The Process Group Approach to Reliable Distributed Computing *

Kenneth P. Birman

July 14, 1993

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 that are substantially easier to develop, exploit sophisticated forms of cooperative computation, and achieve high reliability. This paper reviews six years of research on ISIS, describing the model, its implementation challenges, and the types of applications to which ISIS has been applied.

1 Introduction

One might expect the reliability of a distributed system to follow directly from the reliability of its constituents, but this is not always the case. The mechanisms used to structure a distributed system and to implement cooperation between components play a vital role in determining thre reliability of the system. Many contemporary distributed operating systems have placed emphasis on communication performance, overlooking the need for tools to integrate components into a reliable whole. The communication primitives supported give generally reliable behavior, but exhibit problematic semantics when transient failures or system configuration changes occur. The resulting building blocks are, therefore, unsuitable for facilitating the construction of systems where reliability is important.

This paper reviews six years of research on ISIS, a system that provides tools to support the construction of reliable distributed software. The thesis underlying ISIS is that development of reliable distributed software can be simplified using *process groups* and *group programming tools*. This paper motivates the approach taken, surveys the system, and discusses our experience with real applications.

*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.

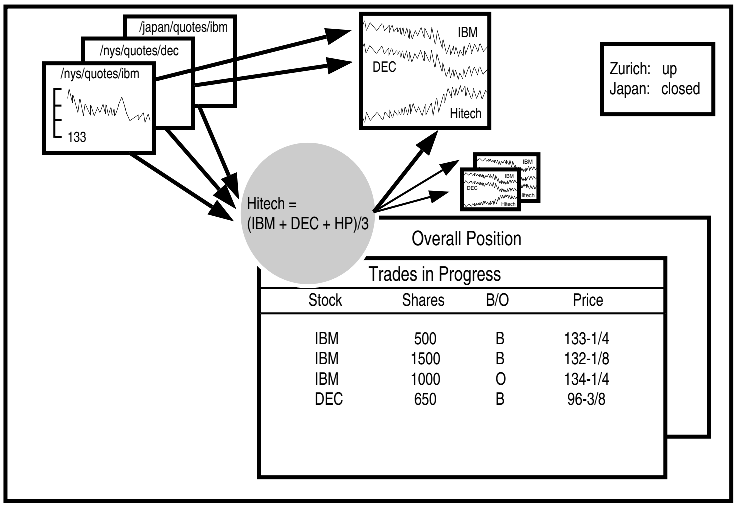

Figure 1: Broker's trading system

It will be helpful to illustrate group programming and ISIS in a setting where the system has found rapid acceptance: brokerage and trading systems. These systems integrate large numbers of demanding applications and require timely reaction to high volumes of pricing and trading information.[1] It is not uncommon for brokers to coordinate trading activities across multiple markets. Trading strategies rely 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 financial instruments. Any 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. A theme of the paper will be that all of these issues revolve around the technology used to "glue the system together". By endowing the corresponding software layer with predictable, fault-tolerant behavior, the flexibility and reliability of the overall system can be greatly enhanced.

Figure 1 illustrates 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. The display illustrates several points:

- *Information backplane.* The broker would construct such a display by interconnecting elementary widgets (graphical windows, computational widgets, etc.) so that the output of one becomes the input to another. Seen in the large, this implies the ability to *publish* messages and *subscribe* to messages

[1] Although this class of systems certainly demands high performance, there are no real-time deadlines or hard time constraints, such as in the FAA's Advanced Automation System [CD90]. This issue is discussed further in Sec. 7.

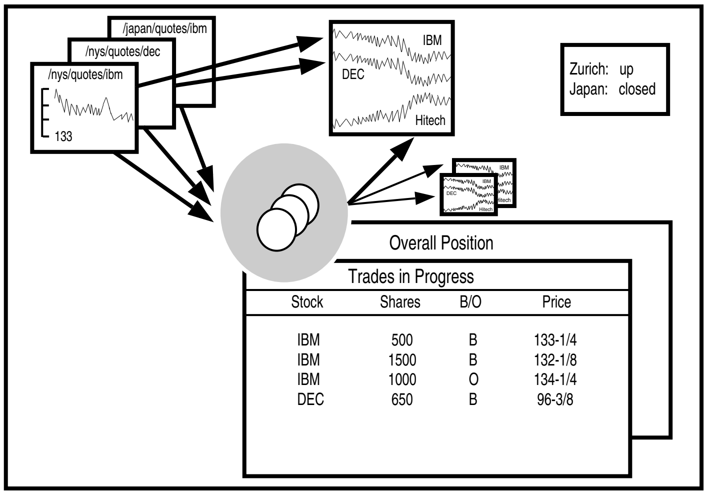

Figure 2: Making an analytic service fault-tolerant

sent from program to program on topics that make up the "corporate information backplane" of the brokerage. Such a backplane would support a naming structure, communication interfaces, access restrictions, and some sort of selective history mechanism. For example, upon subscribing to a topic, an application will often need key messages posted to that topic in the past.

- *Customization.* The display suggests that the system must be easily customized. The information backplane must be organized in a systematic way (so that the broker can easily track down the name of communication streams of interest) and flexible (allowing the introduction of new communication streams while the system is active).

- *Hierarchical structure.* Although the trader will treat the wide-area system in a seamless way, communication disruptions are far more common on wide-area links (say, from New York to Tokyo or Zurich) than on local-area ones. This gives the system a hierarchical structure composed of local area systems which are closely coupled and rich in services, interconnected by less reliable and higher latency wide-area communication links.

What about the reliability implications of such an architecture? In Fig. 1, the trader has introduced a computed index of technology stocks against the price of IBM, and it is easy to imagine that such customization could include computations critical to the trading strategy of the firm. In Figure 2, the analysis program 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 such as this would have to be introduced by the system – the trader's only action being to request it, perhaps by specifying the degree of reliability needed for this analytic program. This means the system must automatically replicate or checkpoint the computation, placing the replicas on processors that fail independently from the broker's workstation, and activating a backup if the primary fails.

The requirements of modern trading environments 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 underway in some remote and inaccessible region, 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. The software of a modern telecommunications switching product is faced with many of the same issues, as is software implementing a database that will be used in a large distributed setting. To build applications for the networked environments of the future, a technology is needed that will make it as easy to solve these sorts of problems as it is to build graphical interfaces today.

A central premise of the ISIS project, shared with several other efforts [LL86, CD90, Pet87, KTHB89, ADKM91], is that support for programming with *distributed groups of cooperating programs* is the key to solving problems such as the ones seen above. For example, a fault-tolerant data analysis service can be implemented by a group of programs that adapt transparently to failures and recoveries. The publication/subscription style of interaction involves an anonymous use of process groups: here, the group consists of a set of publishers and subscribers that vary dynamically as brokers change the instruments that they trade. Each interacts with the group through a group name (the topic), but the group membership is not tracked or used within the computation. Although the processes publishing or subscribing to a topic do not cooperate directly, when this structure is employed, the reliability of the application will depend on the reliability of group communication. It is easy to see how problems could arise if, for example, two brokers monitoring the same stock see different pricing information.

Process groups of various kinds arise naturally throughout a distributed system. Yet, current distributed computing environments provide 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. In short, contemporary distributed computing environments prevent 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 defines the group programming paradigm more carefully and discusses the algorithmic issues it raises. This leads into the ISIS computational model, called *virtual synchrony*. The next part discusses the tools from which ISIS users construct applications. The last part reviews 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 process group usage are seen in most ISIS applications:

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

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. If the sender and subscribers remain operational, messages should be delivered exactly once; if the sender fails, a message should be delivered to all or none of the subscribers. The application programmer should not need to worry about message loss or duplication.
3. Messages should be delivered to subscribers in some sensible order. For example, one would expect messages to be delivered in an order consistent with causal dependencies: if a message m is published by a program that first received $m_1 ... m_i$, then m might be dependent on these prior messages. If some other subscriber will receive m as well as one or more of these prior messages, one would expect them to be delivered first. Stronger ordering properties might also be desired, as discussed later.
4. It should be possible for a subscriber to obtain a history of the group – a log of key events and the order in which they were received.[2] If n messages are posted and the first message seen by a new subscriber will be message m_i, one would expect messages $m_1 ... m_{i-1}$ to be reflected in the history, and that messages $m_i ... m_n$ will all be delivered to the new process. If some messages are missing from the history, or included both in the history and in the subsequent postings, incorrect behavior might result.

- *Explicit groups:* A group is explicit when its members cooperate directly: they know themselves to be members of the group, and employ algorithms that employ the list of members, relative rankings within the list, or in which responsibility for responding to requests is shared. Explicit groups have additional needs stemming from their use of group membership information: in some sense, membership changes are among the information being published to an explicit group. 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, group 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 ranking the group members and then dividing the database into n parts, where n is the number of group members. Each member would do $1/n'th$ of the work, with the ranking determining which member handles which fragment of the database. The members need consistent views of the group membership to perform such a search correctly; otherwise, two processes might search the same part of the database while some other part remains unscanned, or they might partition the database inconsistently.

Thus, a number of technical problems must be considered in developing software for implementing distributed process groups:

[2]The application itself would distinguish messages that need to be retained from those that can be discarded.

- *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.

The first and last of these problems have received considerable study. However, the problems cited are not independent: their integration within a single framework is non-trivial. This integration issue underlies our virtual synchrony execution model.

3 Building distributed services over conventional technologies

In this section we review the technical issues raised above. In each case, we start by describing the problem as it might be approached by a developer working over a contemporary computing system, with no special tools for group programming. Obstacles to solving the problems are identified, and used to motivate a general approach to overcoming the problem in question. Where appropriate, we then comment on the actual approach used in solving the problem within ISIS.

3.1 Conventional message passing technologies

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

- *Unreliable datagrams*: These services automatically discard corrupted messages, but do little additional processing. Most messages 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 results from 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 many possible outcomes: 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. Standard stream protocols include TCP, the ISO protocols, and TP4. Because of pipelining, streams generally outperform RPC when an application sends large

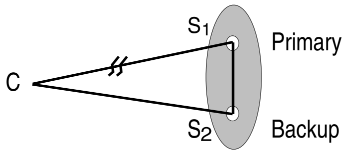

Figure 3: Inconsistent connection states

volumes of data. However, the standards also prescribe rules under which a stream will be broken, using conditions based on timeout or excessive retransmissions. For example, suppose that processes c, s_1 and s_2 have connections with one another – perhaps, s_1 and s_2 are the primary and backup, respectively, for a reliable service of which c is a client. Now, consider the state of this system if the connection from c to s_1 breaks due to a communication failure, while all three processes and the other two connections remain operational (figure 3). Much like the situation after a failed RPC, c and s_1 will now be uncertain regarding one-another's status. Worse, s_2 is totally unaware of the problem. In such a situation, the application may easily behave in an inconsistent manner. In our primary-backup example, c would cease sending requests to s_1, expecting s_2 to handle them. s_2, however, will not respond (it expects s_1 to do so). In a system with more components, the situation would be greatly exacerbated. From this, one sees that a reliable data stream has guarantees little stronger than an unreliable one: when channels break, it is not safe to infer that either endpoint has failed, channels may not break in a consistent manner, and data in transit may be lost. Because the conditions under which a stream break are defined by the standards, one has a situation in which potentially inconsistent behavior is unavoidable.

These considerations lead us to make a collection of assumptions about the network and message communication in the remainder of the paper. First, we will assume that the system is structured as a wide-area network (WAN) composed of local-area networks (LANs) interconnected by wide-area communication links. WAN issues will not be considered in this paper for reasons of brevity. We assume that each LAN consists of a collection of machines (as few as two or three, or as many as one or two hundred), connected by a collection of high speed, low latency communication devices. If shared memory is employed, we assume that it is not used over the network. Clocks are not assumed to be closely synchronized.

Within a LAN, we assume that messages may be lost in transit, arrive out of order, be duplicated, or be discarded because of inadequate buffering capacity. We also assume that LAN communication partitions are rare. The algorithms described below, and the ISIS system itself, may pause (or make progress in only the largest partition) during periods of partition failure, resuming normal operation only when normal communication is restored.

We will assume that the lowest levels of the system are responsible for flow control and for overcoming message loss and unordered delivery. In ISIS, these tasks are accomplished using a windowed acknowledgement protocol similar to the one used in TCP, but integrated with a failure detection subsystem. With this (non-standard) approach, a consistent system-wide view of the state of components in the system and of the state of communication channels between them can be presented to higher layers of software. For example, the ISIS transport layer will only break a communication channel to a process in situations where it would also report to any application monitoring that process that the process has failed. Moreover, if one channel to a process is broken, all channels are broken.

3.2 Failure model

Throughout this paper, processes and processors are assumed to fail by halting, without initiating erroneous actions or sending incorrect messages. This raises a problem: transient problems – such as an unresponsive swapping device or a temporary communication outage – can mimic halting failures. Because we will want to build systems guaranteed to make progrress when failures occur, this introduces a conflict between "accurate" and "timely" failure detection.

One way to overcome this problem, supported by ISIS, integrates the communication transport layer with the failure detection layer to make processes *appear* to fail by halting, even when this may not be the case: a *fail-stop* model [SS83]. To implement such a model, a system uses an agreement protocol to maintain a system membership list: only processes included in this list are permitted to participate in the system, and non-responsive or failed processes are dropped [Cri88, RB91]. If a process dropped from the list later resumes communication, the application is forced to either shut down gracefully or to run a "reconnection" protocol. The message transport layer plays an important role, both by breaking connections and by intercepting messages from faulty processes.

In the remainder of this paper we assume a message transport and failure-detection layer with the properties of the one used by ISIS. To summarize, a process starts execution by joining the system, interacts with it over a period of time during which messages are delivered in the order sent, without loss or duplication, and then terminates (if it terminates) by halting detectably. Once a process terminates, we will consider it to be permanently gone from the system, and assume that any state it may have recorded (say, on a disk) ceases to be relevant. If a process experiences a transient problem and then recovers and rejoins the system, it is treated as a completely new entity – no attempt is made to automatically reconcile the state of the system with its state prior to the failure (recovery of this nature is left to higher layers of the system and applications).

3.3 Building groups over conventional technologies

Group addressing

Consider 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* [BJ87, Cri88]. Such a service maintains a map from group identifiers to membership lists. Deferring fault-tolerance issues, one could implement such a service using 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.[3] 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.

This form of group addressing also raises a scheduling question. The designer of a distributed application will want to send messages to all members of the group, under some reasonable interpretation of the term "all". The question, then, is how to schedule the delivery of messages so that the delivery is to a reasonable set of processes. For example, suppose that a process group contains three processes, and a process sends many messages to it. One would expect these messages to reach all three members, not some other set reflecting a stale view of the group composition (e.g. including processes that have left the group, or omitting some of the current members).

The solution to this problem favored in our work can be understood by thinking of the group membership as data in a database shared by the sender of a multi-destination message (a *multicast*[4]), and the algorithm used to add new members to the group. A multicast "reads" the membership of the group to which it is sent, holding a form of read-lock until the delivery of the message occurs. A change of membership that adds a new member would be treated like a "write" operation, requiring a write-lock that prevents such an operation from executing while a prior multicast is underway. It will now appear that messages are delivered to groups only when the membership is not changing.

A problem with using locking to implement address expansion is cost. Accordingly, ISIS uses this idea, but does not employ a database or any sort of locking. And, rather than implement a membership server, which could represent a single point of failure, ISIS replicates knowledge of the membership among the members of the group itself. This is done in an integrated manner so as to perform address expansion with no extra messages or unnecessary delays and guarantee the logical instantaneity property that the user expects. For practical purposes, any message sent to a group can be thought of as reaching all of its members at the same time.

[3]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).

[4]In this paper the term *multicast* refers to sending a single message to the members of a process group. The term *broadcast*, common in the literature, is sometimes confused with the hardware broadcast capabilities of devices like Ethernet. While a multicast might make use of hardware broadcast, this would simply represent one possible implementation strategy.

Logical time and causal dependency

The phrase "reaching all of its members at the same time" raises an issue that will prove to be fundamental to message-delivery ordering. Such a statement presupposes a temporal model. What notion of time applies to distributed process group applications?

In 1978, Leslie Lamport published a seminal paper that considered the role of time in distributed algorithms [Lam78]. Lamport asked how one might assign timestamps to the events in a distributed system so as to correctly capture the order in which events occured. Real-time is not suitable for this: each machine will have its own clock, and clock synchronization is at best imprecise in distributed systems. Moreover, operating systems introduce unpredictable software delays, processor execution speeds can vary widely due to cache affinity effects, and scheduling is often unpredictable. These factors make it hard to compare timestamps assigned by different machines.

As an alternative, Lamport suggested, one could discuss distributed algorithms in terms of the dependencies between the events making up the system execution. For example, suppose that a process first sets some variable x to 3, and then sets $y = x$. The event corresponding to the latter operation would depend upon the former one – an example of a *local dependency*. Similarly, receiving a message depends upon sending it. This view of a system leads one to define the *potential causality* relationship between events in the system. It is the irreflexive transitive closure of the message send-receive relation and the local dependency relation for processes in the system. If event a happens before event b in a distributed system, the causality relation will capture this.

In Lamport's view of time, we would say that two events are concurrent iff they are not causally related: the issue is not whether they *actually* executed simultaneously in some run of the system, but whether the system was sensitive to their respective ordering. Given an execution of a system, there exists a large set of equivalent executions arrived at by rescheduling concurrent events while retaining the event ordering constraints represented by causality relation. The key observation is that *the causal event ordering captures all the essential ordering information needed to describe the execution:* any two physical executions with the same causal event ordering describe indistinguishable runs of the system.

Recall our use of the phrase "reaching all of its members at the same time". Lamport has suggested that for a system described in terms of a causal event ordering, any set of concurrent events, one per process, can be thought of as representing a logical instant in time. Thus, when we say that all members of a group receive a message at the same time, we mean that the message delivery events are concurrent and totally ordered with respect to group membership change events. Causal dependency provides the fundamental notion of time in a distributed system, and plays an important role in the remainder of this section.

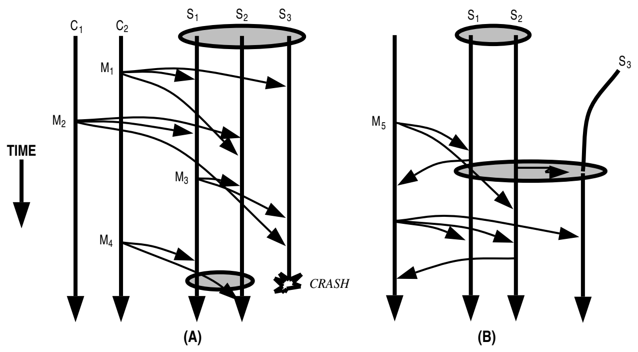

Figure 4: Message ordering problems

Message delivery ordering

Consider Figure 4-A, in which messages $m_1,\ m_2,\ m_3$ and m_4 are sent to a group consisting of processes $s_1,\ s_2$ and s_3. Messages m_1 and m_2 are sent concurrently and are received in different orders by s_2 and s_3. In many applications, s_2 and s_3 would behave in an uncoordinated or *inconsistent* manner if this occurred. A designer must, therefore, anticipate possible inconsistent message ordering. For example, one might design the application to tolerate such mixups, or explicitly prevent them from occurring by delaying the processing of m_1 and m_2 within the program until an ordering has been established. The real danger is that an designer could overlook the whole issue – after all, two simultaneous messages to the program that arrive in different orders may seem like an improbable scenario – yielding an application that usually is correct, but may exhibit abnormal behavior when unlikely sequences of events occur, or under periods of heavy load. (Under load, multicast delivery latencies rise, increasing the probability that concurrent multicasts could overlap).

This is only one of several delivery ordering problems illustrated in the Figure 4. Consider the situation when s_3 receives message m_3. Message m_3 was sent by s_1 after receiving m_2, and might refer to or depend upon m_2. For example, m_2 might authorize a certain broker to trade a particular 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_2 when m_3 is delivered. Perhaps m_2 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 to compensate when it occurs.

In our work on ISIS, this problem is solved by including a context record on each message. If a message arrives out of order, this record can be used to detect the condition, and to delay delivery until prior messages arrive. The context representation we employ has size linear in the number of members of the group within which the message is sent (actually, in the worst case a message might carry multiple such context records, but this is extremely rare). However, the average size can be greatly reduced by taking advantage of repetitious communication patterns, such as the tendency of a process that sends to a group to send multiple messages in succession [BSS91]. The imposed overhead is variable, but on the average small. Other solutions to this problem are described in [PBS89, BJ87].

Message m_4 exhibits a situation that combines several of these issues. m_4 is sent by a process that previously sent m_1 and is concurrent with m_2, m_3, and a membership change of the group. One sees here a situation in which all of the ordering issues cited thus far arise simultaneously, and in which failing to address any of them could lead to errors within an important class of applications. As shown, only the group addressing property proposed in the previous section is violated: were m_4 to trigger a concurrent database search, process s_1 would search the first third of the database, while s_2 searches the second *half* – one sixth of the database would not be searched. However, the figure could easily be changed to simultaneously violate other ordering properties.

State transfer

Figure 4-B illustrates a slightly different problem. Here, we wish to transfer the state of the service to process s_3: perhaps s_3 represents 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 messages received 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 s_3 receives a state which does not reflect message m_5, leaving it inconsistent with s_1 and s_2. Solving this problem involves a complex synchronization algorithm (we won't present it here), probably beyond the ability of a typical distributed applications programmer.

Fault tolerance

Up to now, our discussion has ignored failures. Failures cause many problems; here, we consider just one. Suppose that the sender of a message were to crash after some, but not all, destinations receive the message. The destinations that do have a copy will need to complete the transmission or discard the message. The protocol used should achieve "exactly-once delivery" of each message to those destinations that remain operational, with bounded overhead and storage. On the other hand, we need not be concerned with delivery to a process that fails during the protocol, since such a process will never be heard from again (recall the fail-stop model).

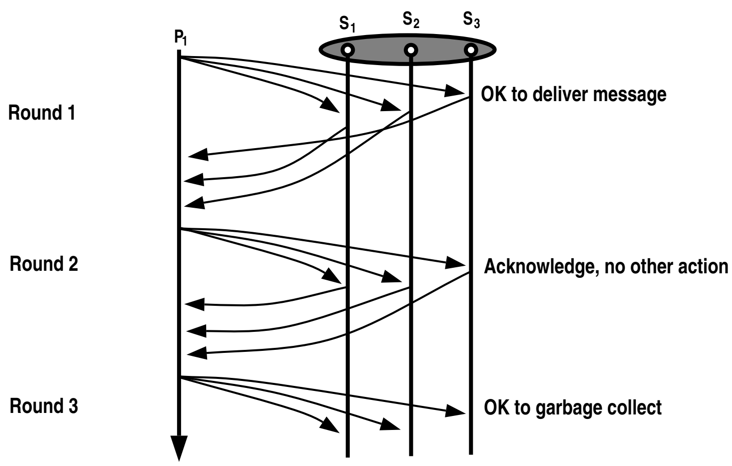

Figure 5: Three-round reliable multicast

Protocols to solve this problem can be complex, but a fairly simple solution will illustrate the basic techniques. This protocol uses three rounds of RPC's as illustrated in Figure 5. 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 the sender's behalf. In the second round, if no failure has occurred,then 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. 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.

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 duplicates and respond to them as they responded after the original reception. The protocol is straightforward, and we leave the details to the reader.

This three-round multicast protocol does not 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). These features make the protocol expensive. Much better solutions have been described in the literature (see [BSS91, BJ87] for more detail on the approach used in ISIS, and for a summary of other work in the area).

Recall that in Sec. 3.1, we indicated that system-wide agreement on membership was an important property of our overall approach. It is interesting to realize that a protocol such as this is greatly simplified because failures are reported consistently to all processes in the system. If failure detection were by an inconsistent mechanism it would be very difficult to convince oneself that the protocol is correct (indeed, as stated, the protocol could deliver duplicates if failures are reported inaccurately). The merit of solving such a problem at a low level is that we can then make use of the consistency properties of the solution when reasoning about protocols that react to failures.

Summary of issues

The above discussion pointed to some of the potential pitfalls that confront the developer of group software who works over a conventional operating system: (1) weak support for reliable communication, notably inconsistency in the situations in which channels break, (2) group address expansion, (3) delivery ordering for concurrent messages, (4) delivery ordering for sequences of related messages, (5) state transfer, and (6) 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 long compared to communication latencies, and that process states are volatile, hence we view these issues as beyond the scope of the current paper.[5] 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. This assertion goes beyond standards such as UNIX to include next-generation systems such as NT, Mach, Chorus and Amoeba.[6] A basic premise of this paper 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 a significant barrier to application developers. The only practical approach is to solve these problems in the distributed computing environment itself, or in the operating system. This permits a solution 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. If the implementation of process groups is left to the application designer, non-experts are unlikely to use the approach. The brokerage application of the introduction would be extremely difficult to build using the tools provided by a conventional operating system.

[5]These issues can be addressed within the tools layer of ISIS, and in fact the current system includes an optional subsystem for management of persistent data.

[6]In fairness, it should be noted that Mach IPC provides strong guarantees of reliability in its communication subsystem. However, Mach may experience unbounded delay when a node failure occurs. Chorus includes a port-group mechanism, but with weak semantics, patterned after earlier work on the V system [CZ83]. Amoeba, which initially lacked group support, has recently been extended to a mechanism apparently motivated by our work on ISIS [KTHB89].

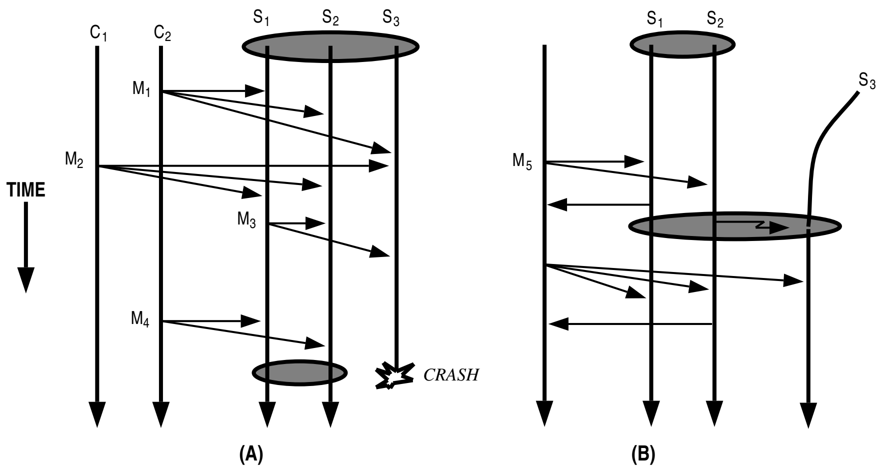

Figure 6: Closely synchronous execution

4 Virtual synchrony

Earlier, it was observed that integration of multiple group programming mechanisms into a single environment is also an important problem. Our work addresses this issue through an execution model called *virtual synchrony*, motivated by prior work on transaction serializability. 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 virtual synchrony model. A comparison of our work with the serializability model appears in Sec. 7.

The basic idea is to encourage programmers to assume a closely synchronized style of distributed execution [BJ89, Sch88]:

- Execution of a process consists of a sequence of events, which may be internal computation, message transmissions, message deliveries, or changes to the membership of groups that it creates or joins.
- 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 receive the same multicasts or observe the same group membership changes see the corresponding local events in the same relative order.
- A multicast to a process group is delivered to its full membership. The send and delivery events are considered to occur as a single, instantaneous event.

Close synchrony is a powerful guarantee. In fact, as seen in Fig. 6, it eliminates all the problems identified in the preceding section:

- *Weak communication reliability guarantees:* A closely synchronous communication subsystem appears to the programmer as completely reliable.
- *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.
- *Delivery ordering for concurrent messages*: In a closely synchronous execution, concurrently issued multicasts are distinct events. They would, therefore, be seen in the same order by any destinations that they have in common.
- *Delivery ordering for sequences of related messages*: In Figure 6a, process s_1 sent message m_3 after receiving m_2 hence m_3 may be causally dependent upon m_2. Processes executing in a closely synchronous model would never see anything inconsistent with this causal dependency relation.
- *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.
- *Failure atomicity*: The close synchrony model treats a multicast as a single logical event, and reports failures through group membership changes which are ordered with respect to multicast. The all or nothing behavior of an atomic multicast is thus implied by the model.

Unfortunately, although closely synchronous execution simplifies distributed application design, the approach cannot be applied directly in a practical setting. First, achieving close synchrony is impossible in the presense of failures. Say that processes s_1 and s_2 are in group G and message m is multicast to G. Consider s_1 at the instant before it delivers m. According to the close synchrony model, it can only deliver m if s_2 will do so also. But, s_1 has no way to be sure that s_2 is still operational, hence s_1 will be unable to make progress [TS92]. Fortunately, we can finesse this issue: if s_2 has failed, it will hardly be in a position to dispute the assertion that m was delivered to it first!

A second concern is that maintaining close synchrony is expensive. The simplicity of the approach stems in part from the fact that the entire process group advances in lock step. But, this also means that the rate of progress each group member can make is limited by the speed of the other members, and this could have a huge performance impact. What is needed is a model with the conceptual simplicity of close synchrony, but that is capable of efficiently supporting very high throughput applications.

In distributed systems, high throughput 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 production exceeds the rate of consumption, or when the sender needs to wait for a reply or some other input (Figure 7). 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

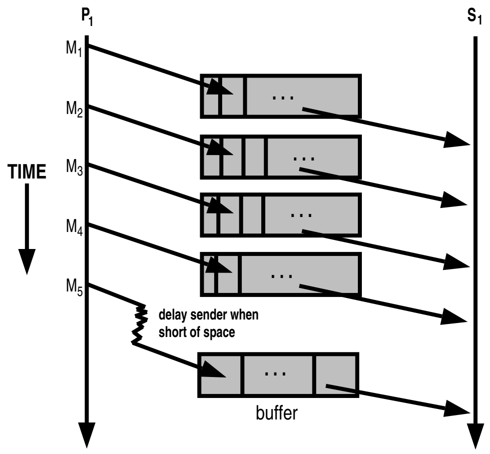

Figure 7: Asynchronous pipelining

manner, permitting both the sender and destination to remain continuously active. Closely synchronous execution precludes such pipelining, delaying execution of the sender until the message can be delivered.

This motivates the virtual synchrony approach. A virtually synchronous system permits asynchronous executions for which there exists some closely synchronous execution indistinguishable from the asynchronous one. In general, this means that for each application, events need be synchronized only to the degree that the application is sensitive to event ordering. In some situations, this approach will be identical to close synchrony. In others, it is possible to deliver messages in different orders at different processes, without the application noticing. When such a relaxation of order is permissable, a more asynchronous execution results.

Order sensitivity in distributed systems.

We are led to a final technical question: "when can synchronization be relaxed in a virtually synchronous distributed system?" Two forms of ordering turn out to be useful; one is "stronger" than the other, but also more costly to support.

Consider a system with two processes, s_1 and s_2, sending messages into a group G with members g_1 and g_2. s_1 sends message m_1 to G and, concurrently, s_2 sends m_2. In a closely synchronous system, g_1 and g_2 would receive these messages in identical orders. If, for example, the messages caused updates to a data structure replicated within the group, this property could be used to ensure that the replicas remain identical through the execution of the system. A multicast with this property is said to achieve an *atomic delivery ordering*, and is denoted ABCAST. ABCAST is an easy primitive to work with, but costly to implement. This cost stems from the following consideration. An ABCAST message can only be delivered when it is known that no prior ABCAST remains undelivered. This introduces latency: messages m_1 and m_2 must be delayed

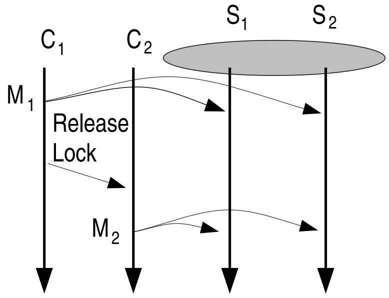

Figure 8: Causal ordering

before they can be delivered to g_1 and g_2. Such a delivery latency may not be visible to the application. But, in cases where s_1 and s_2 need responses from g_1 and/or g_2, or where the senders and destinations are the same, the application will experience a significant delay each time an ABCAST is sent. The latencies involved can be very high, depending on how the ABCAST protocol is engineered.

Not all applications require such a strong, costly, delivery ordering. Concurrent systems often use some form of synchronization or mutual exclusion mechanism to ensure that conflicting operations are performed in some order. In a parallel shared–memory environment, this is normally done using semaphores around critical sections of code. In a distributed system, it would normally be done by using some form of locking or token passing.

Consider such a distributed system, having that property that two messages can be sent concurrently to the same group *only when their effects on the group are independent*. In our example above, either s_1 and s_2 would be prevented from sending concurrently (i.e. if m_1 and m_2 have potentially conflicting effects on the states of the members of G), or if they are permitted to send concurrently, the delivery orders could be arbitrarily interleaved, because the actions on receiving such messages commute.

It might seem that the degree of delivery ordering needed would be *first in, first out*, or FIFO. However, this is not quite right, as illustrated in figure 8. Here, we see a situation in which s_1, holding mutual exclusion, sends message m_1, but then releases its mutual exclusion lock to s_2, which sends m_2. Perhaps, m_1 and m_2 are updates to the same data item; the order of delivery could therefore be quite important. Although there is certainly a sense in which m_1 was sent "first", notice that a FIFO delivery order would not enforce the desired ordering, since FIFO order is usually defined for a (sender,destination) pair, and here we have two senders. The ordering property needed for this example is that if m_1 causally precedes m_2, then m_1 should be delivered before m_2 at shared destinations, corresponding to a multicast primitive denoted 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.[7]

[7]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 a total one: it is a *causal atomic* multicast primitive. An argument can be made that an ABCAST protocol that is not causal cannot be used asynchronously, hence we see strong reasons for implementing ABCAST in this manner.

The major advantage of CBCAST over ABCAST is that it is not subject to the type of latency cited above. A CBCAST message can be delivered as soon as any prior messages have been delivered, and all the information needed to determine whether any prior messages are outstanding can be included, at low overhead, on the CBCAST message itself. Except in unusual cases where a prior message is somehow delayed in the network, a CBCAST message will be delivered immediately upon reception.

The ability to use a protocol such as CBCAST is highly dependent on the nature of the application. Some applications have a mutual-exclusion structure for which causal delivery ordering is adequate, while others would need to introduce a form of locking to be able to use CBCAST instead of ABCAST. Basically, CBCAST can be used when any conflicting multicasts are uniquely ordered along a single causal chain. In this case, the CBCAST guarantee is strong enough to ensure that all the conflicting multicasts are seen in the same order by all recipients – specifically, the causal dependency order. Such an execution system is virtually synchronous, since the outcome of the execution is the same as if an atomic delivery order had been used.

The CBCAST communication pattern arises most often in a process group that manages replicated (or coherently cached) data using locks to order updates. Processes that update such data first acquire the lock, then issue a stream of asynchronous updates, and then release the lock. There will generally be one update lock for each class of related data items, so that acquisition of the update lock rules out conflicting updates.[8] However, mutual exclusion can sometimes be inferred from other properties of an algorithm, hence such a pattern may arise even without an explicit locking stage. By using CBCAST for this communication, an efficient, pipelined data flow is achieved. In particular, there will be no need to block the sender of a multicast, even momentarily, unless the group membership is changing at the time the message is sent.

The tremendous performance advantage of CBCAST over ABCAST may not be immediately evident. However, when one considers how fast modern processors are in comparison with communication devices, it should be clear that any primitive that unnecessarily waits before delivering a message could introduce substantial overhead. For example, it is common for an application that replicates a table of pending requests within a group to multicast each new request, so that all members can maintain identical copies of the table. In such cases, if the way that a request is handled is sensitive to the contents of the table, the sender of the multicast must wait until the multicast is delivered before acting on the request. Using ABCAST the sender will need to wait until the delivery order can be determined. Using CBCAST, the update can be issued asynchronously, and applied immediately to the copy maintained by the sender. The sender thus avoids a potentially long delay, and can immediately continue computation or reply to the request. When a sender generates bursts of updates, also a common pattern, the advantage of CBCAST over ABCAST is even greater, because multiple messages can be buffered and sent in one packet, giving a pipelining effect.

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

[8] In ISIS applications, locks are used primarily for mutual exclusion on possibly conflicting operations, such as updates on related data items. In the case of replicated data, this results in an algorithm similar to a primary copy update in which the "primary" copy changes dynamically. The execution model is non-transactional, and there is no need for read-locks or for a two-phase locking rule. This is discussed further in Sec. 7.

communication subsystem [Bir85], the approach draws on Lamport's prior work on logical notions of time. Moreover, the approach was in some respects anticipated by work on primary copy replication in database systems [BHG87]. Similarly, close synchrony is related both to Lamport and Schneider's *state machine approach* to developing distributed software [Sch90] and to the database serializability model, discussed further below. Work on parallel processor architectures has yielded a memory update model called *weak consistency* [DSB86, TH90], which uses a causal dependency 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 the conditions under which CBCAST can be used in place of ABCAST appears in [Sch88, BJ89].

4.1 Summary of benefits due to virtual synchrony

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 model:

- Allows code to be developed assuming a simplified, closely synchronous execution model.
- Supports a meaningful notion of group state and state transfer, both when groups manage replicated data, and when a computation is dynamically partitioned among group members.
- Asynchronous, pipelined communication.
- Treatment of communication, process group membership changes and failures through a single, event-oriented execution model.
- Failure handling through a consistently presented system membership list integrated with the communication subsystem. This is in contrast to the usual approach of sensing failures through timeouts and broken channels, which does not guarantee consistency.

The approach also has limitations:

- Reduced availability during LAN partition failures: only allows progress in a single partition, and hence tolerates at most $\lfloor n/2 \rfloor - 1$ simultaneous failures, if n is the number of sites currently operational.
- Risks incorrectly classifying an operational site or process as faulty.

The virtual synchrony model is unusual in offering these benefits within a single framework. Moreover, theoretical arguments exist that no system that provides consistent distributed behavior can completely evade these limitations. Our experience has been that the issues addressed by virtual synchrony are encountered in even the simplest distributed applications, and that the approach is general, complete, and theoretically sound.

5 The Isis Toolkit

The ISIS toolkit provides a collection of higher-level mechanisms for forming and managing process groups and implementing group-based software. This section illustrates the specifics of the approach by discussing the styles of process group supported by the system and giving a simple example of a distributed database application.

ISIS is not the first system to use process groups as a programming tool: at the time the system was initially developed, Cheriton's V system had received wide visibility [CZ83]. More recently, group mechanisms have become common, exemplified by the Amoeba system [KTHB89], the Chorus operating system [RAA$^+$88], the Psync system [PBS89], a high availability system developed by Ladin and Liskov [LLS90], IBM's AAS system [CD90], and Transis [ADKM91]. Nonetheless, ISIS was first to propose the virtual synchrony model and to offer high performance, consistent solutions to a wide variety of problems through its toolkit. The approach is now gaining wide acceptance.[9]

5.1 Styles of groups

The efficiency of a distributed system is limited by the information available to the protocols employed for communication. This was a 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 ISIS application interface introduces four styles of process groups that differ in how processes interact with the group, illustrated in Fig. 9 (anonymous groups are not distinguished from explicit groups at this level of the system). ISIS is optimized to detect and handle each of these cases efficiently.

Peer groups: These arise where a set of processes cooperate closely, for example to replicate data. The membership is often used as an input to the algorithm used in handling requests, as for the concurrent database search described earlier.

Client-server groups: In ISIS, any process can communicate with any group given the group's name and appropriate permissions. However, if a non-member of a group will multicast to it repeatedly, better performance is obtained by first registering the sender as a *client* of the group; this permits the system to optimize the group addressing protocol.

Diffusion groups: A *diffusion* group is a client-server group in which the clients register themselves but in which the members of the group send messages to the full client set and the clients are passive sinks.

[9] At the time of this writing our group is working with the Open Software Foundation on integration of a new version of the technology into Mach (the OSF 1/AD version) and with Unix International, which plans a reliable group mechanism for UI Atlas.

Hierarchical groups: A hierarchical group is a structure built from multiple component groups, for reasons of scale. Applications that use the hierarchical group initially contact its *root* group, but are subsequently redirected to one of the constituent "subgroups". Group data would normally be partitioned among the subgroups. Although tools are provided for multicasting to the full membership of the hierarchy, the most common communication pattern involves interaction between a client and the members of some subgroup.

There is no requirement that the members of a group be identical, or even coded in the same language or executed on the same architecture. Moreover, multiple groups can be overlapped and an individual process can belong to as many as several hundred different groups, although this is uncommon. Scaling is discussed further below.

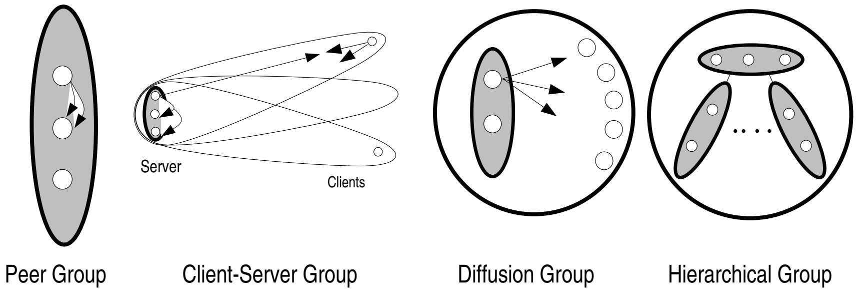

Figure 9: Styles of groups

5.2 The toolkit interface

As noted earlier, the performance of a distributed system is often limited by the degree of communication pipelining achieved. The development of asynchronous solutions to distributed problems can be tricky, and many ISIS users would rather employ less efficient solutions 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, a tool for fault-tolerant primary-backup server design that load-balances by making different group members act as the primary for different requests, and so forth (a partial list appears in Table I. 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.

- *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. Deadlock detection or avoidance must be addressed at the application level. 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 a site recovers, programs automatically restart. For the first site to recover, group state is restored from logs (or initialized by software). For other sites, a process group join and transfer state is initiated.
- *WAN communication:* Reliable long-haul message passing and file transfer facility.

Table I: ISIS tools at process group level

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 the C programming language. 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). The formatted-I/O style of message generation and scanning is specific to the C interface, where type information is not available at runtime.

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 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 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 and ignore all 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 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, a condition readily detected in ISIS.

ISIS interfaces have been developed for C, C++, Fortran, Common Lisp, Ada and Smalltalk, and ports of ISIS exist for UNIX-workstations and mainframes from all major vendors, as well as for Mach, Chorus, ISC and SCO UNIX, the DEC VMS system, and Honeywell's Lynx OS. Data within messages is represented in the binary format used by the sending machine, and converted to the format of the destination upon reception (if necessary), automatically and transparently.

```
#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", name);
        salary = get_salary(name);
        reply(mp, "%d", salary);
  }
send_state()
  {
        struct sdb_entry *sp;
        for(sp = head(sdb); sp != tail(sdb); 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;

/* Looks-up database and registers as a client (for better performance) */
main()
  {
        isis_init(0);
        server = pg_lookup("/demos/salaries");
        pg_client(server);
        ...
  }
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 client of the simple database service

6 Who uses Isis, and how?

This section briefly reviews several ISIS applications, looking at the roles that ISIS plays.

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, now seen from an internal perspective in which groups underlying the services employed by the broker become evident. The architecture is a client-server one, in which the servers 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 is not 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

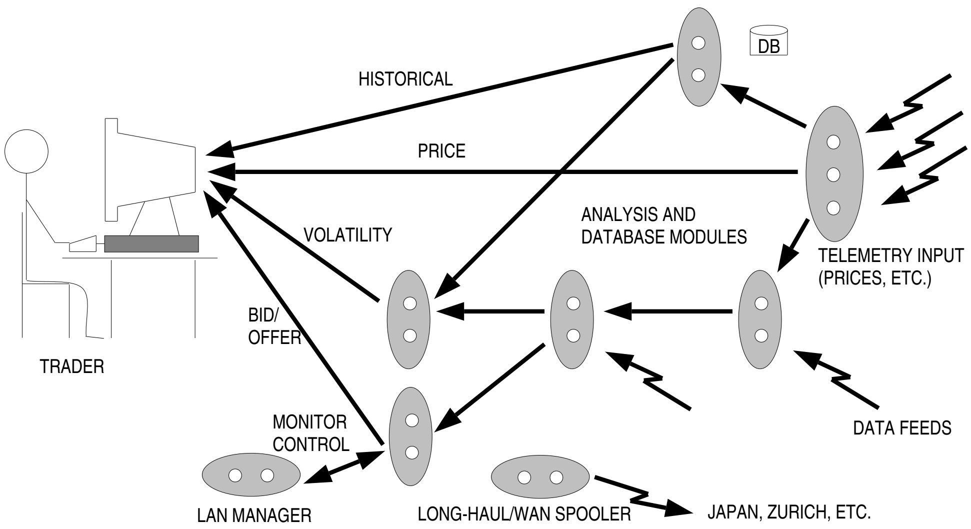

Figure 12: Process group architecture of brokerage system

rebooting, a file or database manager will be able to recover local data files 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 significant theme in financial computing is use of a subscription/publication style. 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 aspect 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 Database replication and database triggers

Although the ISIS computation model differs from a transactional model (see also Sec. 7), ISIS is useful in constructing distributed database applications. In fact, as many as half of the applications with which we are familiar are concerned with this problem.

Typical uses of ISIS in database applications focus on replicating a database for fault-tolerance or to support concurrent searches for improved performance. In such an architecture, the database system need not be aware that ISIS is present. Database clients access the database through a layer of software that multicasts updates (using ABCAST) to the set of servers, while issuing queries directly to the least loaded server. The servers are supervised by a process group that informs clients of load changes in the server pool, and supervises the restart of a failed server from a checkpoint and log of subsequent updates. It is interesting to realize that even such an unsophisticated approach to database replication addresses a widely perceived need among database users. In the long run, of course, comprehensive support for applications such as this would require extending ISIS to support a transactional execution model and to implement the XA/XOpen standards.

Beyond database replication, ISIS users have developed WAN databases by placing a local database system on each LAN in a WAN system. By monitoring the update traffic on a LAN, updates of importance to remote users can be intercepted and distributed through the ISIS WAN architecture. On each LAN, a server monitors for incoming updates and applies them to the database server as necessary. To avoid a costly concurrency control problem, developers of applications such as these normally partition the database so that the data associated with each LAN is directly updated only from within that LAN. On remote LAN's, such data can only be queried and could be stale, but this is still sufficient for many applications.

A final use of ISIS in database settings is to implement database *triggers*. A trigger is a query that is incrementally evaluated against the database as updates occur, causing some action immediately if a specified condition becomes true. For example, a broker might request that an alarm be sounded if the risk associated with a financial position exceeds some threshold. As data enters the financial database maintained by the brokerage, such a query would be evaluated repeatedly. The role of ISIS is in providing tools for reliably notifying applications when such a trigger becomes enabled, and for developing programs capable of taking the desired actions despite failures.

6.3 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 for job placement; if several machines are suitable for a given job, a criteria based on load and available memory is used to select one (this criteria can readily be changed). 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* is an example of a distributed application program that uses NMGR for job placement: it compiles applications by farming out compilation subtasks to compatible machines.

- DECEIT: This system [SBM89] provides fault-tolerant NFS-compatible file storage. Files are replicated both to increase performance (by supporting parallel reads on different replicas) and for 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, Woo91]. 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. User-defined sensors and actuators extend this initial set.

  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. Sensors can be aggregated, for example by taking the average load on the servers that manage a replicated database. 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 imbedded into the UNIX CSH command interpretor. At runtime, LOMITA control statements are expanded into distributed finite state machines triggered by events that can be sensed local to a sensor or system component; a process group is used to implement aggregates, perform these state transitions, and to notify applications when a monitored condition arises.

- 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.

6.4 Other Isis applications

Although this section covered a variety of ISIS applications, brevity precludes a systematic review of the full range of software that has been developed over the system. In addition to the problems cited above, ISIS has been applied to telecommunications switching and "intelligent networking" applications, military systems, such as a proposed replacement for the AEGIS aircraft tracking and combat engagement system, medical systems, graphics and virtual reality applications, seismology, factory automation and production control, reliable management and resource scheduling for shared computing facilities, and a wide-area weather prediction and storm tracking system [Joh93, Tho90, ASC92]. ISIS has also proved popular for scientific computing at laboratories such as CERN and Los Alamos, and has been applied to such problems as a programming environment for automatically introducing parallelism into data-flow applications (Paralex [BAA+92], a beam focusing system for a particle accelerator, a weather-simulation that combines a highly parallel ocean model with a vectorized atmospheric model and displays output on advanced graphics workstations, and resource management software for shared supercomputing facilities.

It should also be noted that although the paper has focused on LAN issues, ISIS also supports a WAN architecture and has been used in WANs composed of up to ten LANs.[10] Many of the applications cited above are structured as LAN solutions interconnected by a reliable, but less responsive, WAN layer.

7 Isis and other distributed computing 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, CASD85], which also uses a process-group based computing model. Similarly, one might wonder how the ISIS execution model compares with transactional database execution models. Unfortunately, these are complex issues, and it would be difficult to do justice to them without a lengthy digression. Briefly, a technology like the one used in AAS differs from ISIS in providing strong real-time guarantees provided that timing assumptions are respected. This is done by measuring timing properties of the network, hardware, and scheduler on which the system runs and designing protocols to have highly predictable behavior. Given such information about

[10]The WAN architecture of ISIS is similar to the LAN structure, but because WAN partitions are more common, encourages a more asynchronous programming style. WAN communication and link state is logged to disk files (unlike LAN communication), which enables ISIS to retransmit messages lost when a WAN partition occurs and to suppress duplicate messages. WAN issues are discussed in more detail in [MB90].

the environment, one could undertake a similar analysis of the ISIS protocols, although we have not done so. As noted earlier, experience suggests that ISIS is fast enough for even very demanding applications.[11]

The relationship between ISIS and transactional systems originates in the fact that both virtual synchrony and transactional serializability are order-based execution models [BHG87]. However, where the "tools" offered by a database system focus on isolation of concurrent transactions from one another, persistent data and rollback (abort) mechanisms, those offered in ISIS are concerned with direct cooperation between members of groups, failure handling, and ensuring that a system can dynamically reconfigure itself to make forward progress when partial failures occur. Persistency of data is a big issue in database systems, but much less so in ISIS. For example, the commit problem is a form of reliable multicast, but a commit implies serializability and permanence of the transaction being committed, while delivery of a multicast in ISIS provides much weaker guarantees.

8 Conclusions

We have argued that the next generation of distributed computing systems will benefit from support for process groups and group programming. Arriving at an appropriate semantics for a process group mechanism is a difficult problem, and implementing those semantics would exceed the abilities of many distributed application developers. 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:

- 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.
- Non-experts find the resulting system relatively easy to use.

[11] It should be noted that a process that experiences a timing fault in the protocols on which the AAS was originally based could receive messages that other processes reject, or reject messages others accept, because the criteria for accepting or rejecting a message uses the value of the local clock [CASD85]. This can lead to consistency violations. Moreover, if a fault is transient (e.g. the clock is subsequently resynchronized with other clocks), the inconsistency of such a process could spread if it initiates new multicasts, which other processes will accept. However, this problem can be overcome by changing the protocol, and the author understands this to have been done as part of the implementation of the AAS system.

This paper is being written as the first phase of the ISIS effort approaches its conclusion. We feel that the initial system has demonstrated the feasibility of a new style of distributed computing. As reported in [BSS91], ISIS achieves levels of performance comparable to those afforded by standard technologies (RPC and streams) on the same platforms. Looking to the future, we are now developing an ISIS "microkernel" suitable for integration into next-generation operating systems such as Mach, NT, and Chorus. This new system will also incorporate a security architecture [RBG92] and a real time communication suite. The programming model, however, will be unchanged.

Process group programming could ignite a wave of advances in reliable distributed computing, and of applications that operate on distributed platforms. 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. By removing these unnecessary obstacles, a vast groundswell of reliable distributed application development can be unleashed.

9 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. Thanks are due to: Ozalp Babaoglu, Micah Beck, Tim Clark, Robert Cooper, Brad Glade, Barry Gleeson, Holger Herzog, Guerney Hunt, Tommy Joseph, Ken Kane, Cliff Krumvieda, Jacob Levy, Messac Makpangou, Keith Marzullo, Mike Reiter, Aleta Ricciardi, Fred Schneider, Andre Schiper, Frank Schmuck, Stefan Sharkansky, Alex Siegel, Don Smith, Pat Stephenson, Robbert van Renesse, and Mark Wood. In addition, the author also gratefully acknowledges the help of Mauren Robinson, who prepared the figures for this paper, and the anonymous referees, whose careful and constructive comments on an initial version of this paper lead to a substantial improvements in the presentation.

References

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

[ADKM91] Yair Amir, Danny Dolev, Shlomo Kramer, and Dalia Malki. Transis: A communication subsystem for high availability. Technical Report TR 91-13, Computer Science Department, The Hebrew University of Jerusalem, November 1991.

[ASC92] T. Anthony Allen, William Sheppard, and Steve Condon. Imis: A distributed query and report formatting system. In *Proceedings of the SUN Users Group Meeting*, pages 94–101. Sun Microsystems Inc., 1992.

[BAA$^+$92] Ö. Babaoğlu, L. Alvisi, S. Amoroso, R. Davoli, and L.A. Giachini. Paralex: An environment for parallel programming distributed systems. In *Proceedings of the 6th ACM International Conference on Supercomputing*, pages 178–187. IEEE, July 1992.

[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.

[BJ87] Kenneth P. Birman and Thomas A. Joseph. Exploiting virtual synchrony in distributed systems. In *Proceedings of the Eleventh ACM Symposium on Operating Systems Principles*, pages 123–138, Austin, Texas, November 1987. 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.

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

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

[Cri88] Flaviu Cristian. Reaching agreement on processor group membership in synchronous distributed systems. Technical Report RJ5964, IBM Research Laboratory, March 1988.

[CZ83] David Cheriton and Willy Zwaenepoel. The distributed V kernel and its performance for diskless workstations. In *Proceedings of the Ninth ACM Symposium on Operating Systems Principles*, pages 129–140, Bretton Woods, New Hampshire, October 1983. ACM SIGOPS.

[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.

[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.

[Joh93] Dag Johansen. Stormcast: Yet another exercise in distributed computing. In Dag Johansen and Frances Brazier, editors, *Distributed Open Systems in Perspective*. IEEE, New York, 1993.

[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.

[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.

[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.

[RBG92] Michael Reiter, Kenneth P. Birman, and Li Gong. Integrating security in a group oriented distributed system. In *Proceedings of the IEEE Symposium on Research in Security and Privacy*, pages 18–32, May 1992.

[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.

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

[Sch90] Fred B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys*, 22(4):299–319, December 1990.

[SS83] Richard D. Schlichting and Fred B. Schneider. Fail-stop processors: an approach to designing fault-tolerant computing systems. *ACM Transactions on Computer Systems*, 1(3):222–238, August 1983.

[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.

[Tho90] Thomas C. Bache *et. al*. The intelligent monitoring system. *Bulletin of the Seismological Society of America*, 80(6):59–77, December 1990.

[TS92] John Turek and Dennis Shasha. The many faces of Consensus in distributed systems. *IEEE Computer*, 25(6):8–17, 1992.

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