WAN communication network, and consist of long-distance TCP channels.

[5]DARPA Contract No. MDA972-88-C-0024

The bandwidth of the spokes is low, hence it is impossible to transfer the bulk of the seismic data collected by the system to the hub. Accordingly, remote systems select and characterize data segments which may contain signals of interest. They send these descriptions to the hub, which may request a full copy of some segment of the signal, or initiate a remote signal analysis operation. In either case, the result will be in the form of a file that must be transferred to the hub. Because the system is automated, the fault-tolerance of these operations is critical to correctness. IMS would malfunction if a requested data segment or signal analysis operation was never received at the hub. This imposes fault-tolerance requirements within the LAN systems running on the hub, on remote LANs, and on the WAN communication subsystem.

The steps involved in a raw data transfer are illustrative of the system architecture used to address these needs. First, the ISIS "long-haul" utility is invoked by an IMS program that needs data from a remote node. This IMS program will be implemented as a process group for reasons of fault-tolerance, using a primary/backup approach. The request sent to the long-haul utility takes the form of a message addressed to a process group (identified symbolically), and giving the remote network or networks to which it should be delivered. The long-haul facility, also built as a fault-tolerant process group, operates by opening a line to the remote network, or spooling the message if the remote LAN is temporarily unreachable. Using an acknowledgement protocol, the facility provides exactly-once, error-free transmission over the long-haul link (even if the link must be closed and reopened during the session, or if the processes handling the link fail). Remotely, the facility locates the process group to which the message was addressed, spooling the message or starting the desired service if necessary. Finally, the message is delivered. If the message contains a reference to a file, the long-haul system automatically transfers the file, too.

Thus, although ISIS process groups and group communication do not transparently span wide-area communication links, ISIS can be an effective tool for developing software that does have this structure. The benefit seen by the developers of IMS was not that ISIS offered a trivial solution to their wide-area problem, but rather that it offered robust tools with which a highly automated piece of software could be constructed. Because IMS is normally operated without supervision, this was an important consideration in system design.

6.3 Graphics example

ISIS has been popular with a community of scientific computing and simulation users, typified by the Cornell Program of Computer Graphics. This group has developed a number of computationally intensive graphics applications, of which a rendering technique called *radiosity* is typical [Gre91]. In broad terms, the approach involves precomputing a mathematical model of a scene to be rendered. The scene can then be illuminated and rendered from various perspectives at much lower cost than if each rendering was done independently. Such techniques play important roles in solid modeling, cooperative design, real-time animation and virtual reality applications.

The Cornell group has used ISIS for several years. Many of the algorithms for this application are "embarrassingly parallel." They consist of long computational steps that can be executed completely independently, with the results being combined periodically. Some of these applications execute for days or weeks on substantial numbers of the fastest workstations now available. ISIS is well suited to controlling such a computation. Because the calculations execute for extended periods of time, the primary emphasis is on control of the collection of machines being used, and dynamic reconfiguration of the application as availability of processors varies during the day. Communication is not a bottleneck, hence the overhead introduced by ISIS is not viewed as a concern. Of course, the experience in developing these sorts of applications is not uniformly positive: for cases where the job step is much shorter, the speed of ISIS communication (presumably, any sort of communication) becomes a limiting factor, hence only certain algorithms and problems can make effective use of the approach. Nonetheless, the community of ISIS users includes a large number of scientific computing and simulation users, all using this style of computing and benefiting from the excellent cost/performance ratio seen in networks of inexpensive workstations. The same approach is used in some ISIS-based utility software, such as the "parallel make" program, which is a version of the traditional UNIX make utility, modified to perform steps in parallel whenever possible.

6.4 Major Isis-based utilities

In the above subsection, we alluded to some of the fault-tolerant utilities that have been built over ISIS. There are currently five such systems:

- NEWS: This application supports a collection of communication topics to which users can subscribe (obtaining a replay of recent postings) or post messages. Topics are identified with file-system style names, and it is possible to post to topics on a remote network using a "mail address" notation; thus, a Swiss brokerage firm might post some quotes to "/GENEVA/QUOTES/IBM@NEW-YORK". The application creates a process group for each topic, monitoring each such group to maintain a history of messages posted to it for replay to new subscribers, using a state transfer when a new member joins.

- NMGR: This program manages batch-style jobs and performs load sharing in a distributed setting. This involves monitoring candidate machines, which are collected into a processor pool, and then scheduling jobs on the pool. A pattern matching mechanism is used to optimize job placement. When employed to manage critical system services (as opposed to running batch-style jobs), the program monitors each service and automatically restarts failed components. *Parallel make*, mentioned above, is an example of a distributed application program that uses NMGR for job placement.

- DECEIT: This system [SBM89] provides fault-tolerant NFS-compatible file storage. Files are replicated for both to increase performance (by supporting parallel reads on different replicas) and

fault-tolerance; the level of replication is varied depending on the style of access detected by the system at runtime. After a failed node recovers, any files it managed are automatically brought up to date. The approach conceals file replication from the user, who sees an NFS-compatible file-system interface.

- META/LOMITA: META is an extensive system for building fault-tolerant reactive control applications [MCWB91]. It consists of a layer for instrumenting a distributed application or environment, by defining *sensors* and *actuators*. A sensor is any typed value that can be polled or monitored by the system; an actuator is any entity capable of taking an action on request. Built-in sensors include the load on a machine, the status of software and hardware components of the system, and the set of users on each machine. An unlimited collection of user-defined sensors and actuators can be added.

  The "raw" sensors and actuators of the lowest layer are mapped to *abstract* sensors by an intermediate layer, which also supports a simple database-style interface and a triggering facility. This layer supports an entity-relation data model and conceals many of the details of the physical sensors, such as polling frequency and fault-tolerance. The interface supports a simple trigger language, which will initiate a pre-specified action when a specified condition is detected.

  Running over META is a distributed language for specifying control actions in high-level terms, called LOMITA. LOMITA code is normally imbedded into conventional C or C++ software. At runtime, the control statements are expanded into a distributed finite state machine triggered by events that can be sensed local to a sensor or system component; a process group is used to perform the state transition More detail on the approach can be found in [Woo91].

- SPOOLER/LONG-HAUL FACILITY: This subsystem is responsible for wide-area communication [MB90] and for saving messages to groups that are only active periodically. It conceals link failures and presents an exactly-once communication interface.

6.5 General remarks

We believe that the simplicity of straightforward applications such as the replicated database server, together with our success in building much more sophisticated applications, supports a basic philosophy: that the functionality of a distributed application can profitably be separated from the distributed protocols and algorithms employed in support of it. If a simple service corresponds to a simple realization, application developers can safely undertake more complex services and algorithms. As seen above, ISIS users have constructed elaborate distributed systems, with very positive results. The complexity traditionally associated with distributed computing is overcome using the virtual synchrony model.

This point is not surprising: database applications are greatly simplified by the availability of database tools (and the serializability model), and window-oriented graphics applications by tools such as Motif, the

popular X-windows toolkit. Embedding the complex aspect of distributed computing into a specialized layer may seem the obvious way to handle this problem. Nonetheless, the main thrust of distributed computing environments in the late 1980's has been on issues of performance, support for remote procedure call, and techniques for making networking as transparent as possible. These are important issues, but they offer little help to the user concerned with distributed consistency and fault-tolerance.

7 Isis and other technologies

Our discussion has overlooked the sorts of real-time issues that arise in the Advanced Automation System, a next-generation air-traffic control system being developed by IBM for the FAA [CD90,CASD86], which also uses a process-group based computing model. Similarly, one might wonder how the ISIS execution model compares with transactional database execution models. These are both complex issues, and it would be difficult to do justice to them without a lengthy digression.

Briefly, the AAS technology differs from ISIS in providing strong real-time guarantees. The real-time characteristics of the current ISIS protocols have not been analyzed and no guarantees are provided. On the other hand, the AAS system has weaker consistency properties than ISIS. For example, AAS application software can have inconsistent views of replicated data, due to transient faults that corrupt a process. Further, such a fault may not prevent the corrupted process from initiating new multicasts. Thus, AAS application software must be designed to tolerate certain types of inconsistencies which would not arise using the ISIS virtual synchrony model. Integration of the two approaches represents as an open problem.

The relationship between ISIS and transactional systems represents a potential source of confusion. The problem originates in the similarity between the virtual synchrony model and a transactional serializability model [BHG87]. In fact, the basic building blocks of the ISIS system (process groups and group multicast) have no direct counterparts in database systems. The converse is also the case: ISIS offers no special support for transactional begin, read, write, commit/abort, concurrency control or rollback mechanisms. Thus, although the theory and protocols used in ISIS draw upon work from the database community, it would be inappropriate to view ISIS as a form of database system.

8 Rethinking Isis for modern operating systems

After six years of experience with the current version of ISIS, we find that the system has grown large and complicated. Meanwhile, improved insight into protocol design [BSS91], together with the emergence of reconfigurable operating systems, have convinced us to attempt to build a simpler, more scalable, and faster version of ISIS. On a high performance workstation over an ethernet (but not exploiting hardware broadcast),

ISIS currently performs about 350 CBCAST's per second from one sender to 4 destinations, or about 1000 per second to a single destination. We feel that numbers like 25,000 per second and 100,000 could be achieved through an approach that makes use of the powerful memory management and lightweight tasking mechanisms available under operating systems like Mach [ABG+86] and Chorus [RAA+88].[6]

At a more fundamental level, we have been studying distributed consistency using formal mathematical tools. Results in these area include the group-based failure detection protocol and the lightweight suite of reliable multicast protocols used to implement CBCAST and ABCAST [RB91,BSS91,Ste91]. In the future, we plan to look at security issues.

9 Conclusions

This paper suggested that the next generation of distributed computing systems will require support for process groups and group programming. Arriving at appropriate semantics for a process group mechanism is a difficult problem, and implementing those semantics fault-tolerantly would exceed the abilities of the average distributed applications designer. A fundamental choice is implied: either the operating system must implement these mechanisms or the reliability and performance of group-structured applications is unlikely to be acceptable.

The ISIS system provides tools for programming with process groups. A review of research on the system leads us to the following conclusions:

- A mechanism for achieving and maintaining distributed consistency is needed to construct reliable large-scale systems from collections of components. It is appealing to present such a mechanism in terms of process groups.
- Process groups should embody strong semantics for group membership, communication, and synchronization. A simple and powerful model can be based on closely synchronized distributed execution, but high performance requires a more asynchronous style of execution in which communication is heavily pipelined. The *virtual synchrony* approach combines these benefits, using a closely synchronous execution model, but deriving a substantial performance benefit when message ordering can safely be relaxed.
- Efficient protocols have been developed for supporting virtual synchrony. These are more complex than the sorts of protocols that have been common in the distributed computing and internetworking community, but the complexity seems to be unavoidable.

[6]This assumes that it will be possible to send 1000 packets per second, and that pipelining will permit each packet to carry 25 to 100 small, asynchronous messages. The former figure is conservative, and the latter corresponds to a levels of piggybacking seen in ISIS today.

Process group programming offers the potential to ignite a new wave of advances in distributed computing, and applications that reply on distributed computing. Using current technologies, it is impractical for typical developers to implement high reliability software, self-managing distributed systems, to employ replicated data or simple coarse-grained parallelism, or to develop software that reconfigures automatically after a failure or recovery. Consequently, although current networks embody tremendously powerful computing resources, the programmers who develop software for these environments are severely constrained by a deficient software infrastructure. The experience we have had with the ISIS system suggests that these obstacles can be overcome, resulting in a distributed programming environment that greatly simplifies the task confronted by the distributed applications programmers.

10 Acknowledgements

The ISIS effort would not have been possible without extensive contributions by many past and present members of the project, users of the system, and researchers in the field of distributed computing. Although it is not practical to list all of these individuals here, Tommy Joseph, Ken Kane and Frank Schmuck played major roles in the initial system design, while the protocols used in the current version of ISIS emerged from research by the author, Andre Schiper and Pat Stephenson. Messac Makpangou and Micah Beck developed the long-haul facility, Alex Siegel the Deceit file system, Tim Clark the network resource manager, and Keith Marzullo, Mark Wood and Robert Cooper developed Meta and Lomita. In addition to these researchers, Holger Herzog, Brad Glade, Barry Gleeson, Guerney Hunt, Mike Reiter, Aleta Ricciardi, and Robbert Van Renesse are all involved in the current system redesign.

The author gratefully acknowledges the help of Mauren Robinson, who prepared the figures for this paper. Robert Cooper, Bradford Glade, Barry Gleeson, Jacob Levy, Stefan Sharkansky, Fred Schneider, and Pat Stephenson all made valuable comments on earlier drafts, resulting in significant improvements in the presentation.

References

[ABG+86] Mike Accetta, Robert Baron, David Golub, R ichard Rashid, Avadis Tevanian, and Michael Young. Mach: A new kernel foundation for unix development. Technical report, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, August 1986. Also in Proceedings of the Summer 1986 USENIX Conference, pp. 93 –112, July 1986.

[ABHN91] Mustaque Ahamad, James Burns, Phillip Hutto, and Gil Neiher. Causal memory. Technical report, College of Computing, Georgia Institute of Technology, Atlanta, GA, July 1991.

[BC90] Ken Birman and Robert Cooper. The ISIS project: Real experience with a fault tolerant programming system. European SIGOPS Workshop, September 1990. To appear in Operating Systems Review, April 1991; also available as Cornell University Computer Science Department Technical Report TR90-1138.

[BHG87] Philip A. Bernstein, Vassos Hadzilacos, and Nathan Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987.

[Bir85] Kenneth P. Birman. Replication and availability in the ISIS system. In *Proceedings of the Tenth ACM Symposium on Operating Systems Principles*, pages 79–86, Orcas Island, Washington, December 1985. ACM SIGOPS.

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

[BSS91] Kenneth Birman, Andre Schiper, and Patrick Stephenson. Lightweight causal and atomic group multicast. *ACM Transactions on Computer Systems*, 9(3), August 1991.

[CASD86] Flaviu Cristian, Houtan Aghili, H. Ray Strong, and Danny Dolev. Atomic broadcast: From simple message diffusion to Byzantine agreement. Technical Report RJ5244, IBM Research Laboratory, San Jose, California, July 1986. An earlier version appeared in the 1985 Proceedings of the International Symposium on Fault-Tolerant Computing.

[CD90] Flaviu Cristian and Robert Dancey. Fault-tolerance in the advanced automation system. Technical Report RJ7424, IBM Research Laboratory, San Jose, California, April 1990.

[DSB86] M. Dubois, C. Scheurich, and F. Briggs. Memory access buffering in multiprocessors. In *Proceedings of the 13th Annual International Symposium on Computer Architecture*, pages 434–442, June 1986.

[Gre91] Donald Greenberg. Computers and architecture. *Scientific American*, 264(2):104–109, February 1991.

[JB89] Thomas Joseph and Kenneth Birman. Low cost management of replicated data in fault-tolerant distributed systems. *ACM Transactions on Computer Systems*, 4(1):54–70, February 1989.

[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.

[LL86] Barbara Liskov and Rivka Ladin. Highly-available distributed services and fault-tolerant distributed garbage collection. In *Proceedings of the Fifth ACM Symposium on Principles of Distributed Computing*, pages 29–39, Calgary, Alberta, August 1986. ACM SIGOPS-SIGACT.

[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.

[MB90] Messac Makpangou and Kenneth Birman. Designing application software in wide area network settings. Technical Report 90-1165, Department of Computer Science, Cornell University, 1990.

[MCWB91] Keith Marzullo, Robert Cooper, Mark Wood, and Kenneth Birman. Tools for distributed application management. *IEEE Computer*, August 1991.

[Pet87] Larry Peterson. Preserving context information in an ipc abstraction. In *Sixth Symposium on Reliability in Distributed Software and Database Systems*, pages 22–31. IEEE, March 1987.

[RAA+88] M. Rozier, V. Abrossimov, M. Armand, F. Hermann, C. Kaiser, S. Langlois, P. Leonard, and W. Neuhauser. The chorus distributed system. *Computer Systems*, pages 299–328, Fall 1988.

[RB91] Aleta Ricciardi and Kenneth Birman. Using process groups to implement failure detection in asynchronous environments. In *Proceedings of the Eleventh ACM Symposium on Principles of Distributed Computing*, Montreal, Quebec, August 1991. ACM SIGOPS-SIGACT.

[SBM89] Alex Siegel, Kenneth Birman, and Keith Marzullo. Deceit: A flexible distributed file system. Technical Report 89-1042, Department of Computer Science, Cornell University, 1989.

[Sch86] Fred B. Schneider. The state machine approach: a tutorial. Technical Report TR 86-800, Department of Computer Science, Cornell University, December 1986. Revised June 1987.

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

[Ske82] Dale Skeen. *Crash recovery in a distributed database system*. PhD thesis, University of California at Berkeley, Department of EECS, June 1982.

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

[Tan88] Andrew Tanenbaum. *Computer Networks*. Prentice Hall, second edition, 1988.

[TH90] Josep Torrellas and John Hennessey. Estimating the performance advantages of relaxing consistency in a shared memory multiprocessor. Technical Report CSL-TN-90-365, Computer Systems Laboratory, Stanford University, February 1990.

[Woo91] Mark Wood. *Constructing reliable reactive systems.* PhD thesis, Cornell University, Department of Computer Science, December 1991.