Exploiting Replication*

Kenneth P. Birman
Thomas A. Joseph

TR 88-917
June 1988

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

* This work was supported by the Defense Advanced Research Projects Agency (DoD) under ARPA order 6037. Contract N00140-87-C-8904, and also by a grant from the Siemens Corporation. The views, opinions, and findings contained in this report are those of the authors and should not be construed as an official Department of Defense position, policy, or decision.

Exploiting Replication*

Kenneth P. Birman
Thomas A. Joseph

Department of Computer Science
Cornell University
Ithaca, NY 14853

June 1, 1988

This is a preprint of material that will appear in the collected lecture notes from *Arctic 88, An Advanced Course on Operating Systems*, Trömso, Norway, July 5-14, 1988. The lecture notes will appear in book form later this year.

* This work was supported by the Defense Advanced Research Projects Agency (DoD) under ARPA order 6037, Contract N00140-87-C-8904, and also by a grant from the Siemens Corporation. The views, opinions and findings contained in this report are those of the authors and should not be construed as an official Department of Defense position, policy, or decision.

Arctic 88 Course Outline

15

Exploiting replication in distributed systems

K. P. Birman and T. A. Joseph

15.1. Replication in directly distributed systems

This chapter examines techniques for replicating data and execution in what we will refer to as *directly distributed systems*: those in which multiple processes interact directly with one another while continuously respecting constraints on their joint behavior. Directly distributed systems are often required to solve difficult problems, ranging from management of replicated data to dynamic reconfiguration in response to failures. Here, we will see that these problems reduce to more primitive order-based consistency problems, which can be solved using primitives such as the reliable broadcast protocols discussed in Chapter 14. Moreover, given a system that implements reliable broadcast primitives, a flexible set of high-level "tools" can be provided for building a wide variety of directly distributed application programs.

15.1.1. Using replication to enhance availability and fault-tolerance

Replication is often central to solving distributed computing problems. For example, modularity and price-performance considerations argue for decentralization of software in factory automation settings. However, many factories contain devices controlled by dedicated processors that require realtime response. Any delay imposed on the controllers by the network must be bounded. In a system where data is not replicated or cached, this would be hard to guarantee because of possible packet loss and unpredictable load on remote servers. Distributed realtime systems thus need ways to replicate information that may be needed along time-critical paths.

Replication can be a powerful tool for solving other types of problems. For example, in a factory automation setting distributed *execution* may be used by applications that need to subdivide tasks by concurrently allocating multiple processes (or multiple devices) to a single piece of work. In order to distribute the execution of a single request over a set of high-speed processes, however, one must also replicate any information that they use to coordinate their actions. A centralized "coordinator" would represent a single point of failure and a potential performance bottleneck.

Fault-tolerance requirements are another major reason for replicating data. In a non-distributed setting a failure rarely affects anything but the user of the crashed program or machine. In a network, the effects of a crash can ripple through large numbers of machines. A program that will survive the failures of programs with which it interacts must have access to redundant copies of critical resources and ensure that its state is never dependent, even indirectly, on information to which only the failed program can access. It may also be necessary to maintain backup processes that will take over from a failed process and complete time-critical computations or computations that have acquired mutual exclusion on shared resources.

15.1.2. The tradeoff between shared memory and message passing

At the heart of any distributed system that distributes or replicates information is the problem of *transferring* information between cooperating processes. Broadly speaking, this can be done in one of two ways: by permitting the processes to interact with some common but passive resource or memory, or by supporting message exchange between them. There are advantages and disadvantages associated with each approach, hence the most appropriate style of information transfer for a particular problem must be determined by an analysis of the characteristics of that problem. For example, most database systems use the shared memory paradigm. In other settings, however, a shared resource might represent a bottleneck that could be avoided using replication and direct message-based interactions between the processes using that resource.

This point is important because the approach used to replicate data depends strongly on way in which processes will interact. For example, considerable recent work (Rashid et al., 1987) has been invested in the development of distributed virtual memory schemes, an approach introduced in the Apollo Domain operating system (Apollo, 1985). Synchronization in such systems is often based on transactional approaches, such as the database replication techniques described in Chapter 12. The shared memory approach to replication and synchronization thus leads to a whole school of thought concerning distributed program design and development.

As noted earlier, in this chapter we will be particularly interested in applications in which processes interact directly with one another and where the actions taken by one process may be explicitly coordinated with those taken by another process. The style of distributed programming needed to support this sort of application, and the most appropriate tools for implementing it, are substantially different than for the shared memory and transactional case.

Below, we start by identifying a set of characteristics of problems that call for direct interactions or cooperation between the processes that solve that problem. This characterization leads to a list of services that a directly distributed system may require. Next, we look at a number of systems in order to understand how they address the problems in this list. Finally, we examine a particular model for solving these problems in a message-passing environment and a set of solutions that can be easily understood in terms of this model.

15.1.3. Assumptions and limitations

Although this chapter explores a number of approaches to replication and distributed consistency, we do make some assumptions that limit the applicability of the treatment. Our model is intended to match a typical local area network or a loosely coupled multiprocessor. The programs and computers in such systems fail benignly, by crashing without sending out incorrect messages. Processors do not have synchronized clocks, hence the failure of an entire site can only be detected *unreliably*, using timeouts. Message communication is assumed to be reliable but bursty, because packets can be lost and may have to be retransmitted.

Two major problems that arise in LAN settings will not be considered here. The first is network partitioning, where the network splits into subnetworks between which communication is impaired (for example, if a LAN bridge fails). Providing replication that spans partitions is a difficult problem and an active research area. Secondly, we will not look at problems that place realtime constraints on distributed algorithms or protocols. Realtime issues are hard to isolate; once they are introduced, the entire system must often be treated from a realtime perspective. That is, although our methods are potentially useful in systems for which a realtime constraint leads the designer to dedicate a computer to some device, we will assume that the realtime aspects of such problems do not extend beyond the control program itself.

15.2. Consistent distributed behavior in distributed systems

When processes cooperate to implement some distributed behavior, an important issue is to ensure that their actions will be "mutually consistent". Not surprisingly, the precise meaning that one attaches to consistency has important implications throughout a distributed systems that presents coordinated behavior. As we have seen in previous chapters of this text, transactional serializability is a widely accepted form of consistency. In intuitive terms, a transactional system acts as if processes execute one by one, with each process modifying data objects in an atomic way that can be isolated from the actions taken by other processes. This leads to a natural question: should *all* types of distributed consistency be viewed as variant forms of transactional consistency, or are there problems that can only be addressed using other methods?

Looking at the factory automation setting, one finds that whereas shared shared memory problems fit well into the standard transactional framework, directly distributed problems generally do not. Consider the following two examples:

> • *Build software for monitoring job status and materials inventories. Updates will be done by the warehouses (quantities on hand), "cell controllers" (requests for materials and changes in job status), and from a central management site (changes in prices, deliveries from suppliers, changes in job priorities, etc). Queries will be done from managerial offices throughout the factory complex.*
>
> • *Develop software for a cell controller operating a set of drills. Each drill is independently controlled by a dedicated microprocessor. The cell as a whole receives a piece of work to do, together with a list of locations, sizes and tolerances for the holes to be drilled. It must efficiently schedule this work among the drills. Drills can go offline for maintenance or because of bits breaking, or come online while the cell is active, hence the scheduling problem is dynamic. Some drills are better suited to heavy low-precision work, while others are suitable for lighter high-precision work. Finally, it is critical that a hole not be drilled twice, even if a drill bit breaks before it is fully drilled, because this would result in a very low precision. Instead, an accurate list of partially drilled holes should be generated for a human technician to check and redrill manually.*

These two problems illustrate very different styles of distributed computing, and distributed consistency means something different for each. The former clearly lends itself to a transactional shared memory approach. One would configure the various programs into a "star", with a database at the center, perhaps replicated for fault-tolerance. Programs throughout the network interact through the database. Transactions are the natural consistency model for this setting. The essential observation to make is that the processes share data but are *independent*. By adopting a transactional style of interaction, they can avoid tripping over one another. Moreover, transactions provide a simple way to ensure that even if failures occur, the database remains intact and consistent.

Now consider the second problem. A star configuration seems much less natural here. The processes in a decentralized cell controller will need explicit knowledge of one another in order to coordinate their actions on a step-by-step basis. They need to reconfigure in response to events that can occur unpredictably, and to ensure the consistency of their views of the system state and one-another's individual states. When a control process comes online after being offline for a period of time, it will have to be reintergrated into the system, in a consistent way which may have very little to do with its state at the time of the failure. When a process goes offline, the processes that remain online need to assume responsibility for finishing any incomplete work and generating the list of holes to be manually checked. Moreover, it is not reasonable to talk about "aborting" partially completed work, since this could result in redrilling a hole.

What should consistency mean in problems like this? All of the above considerations run contrary to the spirit of a transactional approach, where the goal is serializability — *non*-interference between processes. A process in a transactional system is encouraged to run as if in isolation, whereas the cell controller involves explicit interactions and interdependencies between processes. Transactions use aborts and rollback to recover from possibly inconsistent states, but in this example rollback is physically impossible. On the other hand, although the kind of consistency here may not be transactional, one would not want to go to the extreme of concluding that there is no meaningful form of consistency that applies in this setting. Certainly, there should be a reasonable "explanation" for what each control process is doing, and this explanation should be in accordance with the cell controller specification. However, the explanation should be one that holds *continuously*, not just for "committed" operations as in case of transactions. That is, a set of drills that operate concurrently should behave in mutually consistent ways *at all times*.

This leaves us with two choices. One option is to look at how the transactional model could be extended to cover these new requirements. The idea of extending transactions is hardly a new one, and has previously led to mechanisms like *top-level* transactions[1] (Liskov *et al.*, 1987), mixtures of serializable and non-serializable behaviors (Herlihy, 1986a; Lynch, Blaustein, and Siegel, 1986), and specialized algorithms for concurrently accessing data structures like B-tree indexes. The trouble is that these introduce complexity into a model that was appealing for its simplicity. Moreover, these methods have been around for some time, and have proved appropriate only for a narrow set of problems. The second option, pursued here, is to develop a different style of distributed computation better matched to problems like the ones arising in a cell controller. The focus of this style of computation will be on enabling programs to reason consistently about one-another's states and actions.

15.3. A toolkit for directly distributed programming

One can think of a system that implements transactions as a collection of tools for solving problems involving shared data. These tools provide for synchronization, data access and update, transaction commit, and so forth. In this section, we approach the problem of building directly distributed software by postulating a set of tools for helping directly distributed processes to coordinate their actions. Later, we will examine a variety of systems, asking how close they come to solving these problems.

[1] A top-level transaction is essentially a way of sending a message from "within" the scope of an uncommitted transaction to other transactions running outside that scope. It provides an escape from the shared memory paradigm into the message passing one. The fact that such a mechanism is needed within transactional systems is strong evidence that no single approach addresses all types of distributed [illegible].

15.3.1. Components of the toolkit

What sorts of tools would the builder of a directly distributed system need? Although not exhaustive, the list of tools that follows is intended to be fairly extensive.

Process groups:
: A way to form an association between a set of processes cooperating to solve a problem.

Group communication:
: A location-transparent way to communicate with the members of a group or a list of groups and processes. In some systems, group communication consists only of a way to [illegible] some single member of the named group. In others, communication is broadcast-oriented[2] and *atomic*, meaning that all members of the destination group receive a given message unless a failure occurs, in which case either all the survivors receive it or none does. A problem that must be addressed is how group communication should work when the group membership is changing at the time the communication takes place. Should the broadcast be done before the change, after it, or is it acceptable for some group members to observe one ordering and some the other? Should message delivery to an unresponsive destination be retried indefinitely, or eventually interrupted — with the attendant risk that the destination was just experiencing a transient failure and is actually still operational? We will see that the way in which a system resolves these issues can [illegible]t the type of problems that process groups in the system can be used to solve.

Replicated data:
: A mechanism permitting group members to maintain replicated data. Most approaches provide a 1-copy consistency property, analogous to 1-copy serializability.

Synchronization:
: Facilities for synchronization of concurrent activities that interact through shared data or resources.

Distributed execution:
: Facilities for partitioning the work required to solve a problem among the members of a process group.

State monitoring mechanisms:
: Mechanisms for monitoring the state of the system and the membership of process groups, permitting processes to react to the failure of other group

[2] A group broadcast should not be confused with a hardware broadcast. A group broadcast provides a way to communicate with all members of some group. It might or might not make use of hardware facilities for broadcasting to all the machines connected to a local area network. Here, unless we explicitly indicate that we are talking about a hardware broadcast, the term broadcast will always mean broadcast to a group.

members.

Reconfiguration mechanisms:
: Facilities with which the system can adapt dynamically to failures, recoveries, and load changes that impact on work processing strategies.

Recovery mechanisms:
: Mechanisms for automating recovery, which could range from a way to restart services when a site reboots to facilities for reintegrating a component into an operational system that is actively engaged in distributed computations.

15.3.2. Consistency viewed as a tool

Let us return to the issue of consistency. In the context of a set of tools, a mechanism that provides for consistent behavior can also be understood as a sort of tool, but it is a more abstracted one than the sorts of "tools that do specific things" listed above. For example, in a shared memory setting consistent behavior generally means that the accesses made to the data by client processes are serializable (Bernstein and Goodman, 1981), and that some invariant holds on the state of programs themselves. Serializability is thus a tool for building transactional systems. In an directly distributed setting, we don't have a data manager or shared data items, hence the serializability constraint is lost. Nonetheless, one needs a way to establish that the processes in the system, taken as a group, satisfy some set of system-wide invariants in addition to local ones on their states.

Any notion of distributed consistency will be incomplete unless it takes into account the *asynchronous* nature of the systems in question. In particular, a definition of consistency based on respecting global properties or invariants must somehow take time into account. When one says that two actions taken at different locations are in accord with a global predicate, that statement will have no meaning until it is decided *when* the predicate should be evaluated. This temporal dependency is particularly striking if the notion of consistency changes while the system executes. Thus, consistent behavior in an idle cell controller is quite different from consistent behavior while work is present. Taking a more extreme example, consistent behavior of a distributed program for controlling a nuclear reaction means one thing during normal operation, but something entirely different if a cooling pump malfunctions. Since the switch from one rule to another cannot occur instantaneously, we need a notion of consistency that is simple, but at the same time "dynamic".

Distributed systems designers have approached the consistency issue in several ways. Much theoretical work starts with a rigorous notion of distributed consistency. However, this work often relies on simplified system models that may not correspond to real networks. For example, the theoretical study of Byzantine agreement establishes limits on the achievable behavior of a distributed agreement protocol. The failure modes permitted include malicious behaviors that real systems do not experience, and the model assumes that all processors share a common clock (so that they can run in lock-step). Unfortunately, however, real systems generally have multiple, independent processor clocks. Even if this were not the case, the cost of Byzantine agreement turns out to be very high. Similarly, innumerable papers have presented complex protocols to solve distributed problems, remarkably few of which have ever been implemented. Any practitioner who reads the literature discovers that many of these are in fact not "implementable" because they make unrealistic assumptions.

At the other extreme, most existing "distributed" operating systems provide little more than a message-passing mechanism, often available only through a cumbersome and inflexible communication subsystem. Systems like this simply abandon any rigorous form of consistency in favor of probabilistic behavioral statements. When attempts have been made to formally specify the behavior of real distributed systems, the results have often included so much detail that it becomes hard to separate the abstract behavior of the system from the implementation and interface it provides. Thus, a formal specification of a distributed system often includes details of how the message channels work, how addressing is handled, and so forth. While this information is unquestionably of value in designing applications that depend on a precise characterization of system behavior, high level issues such as "consistency" are obscured by such a treatment. As we will see, few of the problems in our list could be solved using a message-passing approach, and a highly detailed formalism describing exactly how the message-passing mechanism works offers little help.

An intermediate approach, which will be adopted here, restricts system behavior in order to simplify the solutions to problems like the ones that arise in the toolkit. On the one hand, these restrictions must be efficiently implementable. On the other, it must be possible to talk in abstract terms about how distributed programs execute in the system, what it means for them to behave consistently, and how consistency can be achieved. Specifically, given a distributed system, we would like to be able to describe its behavior formally in a way that will help establish the correctness of algorithms that run under it. If this requires restrictions on the permissible behavior of the system, we will need to understand how these restrictions can be enforced and how weak they can be made.

15.3.3. Other properties needed in a toolkit

More will be needed than a set of tools if the intention is to solve real-world distributed computing problems. Questions of methodology, efficiency of the implementation, and usability must also be addressed. For example, it is easy to solve database problems using transactions. To be able to say the same about directly distributed software, one would need to demonstrate that the tools lead to a natural and intuitive programming style in which problems can be isolated and solved one by one, in a step-wise fashion. And, it must be easy to establish that the solutions will tolerate the concurrency and configuration changes characterizing asynchronous distributed systems. That is, given a notion of consistency, it should be reasonably easy to establish that a particular system in fact achieves consistent behavior.

We will also want to pose questions about the extent to which the tools influence each other. Ideally, one would want tools that operate completely independently from one another. Otherwise, by extending the functionality of a system in one way, one would risk breaking the preexisting code. As we will see, "orthogonality" of a set of tools is arrived at using mechanisms closely related to the ones by which consistency is achieved.

Efficiency is also an important consideration. Nobody will use a set of tools unless it yields programs that perform as well (or better) than software built using other methods. Moreover, the absolute level of performance achieved must be good enough to support the kinds of applications likely to employ direct distribution.

A final issue relates to questions of scale. Our tools treat direct distribution as a problem "in the small." One also needs to construct larger systems out of components built using these tools, in a way that isolates the larger-system issues from the implementation of the directly-distributed components of which it is built. Otherwise, it may be impractical to talk about system design and interface issues without simultaneously addressing implementation details.

15.4. System support for direct interactions between processes

A variety of existing systems provide facilities that could be used when building directly distributed software. Below, we look at how close these come to addressing the major items in our list of tools.

15.4.1. Basic RPC mechanisms and nested transactions

Most operating systems provide *remote procedure calls* (Birrell and Nelson, 1984). The technological support for remote procedure calls has advanced rapidly during the past decade, and sub-millisecond RPC times for inter-site communication should be common in operating systems in the near future. RPC does not, however, address any of the problems in the above list. Thus, the programmer confronted by a direct distribution problem would be in a very difficult situation when using a system for which RPC is the primary communication mechanism. Short of building a complex application-level mechanism to resolve these problems, there would seem to be no way to build directly distributed software using an unadorned RPC facility.

To make this more concrete, let us consider a specific problem that might arise in the context of the toolkit. Among the many issues that the tools must address, a key problem is to synchronize the actions of a set of processes that are performing some action concurrently. This is an instance of the well known "mutual exclusion" problem, and there is no doubt that any system supporting direct interactions between processes will need a mutual exclusion mechanism.

A typical solution might implement a token managed with rules like the following:

> *A set of processes shares exactly one copy of a token, using operations to pass and request it. If the holder of the token fails, a pass is done automatically on its behalf, in such a way that the token is never permanently lost unless all processes fail, and duplicate tokens never arise within the operational set. New processes can join the set dynamically.*

How would one go about solving this problem using remote procedure calls? Typical RPC implementations detect failures using timeouts. Since timeouts can be inaccurate, an agreement protocol is needed to deal with token-holder failures. For example, one could try to inform all operational processes of each pass, so that they know which process to request the token from. However, in addition to the inaccuracy of the failure detection mechanism, the solution must deal with the possibility that the token could be in motion at the time of a request. Dynamic group membership changes make these problems even more complicated.

Although there are several systems that extended RPC to deal with failures and concurrency, their orientation has been on the shared memory paradigm, by extending RPC into a form of *nested transaction*. In a sense this is not surprising, since RPC is a pairwise mechanism in which the caller is the active participant and the called process is passive until it receives a request from the caller — a structure strongly evocative of the relationship between a transaction manager and a set of data managers on which it performs operations in a database setting. The ARGUS language and the CAMELOT system both take this approach, with ARGUS focusing on linguistic aspects of the problem and CAMELOT on performance and on creative use of virtual memory mechanisms. For brevity, we will not discuss either of these systems in detail here. However, it is important to recognize that the token passing problem remains difficult to solve in either system, and the same can be said for many (not all!) of the other tools in our list. The reader may want to try and design a token passing protocol using any of these approaches (pure RPC, ARGUS or CAMELOT): although feasible, it isn't easy!

Token passing is just a simple example of the sort of problems that a directly distributed system would have to solve. In a setting where token passing is difficult, the implementation of complex directly distributed systems will surely be impractical. Some experimental evidence to support this claim exists: many systems support RPC but few provide mechanisms like the token passing facility outlined above. One system that does, at least internally, is Digital Equipment's VAX-Clusters system, which uses a locking facility similar to the token mechanism (Kronenberg, Levy, and Strecker, 1986). However, the lock manager implementation is complex, and few application designers could undertake a similar effort.

15.4.2. Quorum replication methods

Many database systems manage replicated data using quorum schemes (see Chapter 13). Quorum mechanisms support replicated data without the added "baggage" of a full-blown transactional system. Do they offer an appropriate primitive on which to base a complex directly distributed program?

To answer this question, let us briefly review the mechanism that a quorum replication facility requires. The basic idea of a quorum replication scheme is that read and write operations must be performed on sufficiently many copies of the replicated data object to ensure that any pair of writes overlap on at least one replica and that any read overlaps with the most recent write.

When a quorum scheme is implemented, update operations require two phases, while read-only operations can be done in one phase. An update is transmitted during the first phase, and then committed in the second phase if a quorum of copies were updated and aborted otherwise. The abort is needed to avoid an uncertain outcome if some processes failed just after doing the update but before getting a chance to reply. Consequently, recovering processes must recover by determining the status of uncommitted operations that were underway at the time of the failure.

Now, consider the implications of this in a fault-tolerant setting. It will be impossible to avoid running both reads and writes synchronously (meaning that neither can be performed on any single data item). The problem is that if we don't want writes to block whenever a failure occurs, the write quorum will necessarily be smaller than the full set of replicas; for example, in a scheme that will provide continued availability in the presense of 2 failures, the write quorum size must be at least 2 smaller than the total number of copies. To ensure overlap with the writes, it follows that the read quorum must be larger than the number of simultaneous failures that we want to tolerate, e.g. 3 copies in the above example. Thus, although read operations can be done in one phase, they cannot be done on any single copy of the data item. In light of this, one sees that although quorum schemes are conceptually easy to describe, they have important drawbacks.

Specifically, a quorum replication mechanism can be expected to run slowly, because of the need to execute both reads and updates synchronously (that is, replies are needed from remote copies of data items before these operations can be completed). And, a fairly complex recovery mechanism must be implemented to support the multiphase commit done on writes and to handle the recovery of a process that had a copy of a data item that was being updated at the time of a failure. If every approach to these problems were equally synchronous, this objection would not be an important one. However, as we will see below, there are asynchronous alternatives of comparable complexity, and would normally outperform a fault-tolerant quorum scheme.

Could quorum methods be used to solve the general set of direct distribution tools enumerated earlier? Consider the token passing problem. It would certainly be possible to use quorum methods to update variables identifying the current token holder and request queue. However, one would be faced with the issue of maintaining a list of processes holding copies of this information. The problem here is that database systems are fairly static; a copy of a replicated database may be online or offline, but the set of copies doesn't change very often. Thus, databases usually define quorum sizes statically, taking both failed and operational replicas into account. In contrast, the problem given above requires that the set of processes involved change dynamically, with new processes joining in an unpredictable manner and old members dropping out permanently. We know of no quorum based scheme that explicitly supports this sort of dynamicism, although some of the extended quorum algorithms developed by Herlihy (1986b) may be capable of solving this problem. Thus, a fault-tolerant quorum-based token passing algorithm is likely to be costly both in terms of code required and the performance that it can achieve, and may prove restrictive with regard to the degree of dynamic behavior it can accommodate.

Just as transactions have begun to appear in higher-level systems and languages, so have quorum replication techniques. In particular, the AVALON language, which was built on top of CAMELOT, provides mechanisms for maintaining quorum-based replicated objects in which quorum sizes change dynamically (Herlihy, 1986b). The approach works well in settings with frequent communication partitioning, site failures and recoveries — in contrast to some of the methods that will be discussed below, which perform better than quorum schemes when problems of these sorts do not arise, but may block during communication partitioning and impose a high overhead when site failures and recoveries occur.

15.4.3. The V system

Until now, the systems discussed in this chapter have been those that do not really support direct interactions between processes. V is an RPC-based system that simultaneously places a strong emphasis on performance and on providing system support for forming process groups and broadcasting requests (Cheriton and Zwaenepoel, 1985). By virtue of supporting process groups, V is able to address many of the problems in our toolkit. However, V was not designed with fault-tolerance or distributed consistency as a primary consideration, and provides little support for the application designer for whom these are major issues. For example, recall the problem of how a group broadcast mechanism should work when group membership is changing or processes fail. Although V makes a "best effort" to deliver messages to all members of a process group, V gives no absolute guarantees that all receive a given broadcast, or that messages are received in some consistent order relative to a membership change.

A V-style broadcast is well suited for some types of directly distributed applications. If an application is broadcasting to a network resource manager, for example to find the mailbox for a user, it may not matter very much if some processes fail to receive the request. It is easy to program around the uncertainty, ensuring that behavior is correct in all but the most improbable scenarios.

Thus, when broadcasting mailbox location updates, it may not matter if some processors miss occasional updates (Lampson, 1986). In this example, and in others with a similar character, the V broadcast primitive is suitable for implementing replication.

Our token passing problem is no easier to solve in V than in a standard RPC setting. Similarly, replicated data with a 1-copy behavior constraint would be hard to implement on top of the standard V broadcast: if an update fails to get through, or two updates arrive in different orders at different group members, the copies could end up with inconsistent values. To solve either problem, a non-trivial mechanism would be needed at the application level, and as noted earlier few application designers would be capable of undertaking such a complex and uncertain development effort.

15.4.4. The Linda Kernel

One problem with using shared memory in a directly distributed application is that the processes interacting through the memory are also faced with a complex synchronization problem. At Yale, Carriero and Gelernter have developed a set of language extensions that solve both of these problems simultaneously (Carriero and Gelertner, 1986). Linda enables them to use shared memory as a tool for building directly distributed software. Before discussing the system in any detail, it should be noted that the present implementations of Linda are designed for a different environment from the one that interests us here: parallel processors, where failure is less of an issue and concurrent execution is paramount. Of course this does not preclude a Linda implementation for loosely coupled distributed systems subject to failure, but substantial re-engineering would be needed. What makes Linda interesting is that it permits simple solutions to some of the toolkit problems that the mechanisms reviewed above are unable to handle, and this makes the system interesting in the light of our objectives in this chapter.

The Linda approach is based on the idea of a shared collection of *tuples*. The operations provided are *out* (add a tuple to the space), *in* (read and remove a tuple) and *read* (read a tuple without deleting it). Tuples can be extracted by specifying the values of some fields and just the data types for others; in this case Linda performs a pattern matching operation that finds some tuple matching the specified fields and returns the values contained in the remaining fields. The caller can specify whether or not an *in* operation should block if it cannot be immediately satisfied. The basic idea of Linda is to allow a set of processes to concurrently execute tuple-space operations, but to perform those operations in a logically (but not necessarily physically) serialized fashion.

There have been several implementations for Linda. Among these is one in which *in* and *out* are broadcast using an ordered protocol and executed in parallel by all processes, one in which *out* is performed locally, and *in* broadcast to all processes, and one that operates by dividing tuple space among the various processes in such a way that one can map a tuple to its handler using a simple hash function. That process then resolves the *in* or *out* requests presented to it in the order they arrive. Thus, although the tuple space is conceptually shared, it is not necessarily physically replicated, and different data objects may be managed by different processes.

One could easily build a solution to the token passing problem in Linda. The token would be represented by a tuple, and the processes would use the blocking version of *in* to request it and *out* to pass it. The solution would be subject to some limitations, but here one needs to distinguish intrinsic issues from consequences of the engineering decisions made as part of the Linda implementation. For example, the dynamic group membership aspect of the problem could be solved easily in a version of Linda that performs *out* locally and broadcasts *in* operations, but would be much more difficult in a version of Linda that requires that the set of processes managing the tuple space be static. Another problem that arises is that none of the present implementations of Linda can tolerate failures: if the process that manages some fragment of tuple space process crashes, its part of tuple space is simply lost. Although one could solve this by replicating the tuple space, to do so would just push the issues that were identified above into the Linda implementation, since Linda itself would now need to implement a correct and fault-tolerant replication mechanism. Thus, the Linda primitives somehow embody a property that makes it easy to solve problems like the token passing one. However, solutions to problems like the replicated data problem are still needed before we can apply this successful aspect of the Linda system in the general setting of our toolkit.

Linda has been applied successfully in several settings. In the area of parallel simulation (that is, simulations run on distributed or parallel systems), Linda has been used primarily to build parallel solutions to a range of problems. Nearly full utilization of the processors is often cited.

For example, and FIGURE 15.1 illustrate a skeletal solution to the drilling problem introduce at the start of this chapter, using Linda tuples to describe the work to be done and the outcome. Each hole to be drilled is described by a "pending work" tuple. A drill control processor selects a tuple on which to work, drills the hole, and then outputs a tuple describing the outcome.

Because Linda was not designed to address fault-tolerance, the above code lacks the mechanism needed to detect failures and generate a list of holes that a technician should recheck. That is, there is no good way to generate a `MUST_CHECK` tuple on behalf of a crashed control process in present versions of Linda. Likewise, it is hard to see how one could handle dynamic scheduling using Linda's tuple-matching mechanism. The problem here is that the language lacks a way for the user to provide a tuple selection criteria. Thus, to pick the optimal hole subject to some user-specified metric, such as the hole that minimizes the total expected drilling time, it would seem necessary to examine the full set of tuples. (Of course, there may well exist a clever encoding of the problem into a tuple-space data structure that would efficiently solve this.) One could also certainly imagine extensions of the language in which this problem could be addressed.

```
NewBatch(hole_list)
  {
    /* Load workspace with hole descriptions */
    for(each hole in hole_list)
        out("to_do", hole.x, hole.y, ...);

    /* Wait for all to be processed */
    for(each hole in hole_list)
    {
        /* in will block until outcome is known */
        in("done", hole.x, hole.y, int status);
        if(status == MUST_CHECK)
            print("Must check hole at ...\n", hole.x, hole.y);
    }
  }
```

FIGURE 15.1. Generating work for a cell controller in the C-Linda

```
/* A typical drill controller */
DrillControl()
  {
    forever
    {
        in("to_do", int x, int y, ...);

        /* Position the drill, then make the hole */
        PositionDrill(x, y);
        outcome = DrillHole(HoleSpecs);

        /* Record outcome */
        out("done", x, y, outcome);
    }
  }
```

FIGURE 15.2. A control process for a single drill

We find Linda intriguing because it points to a possible structure for the kind of problems we are interested in. The essential observation is that when all processes in a system cooperate through a mechanism that orders elementary operations that might interfere with one another, distributed consistency is surprising easy to achieve. As we will see below, by substituting ordered reliable broadcasts for ordered tuple-space operations, one can implement fault-tolerant solutions to most of the issues that the toolkit raises. Just as the partial solution to the drilling problem shown above arises naturally out of the structure that Linda suggests, fault-tolerant solutions to the other problems in the toolkit result from this extended approach.

15.4.5. The HAS project

At IBM, the HAS project explored a closely related approach. HAS supports Δ-*common storage*, which is much like the Linda tuple space but defined in terms of abstract operations on a shared memory. As in the case of the realtime broadcast protocol discussed in Chapter 14, updates are completed within a period of time bracketed by upper and lower bounds expressed in terms of a computed parameter, Δ. That is, no update can be completed in *less* than a certain minimum time, but neither will any be delayed for longer than a specified maximum time. In addition to the shared memory abstraction, HAS provides a processor grouping mechanism on top of this layer of protocols, like the one we suggested for individual processes but at a coarser granularity (Cristian *et al.*, 1986; Cristian, 1988).

In contrast to Linda, HAS was designed for use in loosely coupled processors communicating on high speed point-to-point channels as well as broadcast media such as token rings, and subject to a variety of failure modes. Moreover, the HAS methodology provides tolerance to a range of failures including Byzantine failure modes in which processes can behave in arbitrary malicious ways and experience bizarre clock failures. However, any protocol that would actually tolerate worst case behavior for a wide range of possible failure modes would perform very poorly, translating to a very long minimum delay when updating the shared memory. To arrive at a practical facility, the group exploited the observed failure characteristics of this environment, which permitted them to make a tradeoff between the types of failures that their implementation actually tolerates and its performance. For a realistic scenario, with relatively high speed processors on a fast token ring, this approach resulted in a realtime atomic delivery protocols in which the minimum delay to update the shared memory was fairly small — on the order of 100ms.

With current technology, the HAS approach would not scale well to very large networks because the realtime performance characteristics of such a system would be very poor in comparison to the small, closely coupled machines used to achieve the sort of performance cited above. An open question relates to how changes in communication hardware and increased processor speeds could impact on the way the system scales.

15.4.6. The ISIS system

Like Linda and HAS, the ISIS system adopts an approach based on synchronous execution, whereby every process sees the same events in the same order (Birman and Joseph, 1987a). However, ISIS simultaneously seeks to provide fault-tolerance, effective replication mechanisms, and good performance in larger local area networks. The system starts with the observation that synchronous execution models offer strong advantages. Their primary disadvantage is one of cost: without hardware support, a distributed lock-step execution performs poorly. Even with hardware support, a lock-step style of computation does not time.

To address this, ISIS provides an *illusion* of synchronous execution, in much the same sense that transactional serializability provides the illusion of a sequential transaction execution. Whenever possible, ISIS [illegible] synchronization in order to reduce the degree to which processes can delay one another and to better exploit the parallelism of a distributed environment. We use the term *virtual synchrony* to refer to this approach, because the system appears to be synchronous but actually is fairly asynchronous. For example, processes are permitted to initiate an operation asynchronously, by broadcasting a request without pausing to wait for a reply.[3] When this is done, ISIS behaves as if the messages were delivered immediately, preventing any action that occurs "after" the broadcast was sent from seeing a system state from "before" the broadcast was sent. Similarly, ISIS delivers broadcasts with common destinations in the same order everywhere — except when it is possible to infer that the application does not need such strong ordering. In such cases, ISIS can be told to relax the delivery ordering rules, which permits it to use a cheaper broadcast protocol.

ISIS differs from Linda and HAS in providing a message-oriented rather than a shared memory interface. The basic ISIS facilities include tools for creating and managing process groups, group broadcast, failure detection and recovery, distributed execution and synchronization, etc. A Linda-style replicated tuple space is easy to implement in ISIS, as is 1-copy replicated data. Moreover, the implementations can readily be customized to address special requirements of the application program, such as the selection of the optimal next hole for a controller to drill. We will be looking at these mechanisms in more detail below, and then at an example illustrating how ISIS could be used to solve the drill control problem.

15.5. An execution model for virtual synchrony

One desirable feature of systems like ARGUS, CAMELOT, Linda, HAS and ISIS is that one can write down a model describing the execution environment they provide. In the case of ARGUS or CAMELOT the model is based on nested transactions, and the lowest-level elements are data items and operations upon them. Models for the latter systems are similar but oriented towards the

[3] Note the difference between this and an RPC, where such a pause is built in even if no reply is desired. For example, on the SUN 3 version of ISIS a program that issues an asynchronous broadcast to 5 destinations would resume executing after a delay lasting for a small fraction of a millisecond. The remote message deliveries occur within 5-10 milliseconds. With RPC, which has a 10 millisecond round-trip time under UNIX on a SUN 3, the sender would be delayed by 50 milliseconds, plus any costs associated with the group addressing protocol. Delivery would take as long as 45 milliseconds between the start of the broadcast and the arrival of the last message. On systems with faster processors and cheaper RPC costs, the costs here might scale, but the same argument could still be made. The advantage is that when ISIS sends acknowledgement messages, it overlaps them with concurrent execution in the sender, winning improved performance.

representation of synchronous executions. Before looking at virtually synchronous algorithms for the tools enumerated earlier, it will be helpful to start by defining such a model and giving virtual synchrony a more precise meaning.

The elements of the execution model we will be working with are processes, process groups, and broadcast events. Broadcast events include more than group communication. Point-to-point messages are treated as a broadcast to a singleton process-group. Failures are treated as a kind of broadcast too: a last message from the dying process informing any interested parties of its demise. Data items are not explicitly represented, although one can superimpose a higher level on top of this basic model in which operations and the values of data become explicit.

15.5.1. Modeling a synchronous execution

One way to understand a model is as a formalism for writing down what an "external observer" might see when watching the system execute from someplace outside of it. The external observer provides a notion of global time to relate the actions taken by distinct processes. One defines an execution to be *synchronous* if the external observer can confirm that whenever two processes observe the same event, they do so at the same instant in time. This is illustrated in FIGURE 15.3, where time advances from top to bottom.

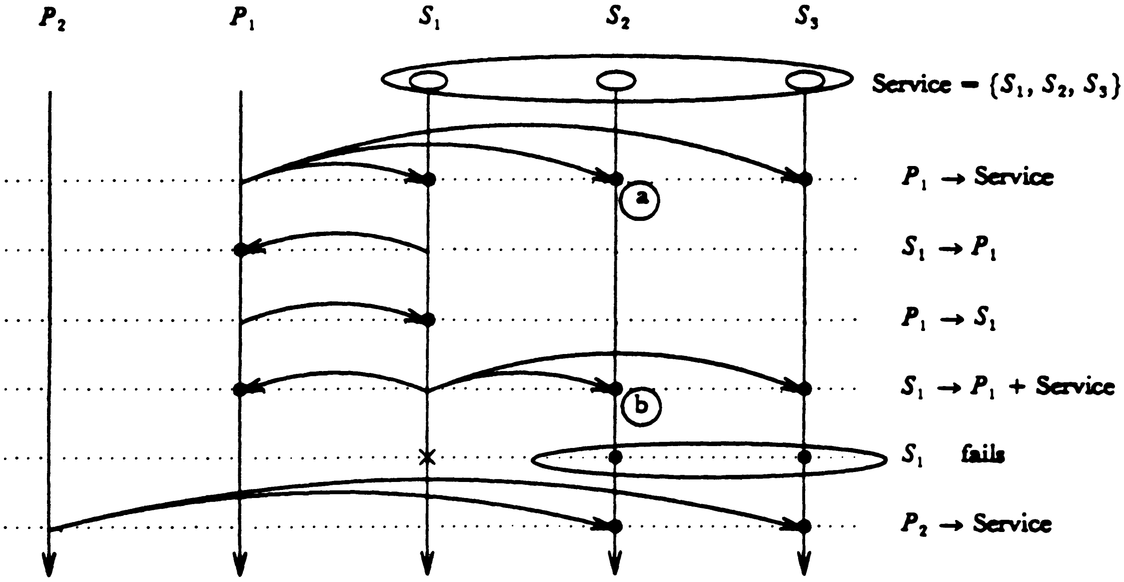

FIGURE 15.3. A synchronous execution. A pair of client processes, identified as P_1 and P_2 interact with a process group containing three server processes. Execution advances from top to bottom in a lock-step manner. Several message exchanges and a failure are shown.

In a synchronous model it is easy to specify the meaning of an *atomic* ("all or nothing") broadcast to a process group. At the time at which a broadcast is delivered, it must be delivered to all *current* members of the group. Thus the set of destinations is determined by the event sequence (processes joining or leaving the group) that occurred prior to that time. This does not tell us how to implement such a broadcast, but it does give a rule for deciding whether a broadcast is a atomic or not. We will be making use of this rule below.

Among the above systems, Linda comes closest to providing a synchronous execution. However, a genuinely synchronous execution would be impractical to implement in a local area networking environment. To do so all the processes would need access to a common clock and to execute at fixed speeds, neither of which is normally possible.

15.5.2. Modeling a loosely synchronous execution

An execution is said to be *loosely synchronous* if all processes observe events in the *same order*. FIGURE 15.4 illustrates such an execution. An external observer who notes the time at which events are executed may see the same event processed at different times by different processes. However, the events will still be executed in the same order as they would have been in a truly synchronous execution. Hence if the system is not a realtime one (and this is something we assumed at the outset), processes that behaved correctly in a truly synchronous setting should still behave correctly in a loosely synchronous one (Neiger and Toueg, 1987).

More formally, for every loosely synchronous execution E, there exists an *equivalent* truly synchronous execution E'. The two executions are equivalent in the following sense. Let E_p be the sequence of events observed by process p in an execution E. Then $E'_p = E_p$ for all p, i.e. every process observes the same sequence of events in E and E'. Unless a process has access to a realtime clock, it cannot distinguish between E and E'. FIGURE 15.4 is indistinguishable from FIGURE 15.3 under this definition.

Any synchronous system is also loosely synchronous. Thus, Linda and HAS are both loosely synchronous systems; the global event ordering being imposed by hardware in the former case, and by a software protocol in the latter.

15.5.3. Modeling a virtually synchronous execution

A *virtually synchronous* execution is related to a loosely synchronous one in much the same way that a *serializable* execution is related to a serial one. The characteristic of a virtually synchronous system is that although an external observer may see cases in which events occur in different orders at different processes, the processes themselves are unable to detect this. For example, FIGURE 15.5 is a copy of FIGURE 15.4 with the delivery of event a delayed to occur after b at one destination. We would call this execution virtually synchronous if, after both a and b have terminated, no process in the system can contradict a claim that a executed first everywhere. Evidence of the order in which operations took place could be explicit in the value of some variable, or it could be reflected in the

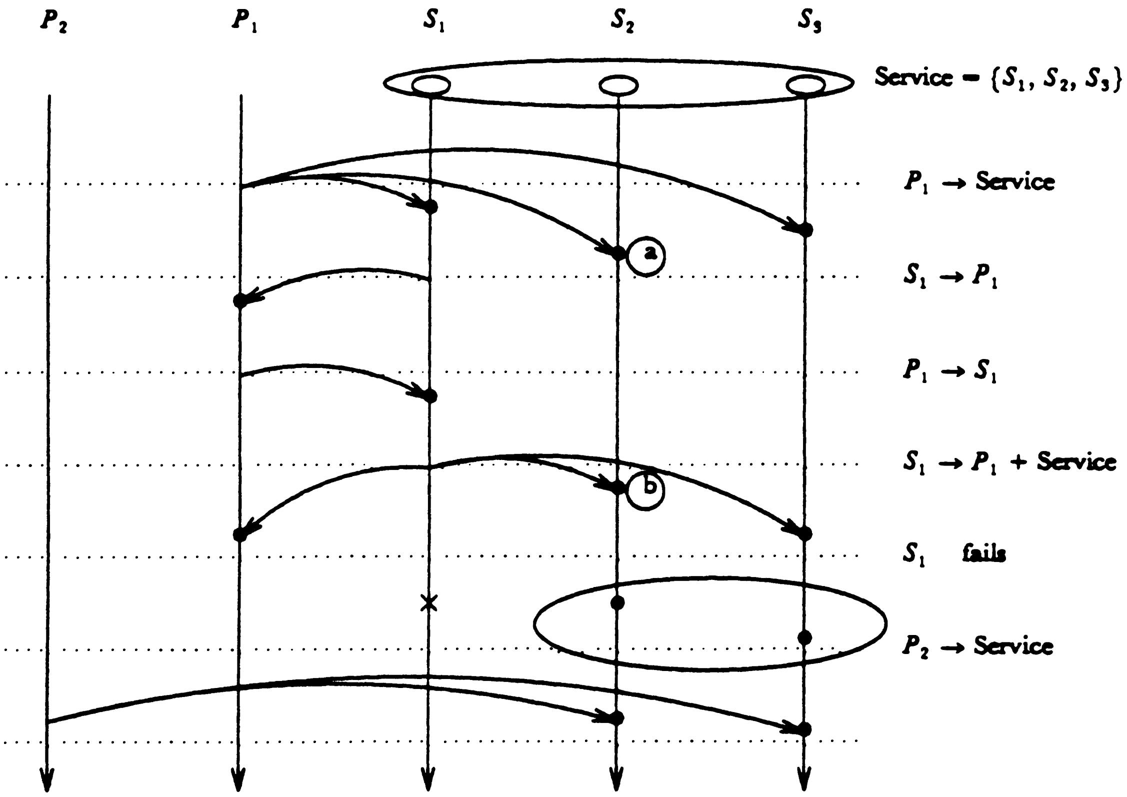

FIGURE 15.4. A loosely [illegible] execution

response to one of the requests or the actions that a process took after receiving some request.

One probl[illegible] with the sort of relaxation of order seen here is that it looks very narrow in applicability. To be of in[illegible], one needs to be able to identify relaxations that can be applied in a systematic manner and ac[illegible]y correspond to protocols of different cost. In particular, it is highly advantageous to substitute a one-phase broadcast protocol for a two-phase protocol, and this is the sort of relaxation of ordering that we are after here. The idea in building a virtually synchronous sys[illegible] is to look for such cases and to exploit th[illegible]. This was also our hidden motive in introducing a list of tools. Whereas it is unlikely that one could systematically find ways to relax order in arbitrary applications, it is entirely reasonable to do so in applications that [illegible] uniformly structured and interact primarily through standard "toolkit" interfaces. In such a setting, one could optimize the tools as a way of optimizing all the application software that later [illegible] built on th[illegible].

Recalling the definition of loose synchrony in [illegible] previous section, an execution E is *virtually synchronous* if there exists some truly synchronous execution E' equivalent to E. However, we broaden our notion of equivalence between executions by requiring only that $E'_p \approx E_p$, for all p. Here "$\approx$" means that two event sequences are *indistinguishable*, but not nec[illegible]ly identical. The

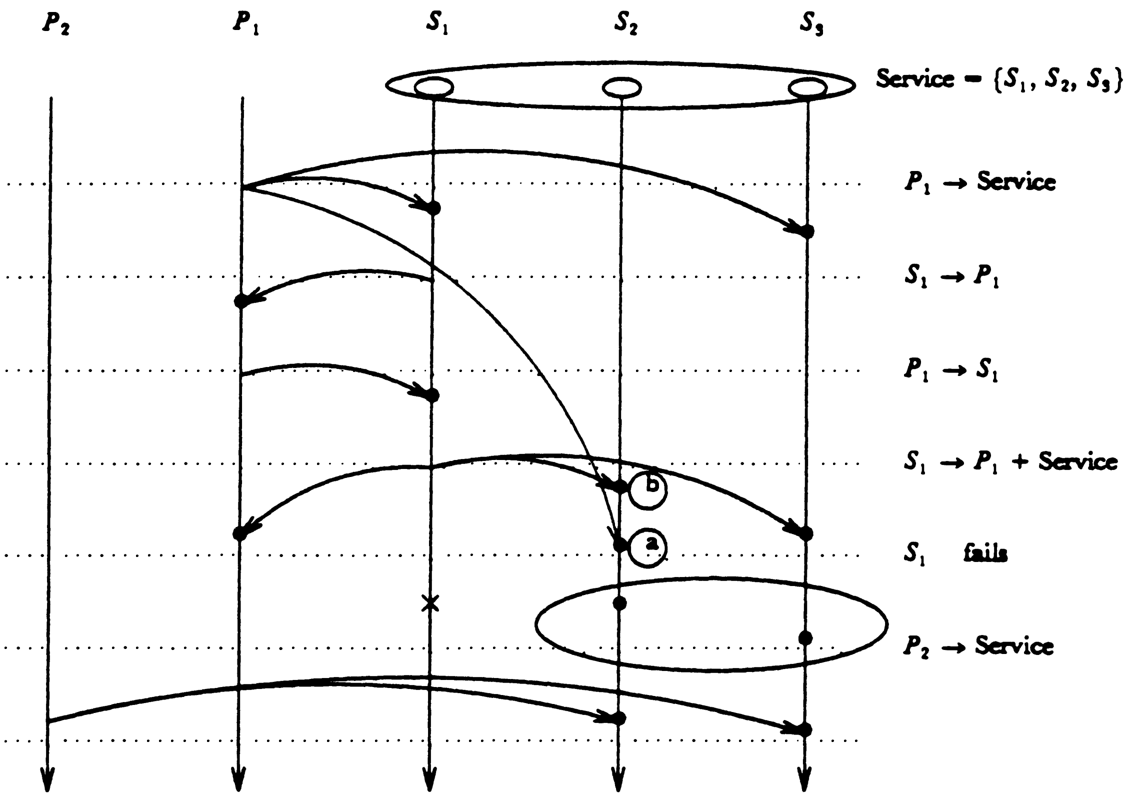

FIGURE 15.5. A virtually synchronous execution

determination of which event sequences are distinguishable depends on the semantics of the individual events in a particular application. A formal definition of this sort of equivalence and a theory of virtual synchrony have been developed by Schmuck (1988).

15.6. Comparing virtual synchrony with other models

15.6.1. Transactional serializability

Is our model really any different than a transactional one? We argue that virtual synchrony is a substantial generalization of transactional serializability.

Clearly, if a system is serializable, it is virtually synchronous. On the other hand, a virtually synchronous execution need not be serializable. First, there is nothing like a *transaction* in a virtually synchronous system. Consider a pair of processes, executing concurrently, that interchange a series of messages leading to a dependency of each on the state of the other. In a transactional setting, this could only occur if each interaction was a separate top-level transaction — a series of atomic actions with no subsuming transactions at all. However, transactional work has generally not considered this case directly, and it is normally not even stated that the serialization order for such top-level actions should be the order in which they were initiated. For many concurrency control schemes, such as two-phase locking, there is no *a-priori* reason that this would be the case: a single transaction might asynchronously initiate two top-level transactions, first T_1 and then T_2, which would be serialized in the order T_2 followed by T_1. [4]

Virtual synchrony imposes an explicit correctness constraint on sequences of interactions like this, namely that unless the order is irrelevant, the events be observed in the order they were initiated, even if they were initiated asynchronously, and even if order arises through a very indirect dependency of one action on another. Moreover, virtual synchrony talks about process groups and distributed events (broadcasts, failures, group membership changes, and so forth). None of these issues arise in a conventional transactional setting. In light of their importance within the directly distributed tools we listed earlier, and the apparent difficulty of layering them on top of transactions, these are significant differences.

Transactions and virtual synchrony both depend strongly on the semantics of operations. In the case of transactions, this was first observed when an attempt was made to extend transactional serializability to cover abstract data types (Schwarz and Spector, 1984). Whereas it is easy to talk about concurrency control and serializability for transactions that read and write (possibly replicated) data items, it is much harder to obtain good solutions to these problems for transactions on abstract data types. In the case of virtual synchrony, the problem arises because the model lacks data items or any other fixed referent with well-known semantics. One can only decide if an execution is virtually synchronous if one knows a great deal about how the system executes. This is an advantage in that the definition is considerably more powerful than any data-oriented one. There are many virtually synchronous systems that could not be interpreted as synchronous by somehow making the model knowledgeable about data. On the other hand, the presence of semantic knowledge makes it hard to talk about correct or efficient system behavior in general terms, without knowing what the system is doing. As we will see shortly, one can only do this through a detailed analysis of the algorithms on which a particular system relies.

15.6.2. Virtual synchrony in quorum-based schemes

Earlier, we saw some examples of how a quorum scheme might be used to obtain consistent behavior in a relatively unstructured setting. Such an approach can be understood as a form of virtual synchrony. The basic characteristic of a quorum scheme is its ***quorum intersection relation***, which specifies how large the quorums for each type of operation must be (Herlihy, 1986b). If two operations potentially conflict — that is, if the outcome of one could be influenced by the outcome of the other — then their quorums will intersect at

[4] For example, T_1 might block waiting for a lock and then update variable x, while T_2 acquires its locks and manages to update x before T_1.

one or more processes. One can thus build a partial order on operations, such that all conflicting operations are totally ordered relative to one another, while non-conflicting operations are unordered. Since non-conflicting operations always commute, the executions of a quorum-based system are indistinguishable from any extension of this order into a total one. Such a total order can be understood as a description of a synchronous execution that would have left the system in the same state as it was in after the quorum execution. Thus, a quorum execution is virtually synchronous.

15.7. System support for virtual synchrony

15.7.1. The ISIS virtually synchronous toolkit

Let us now return to the ISIS system and look more carefully at some of the virtually synchronous algorithms on which it is based.

15.7.2. Groups and group communication

The lowest level of ISIS provides process groups and three broadcast primitives for group communication, called CBCAST, ABCAST and GBCAST. The primitives were discussed previously, in Chapter 14, and their integration into a common framework supporting group addressing is covered elsewhere (Birman and Joseph, 1987b). We therefore focus on their joint behavior while omitting implementation details.

In ISIS, a *process group* is an association between a group address and some set of members. Membership in a process group has low overhead, hence it is assumed that processes join and leave groups casually and that one process may be a member of several groups.

A *view of a process group* is a list of its members, ordered by the amount of time they have belonged to the group. ISIS includes tools for determining the current view of a process group and for being notified of each view change that occurs. All members see the same sequence of views and changes.

The destination of a broadcast in ISIS is specified as a list of groups. Group membership changes are synchronized with communication, so that a given broadcast will be delivered to the members of a group in the same membership view.

Recall that a broadcast is *atomic* if it is delivered to *all* members of each destination group. Here, "all" refers to all the group members listed in the process group view in which delivery takes place, which might not be the same as the membership when the broadcast was initiated.[5] A *virtually atomic* delivery is one in which all group members *that stay operational* receive the message in the same view. The ISIS broadcast primitives are all virtually atomic. Thus, the

[5] In ISIS, it will be the same or a subset of the initial membership.

recipient of an ISIS broadcast can look at the "current" group membership (in a virtually synchronous sense) and act on the assumption that all of the listed processes also received the message. It may subsequently see some of them fail, perhaps without having acted on the message.

CBCAST, ABCAST and GBCAST differ in their delivery ordering properties. Before we review these differences, recall the definition of the potential causality relation on events, $\rightarrow$ introduced in Chapter 14: $e \rightarrow e'$ means that there may have been a flow of information from event e to event e' along a chain of local actions linked by message passing.

Let $BCAST(a)$ denote the initiation of broadcast a and $DELIVER(a)$ the delivery of some a to some destination. All three flavors of broadcast ensure that if $BCAST(a) \rightarrow BCAST(b)$ for broadcasts a and b (below, $a \rightarrow b$), then $DELIVER(a)$ will precede $DELIVER(b)$ at any common destinations. In fact, they satisfy an even stronger property, namely that if $a \rightarrow b$ then even if a and b have no common destinations, b will be delivered only if a can be delivered too. This ensures that if some subsequent broadcast c is done, with $b \rightarrow c$, and a and c have common destinations, the system will be able to respect its delivery order constraints. The ISIS delivery ordering constraint can be thought of as a FIFO rule based not on the order in which individual processes transmitted broadcasts, but on the order in which threads of control did so. Here, a thread of control is any path along which execution may have proceeded.

CBCAST satisfies exactly the above delivery constraint. If a and b are concurrent, then CBCAST might deliver a and b in different orders. ABCAST provides a delivery order that extends $\rightarrow$ so that if a and b are two concurrent ABCASTs, a delivery order will be picked and respected at all shared destinations. However, ABCAST and CBCAST are unordered with respect to each other. GBCAST, in contrast, provides totally ordered delivery with respect to *all* sorts of broadcasts. Thus, if g is a GBCAST and a is any sort of broadcast then g and a will be delivered in a fixed relative order to all shared destinations.

A system that uses only ABCAST to transmit broadcasts is loosely synchronous. For this reason, ISIS uses ABCAST as its default protocol when not told otherwise by the programmer. However, ABCAST is costly. Like the quorum protocols, it sometimes delays message deliveries in a way that would be noticeable to the sender. CBCAST is much cheaper, especially when invoked asynchronously.[6] This leads to the question of what when synchronization can be relaxed by changing an ABCAST to a CBCAST in a broadcast-based algorithm.

[6] The implementation of ISIS is more complex than the earlier discussion of these protocols made it appear. For reasons of brevity, the [illegible] issues are not discussed here. However, the reader should be aware that to make effective use of protocols such as these, a substantial engineering investment is needed. This ranges from the requirement for a system architecture that imposes low overhead to [illegible] for scaling the protocols to run in large networks and to avoid [illegible] when communication patterns overload the most costly aspects of the protocols (Birman, Joseph, and Schmuck, 1989).

15.7.3. When can synchronization be relaxed?

Let us examine the degree to which some specific algorithms depend on the ordering characteristics of the broadcasts used for message transmission. We begin with some examples drawn from a single process group with fixed membership:

- A replicated tuple space, supporting the Linda *in*, *out* and *read* operations but using replicated data to achieve fault-tolerance.
- A shared token, supporting operations to *request* it, to *pass* it, and to determine the current holder.
- Replicated data. There are two cases: a variable that can be updated and accessed at will, and a variable that can only be accessed by the holder of a token (lock) on it. We look only at the second case, as the first one is essentially the same as for the Linda tuple space.

15.7.4. Shared tuple space

Say that we wish to replicate an Linda tuple space (refinements such as fragmenting the space using a hashing rule could be superimposed on a solution to this basic problem). In ISIS, this would be done by having the processes that will maintain replicas form a process group. What kind of ordering would be needed here? Except for the *read* operation, which can be done locally by any process managing a replica, all operations change the tuple space. Consequently, they all potentially conflict with one another, suggesting that ABCAST is the appropriate protocol.

Notice that it is possible to relax the relative ordering when two operations that affect independent tuples. Could we take advantage of this to replace ABCAST with a cheaper protocol? Such a protocol would look at the type and arguments of each operation. It would chose a global order for operations that actually conflict with one another, while permitting non-conflicting operations to execute in arbitrary orders. At first glance, it may sound like one could design a hybrid protocol that would run in two phases in the case of a conflict but deliver in one phase if no conflict were found. However, on closer examination one sees that if the *possibility* of conflict exists, a two phase protocol is always needed. The problem is that when an operation o arrives at a replica r, it is not sufficient to know that no conflicting operations are underway at r; one needs to know that none are underway at *any* replica. This precludes delivery during the first phase. But, if two phases will be needed, there is no benefit to be gained by including a test for conflict. It seems more reasonable to just use a normal ABCAST.

15.7.5. Shared token

The shared token is interesting because it admits a variety of possible implementations. The most synchronous implementation is the easiest to understand. In this algorithm, both *request* and *pass* operations are transmitted using a globally ordered group broadcast. Members maintain a queue of pending requests. A token holder wishing to do a pass operation first waits until at least one request is pending, then broadcasts the pass operation. On receiving such a broadcast, all processes mark the request at the head of the queue as having been granted.

What if we wanted to use a cheaper broadcast primitive? Since the algorithm depends on a totally ordered request queue, we cannot use a cheaper protocol for sending requests without major algorithmic changes. On the other hand, it might be possible to use a less ordered protocol for transmitting *pass* operations. This, however, raises a subtle issue. It may be possible for a *request* message to reach one group member much earlier (in real time) than some other. If we change the broadcast primitive, such a sequence could result in a race, where a subsequent *pass* operation arrives at a slow process before the request from which it will be satisfied (FIGURE 15.6).

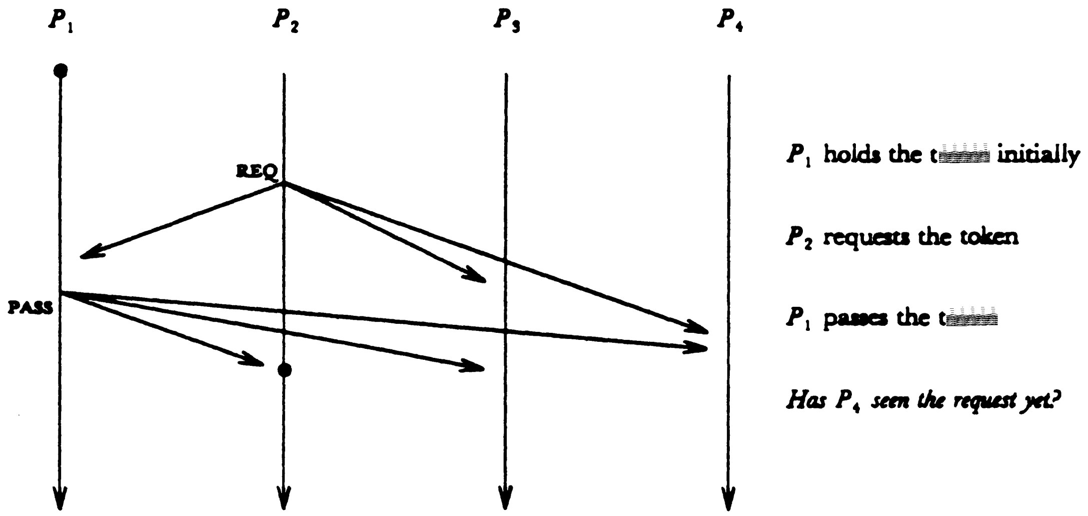

FIGURE 15.6. A race could develop when using a weakly ordered broadcast

Likewise, a process about to pass the token could decide to satisfy a request that has already been granted, i.e. if it were to receive the token before receiving the pass message corresponding to that earlier request. Clearly, this would lead to error.

Fortunately, although the situations described above could arise when using a totally unordered protocol, or one that is FIFO on a point-to-point basis, it cannot occur with a CBCAST protocol. To see this, notice that a process cannot try to pass the token unless it has first requested it and then received it from some previous holder. Let R_i denote the i'th token request to be satisfied and P_i the pass done by the process that issued R_i. We have $R_i \rightarrow P_i$ and $\forall j < i: P_j \rightarrow P_i$. Thus, $\forall j < i: R_j \rightarrow P_i$. In other words, when CBCAST delivers a particular *pass* message, the destination will always have received the prior *request* operations and *vice versa*, eliminating the source of our concern.

This reasoning inspires a further refinement. Why not broadcast *request* operations using CBCAST as well? The preceding analysis shows that any process receiving a *pass* will have received the *request* to which that pass corresponds. Thus, the only problem this change would introduce is due to the loss of a global request ordering: different processes could now receive requests in different orders. This means that it will no longer be possible for each process to determine, in parallel with the others, the new holder of the token: they [illegible] have no basis for making consistent decisions. On the other hand, the decision could be made by the process about to send a *pass* message. If there is no pending request, that process will have to defer its *pass* until a request turns up. Given a *pass* message that indicates the identity of the new holder, all processes can find and remove the corresponding request from their set of pending requests, where it will necessarily be found.

FIGURE 15.7 illustrates this behavior schematically. The darker lines show the path along with the token is passed, which is precisely the part of the $\rightarrow$ relation used in the above argument.

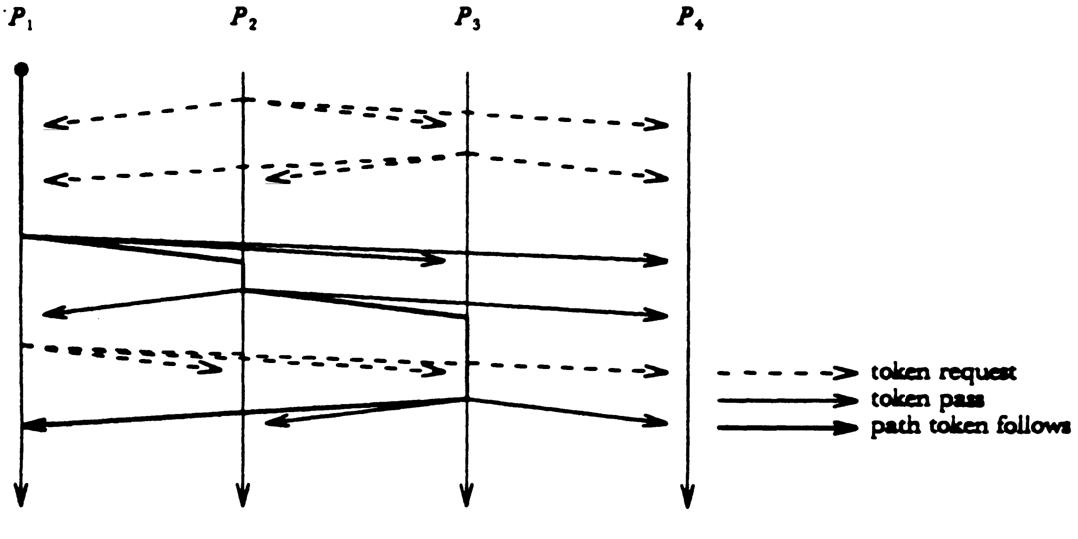

FIGURE 15.7. A virtually synchronous token-passing algorithm

To summarize, a token-passing problem admits a variety of correct solutions. The cheapest of these, from the point of view of message transmission, is the third. It depends only on the ordered delivery of messages that [illegible]e to one another under $\rightarrow$. A slight price is paid when a *pass* operation is done and there is no pending *request*: the broadcast corresponding to the *pass* must be delayed, and this will have the effect of introducing a delay before the *request* can be satisfied when it is finally issued. In the ISIS system, the benefits of using an asynchronous one-phase protocol to implement the broadcast far outweigh any delay incurred in this manner.

Token passing is an especially interesting problem because it captures the essential behavior of any system with a single locus of control that moves about the system, but remains unique. Many algorithms and applications have such a structure. Thus, if we can solve token passing efficiently, there is some hope for solving a much larger class of problems efficiently as well. Notice in particular how our final optimization replaced a global ordering decision (in the ABCAST) with a local one in the sender, and then took advantage of the fact that the sender holds mutual exclusion to propagate the decision using an inexpensive protocol. Viewed in this manner, one sees that the original algorithm was discarding ordering information and then paying a price to regenerate it! The underlying lesson is clear: in constructing efficient order-based algorithms, one must make every effort to preserve and exploit any sources of distributed order available to the application.

15.7.6. Replicated data with mutual exclusion

The usual reason for implementing tokens is to obtain mutual exclusion on a shared resource or a replicated data item. In an unconstrained setting, like the Linda tuple space, we saw that correct behavior may require the use of a synchronous broadcast. What if updates are only done by a process that holds mutual exclusion on the object being updated, in the form of a token for it?

If W_i^k denotes the k'th replicated write done by the i'th process to hold the token, we will always have $R_i \rightarrow W_i^0 \rightarrow \cdots W_i^n \rightarrow P_i$. Now, since P_i denotes the passing of the token to the process that will next obtain it, it follows that if a process holds a token, then all *write* operations done by prior holders precede the *pass* operation by which the token was obtained. Thus, if CBCAST is used to transmit *write* operations, any process holding the token will also see the most current values of all data guarded by the token.

It follows that we can obtain 1-copy behavior for a replicated variable using a token-passing and updating scheme implemented entirely with asynchronous one-phase broadcasts. Any process holding the token will "know" it also possesses an up-to-date state. This kind of knowledge is formalized in Taylor and Panangaden (1988). Moreover, execution can be done by reading and writing the local copies of replicated variables without delay — just as for a non-replicated variable — and leaving the corresponding broadcasts to complete in the background.

FIGURE 15.8 illustrates replicated update using token passing in this manner. All the updates occur along the dark lines that highlight the path along which the token travels, which is exactly the $\rightarrow$ relation used in the above argument. Although the system has the freedom to delay updates or deliver them in batches, it can never deliver them out of order or pass a token to a process that has not yet received some pending updates. The algorithm is thus executed as if updates occurred instantaneously.

What about an application that uses multiple data items, and multiple locks? The algorithm described above can yield very complex executions in such a setting, because of delayed delivery of update messages and deliveries that can occur in different orders at different sites. Nonetheless, such a system always has

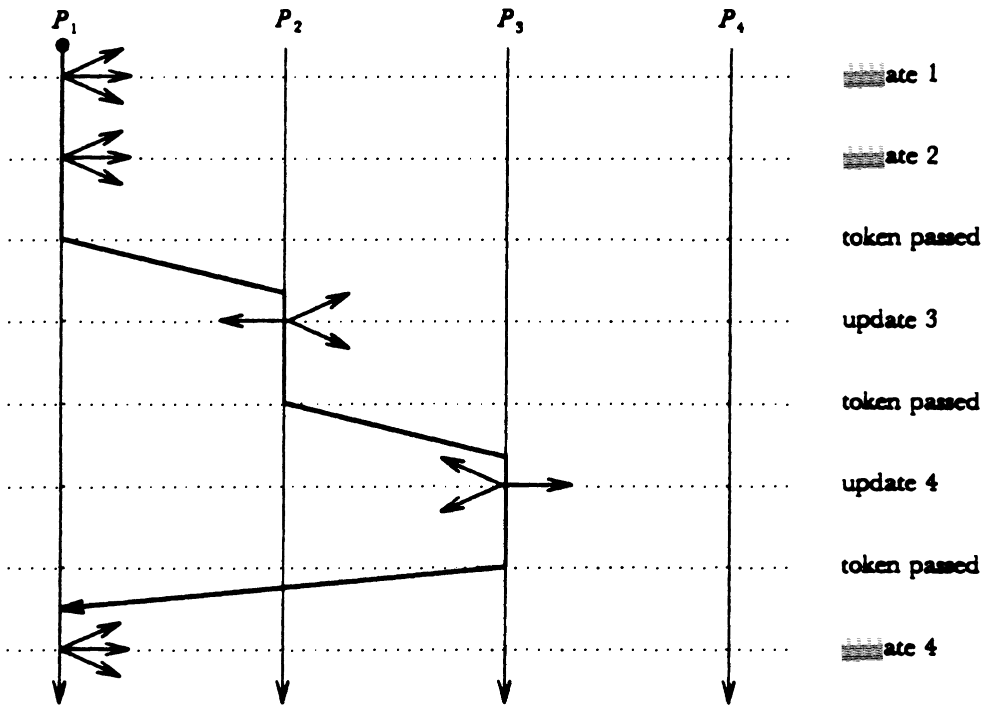

FIGURE 15.8. A virtually synchronous replicated update algorithm

at least one synchronous global execution that would have yielded the same outcome. To see this, observe that $\rightarrow$ for this system is a set of paths like the one seen in FIGURE 15.8, each consisting of a sequence of *write* and *pass* operations. These paths cross when a token is passed to some process p that subsequently receives a second token (FIGURE 15.9). Such a situation introduces edges that relate operations in the former path to operations in the latter. Similarly, if a process reads a data item, all the subsequent actions it takes will be ordered after all the previous updates to that data item. Although it may be hard to visualize, the resulting $\rightarrow$ relation is an acyclic partial order. It can therefore be extended into at least one total order, and in general many such orders, each of which describes a synchronous global execution that would yield the same values in all the variables as what the processors actually saw.

Thus, although the update algorithm is completely asynchronous and no process ever delays except while waiting for a token request to be granted, the execution is indistinguishable from a completely synchronous one such as would result from using a quorum write (Herlihy, 1986b) for each update. The performance of our algorithm is much better than that of a synchronous one, because a synchronous update involves sending messages and then waiting for responses, whereas an asynchronous update sends messages without stopping to wait for replies. No process is ever delayed in the execution illustrated by the figure, except when waiting for a token to be passed to it.

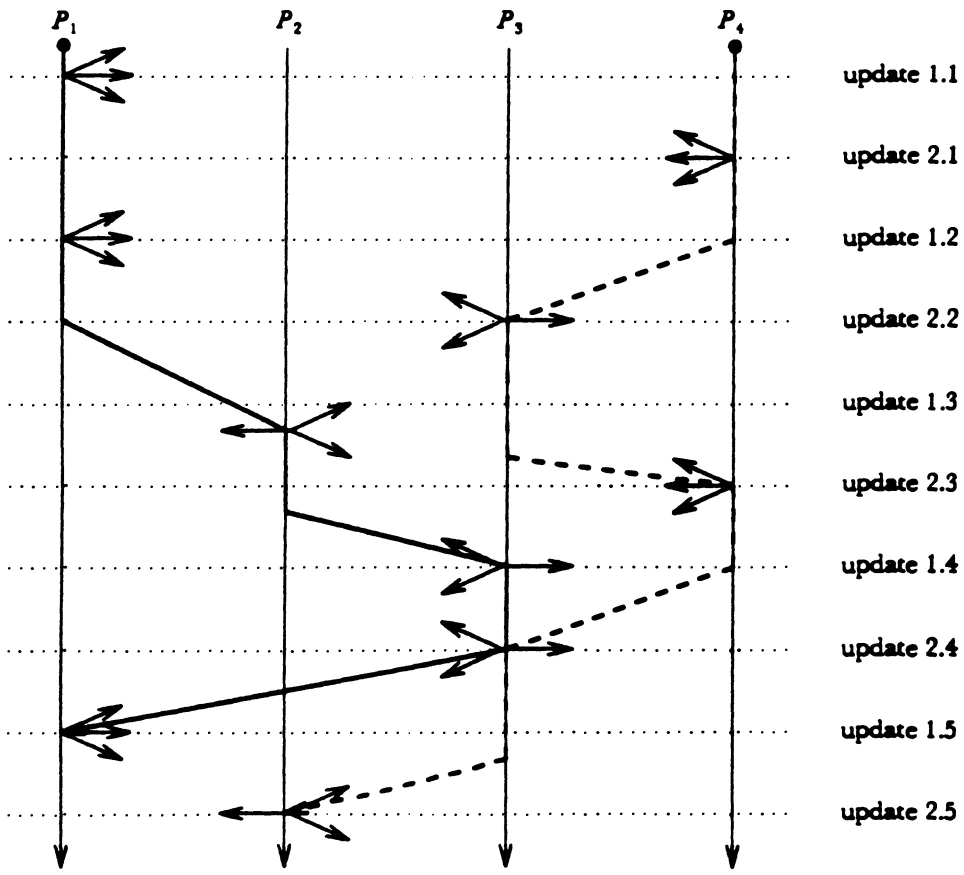

FIGURE 15.9. Replicated updates with multiple tokens

A similar analysis can be undertaken for replicated data with local read- and replicated write-locks, although we will not do this here. The existence of local read-locks implies that write-locks must be acquired synchronously, with each process granting the lock based on its local state, and the write-lock considered to be held only when all processes have granted it. This leads to an algorithm in which read-locks are acquired locally, write-locks are acquired using a synchronous group broadcast, and updates and lock release are done using asynchronous broadcasts. In a refinement, the breaking of read-locks after failures can be prevented by asynchronously broadcasting information about pending read-locks in such a way that any updates that depend on a read lock are related to the read-lock broadcast under $\rightarrow$. This method was first proposed by Joseph and Birman (1986).

15.7.7. Dealing with failures

The analysis of the preceding section overlooks failures and other dynamic group membership changes. In many applications one wishes to deal with such events explicitly, for example by granting the token to the next pending request in the event that the current holder fails. Recall that group membership changes must be totally ordered with respect to other events in order to ensure the virtual atomicity of broadcast delivery. Since ISIS does this, any broadcast sent in the token or update algorithm will be received by all members of the group that stay operational, and in the same view of the current membership.

Say that the rule we would like to implement is:

> *All group members monitor the holder of the token. If the holder fails, the oldest process in the group takes over and passes the token on its behalf.*

For a solution to be correct, we will need to be able to establish two things about the system. First, because the rule depends on the ordering of group members by age, this information must be consistent from member to member. Group views have this property in ISIS.

Secondly, we need to know that any view change reporting a failure will be ordered after all broadcasts done by the failed process. This ensures that if the failed process did a pass before dying, then either no process saw it happen or all processes saw the broadcast and are already watching the new holder. That is, if X_p^i is the i'th action taken by p and F_p denotes the event reporting the failure of p, we need $\forall p,i: X_p^i \rightarrow F_p$. Certainly, in any synchronous execution a failed process takes no further actions, hence this condition will also hold for any virtually synchronous execution.

Thus, one could readily implement a fault-tolerant token-passing algorithm in a virtually synchronous environment.

Notice that the failure ordering property links the atomicity of one broadcast to the atomicity of a subsequent one. A conventional atomic broadcast places an all or nothing requirement on broadcast delivery. But, this does not rule out the transmission of a broadcast a that will not be delivered anywhere because of a failure, followed by the transmission of a broadcast b from the same sender that will be delivered. In a virtually synchronous system, such behavior is not permitted.

15.8. Other virtually synchronous tools

We have seen virtually synchronous solutions to two problems in the list of tools enumerated at the start of this chapter: replicated data management and synchronization. Let us briefly address the other problems in the list.

15.8.1. Distributed execution

There are several ways to distribute an execution over a set of sites in a virtually synchronous setting. The ISIS toolkit supports all of the following:

Pool of servers: The Linda system illustrates a style of distributed execution that we call the *pool of servers*. In this approach, a set of processes share a collection of work-description messages, extracting them one at a time, performing the indicated operation, and then placing a completion message back into the pool for removal by the process that initiated the work. The approach is simple and lends itself to environments where the processes composing a service are loosely coupled and largely independent of one another. It can be made fault-tolerant by maintaining some sort of "work in progress" trace that can be located when a process is observed to fail. On the other hand, this method of distributed computing is potentially costly because it relies so heavily on synchronous operations. Were such a system to access its tuple space frequently, a bottleneck could develop.

Redundant computation: A redundant computation is one in which a set of processes perform identical operations on identical data. The approach was first proposed by Cooper for use in the Circus system (Cooper, 1985). Redundant computation has the advantage of fault-tolerance, and when the operation involves updates to a replicated state it is often the most efficient way to obtain a replicated 1-copy behavior. On the other hand, it is unclear why one would want to use a redundant computation for an operation that does not change the state of of the processes involved. With the exception of a realtime system operating under stringent deadlines, where it might increase the probability of meeting the deadline, such an approach would represent an inefficient use of computational resources. And, in a computation that is at all deterministic, the method is clearly inapplicable.

Redundant computation is easily implemented in a virtually synchronous environment. The event initiating a computation is broadcast to all the processes that will participate in the computation. They all perform the computation in parallel and respond to the caller, sending identical results. The caller can either continue computing as soon as the first result is received, or wait to collect replies from all participants.

ISIS does not permit redundant computations to be nested unless the application makes provisions to handle this possibility. In contrast, the Circus system supports nested redundant computations in a way that is transparent to the user, even permitting replicated callers to invoke non-idempotent operations and operations implemented by a group with a replication factor different from that of the caller. Cooper discusses these problems, as well as mechanisms for guarding against incorrect replies being sent by a faulty group member, in Cooper (1985).

Coordinator-cohort computation: A coordinator-cohort computation is one in which a single process executes a request while other processes back it up, stepping in to take over and complete the request if a failure occurs before the response is sent (Birman and Joseph, 1987a). Such a computation could make good use of the parallelism inherent in a group of processes, provided that different coordinators are picked for different requests (in this way sharing the load). Moreover, it can be used even in non-deterministic computations. However, if the distributed state of the processes involved is changed by a request, the coordinator must distribute the updates made to its cohorts at the end of the computation. In situations where a redundant computation was a viable possibility, the cost of this style of updating should be weighed against that of running the entire computation redundantly and eliminating the communication overhead.

Implementation of a coordinator-cohort computation is easy in a virtually synchronous setting. The request is broadcast to the group that will perform the computation. The caller then waits for a single response. In many applications, the broadcast can be done using a one-phase protocol such as CBCAST, although this decision requires an analysis similar to the one used for the token passing example. The participants take the following actions in parallel. First, they rank themselves using such information as the source of the request, the current membership of the group doing the request, and the length of time that each member has belonged to the group. Since all see the same values for all of these system attributes, they all reach consistent decisions. The coordinator starts computing while the cohorts begin to monitor the membership of the process group. The coordinator may disseminate information to the cohorts while doing this, or use mechanisms like the token for synchronization. When the coordinator finishes, it uses CBCAST to atomically reply to the caller and (in the same broadcast) send a termination message to the cohorts. If a coordinator fails before finishing, its cohorts react as soon as they observe the failure event (recall that a broadcast sent prior to the failure is delivered before the failure notification). The cohorts recompute their ranking, arriving at a new coordinator that terminates the operation. If the original coordinator sent information while computing, there are a number of options: the cohorts can spool this and discard it if a failure occurs, or could apply it to their states and take over from the coordinator by picking up from where it died.

Unless the application is sensitive to event orderings, this algorithm can be implemented with asynchronous CBCAST's. As in the case of the token algorithm, a highly concurrent execution would result.

Subdivided computation: A subdivided computation arises when each participant does *part* of a requested task. The caller collects and assembles these to obtain a complete result. For example, each member of a process group might search a portion of a database for items satisfying a query, with the result being formed by merging the partial results from each subquery. As in the case of a coordinator-cohort computation, the participants in a subdivided computation can draw on a number of properties of the environment to divide the computation. Provided that all use the same decision rule, they will reach the same decision. Dealing with failures, however, is problematic in this case. A simple solution is to identify the results as, e.g., "part 1 of 3." A caller that receives too few replies because some processes have failed can retry the query, or perhaps just the missing part.

15.8.2. System configuration and reconfiguration

Above, we used the term "configuration" as a synonym for the view of processes groups and processors in the system — that is, a list of the operational members, ordered by age. However, some systems have a software configuration that augments this view-based configuration and is also used for deciding how requests should be processed. This suggests that software designers need access to a broadcast primitive like the one ISIS uses to inform process group members of group membership change. The GBCAST primitive can be used for this purpose. Because GBCAST is atomic and totally ordered with respect to both CBCAST and ABCAST, one can use it to transmit updates to a replicated configuration data structure shared by the members of a process group. Such an update would otherwise be implemented just like any other update to replicated data, but because of the strong ordering property of the GBCAST, all processes see them in the same order with respect to the arrival of other messages of all kinds. Thus, when a request arrives or some other event is observed, the amended configuration can be used as part of the algorithm for deciding how to respond.

15.8.3. Recovery

When a process recovers, it faces a complex problem, which in ISIS is solved by the *process-group join tool*. A recovering process starts by attempting to rejoin any process groups which the application maintains. When invoked, the tool checks to see if the specified process group already exists and if any other process is trying to recover simultaneously. A given process will observe one of the following cases:

1. The group never existed before and this process is the first one to join it. The group is created and the caller's initialization procedure invoked. If two processes restart simultaneously, ISIS forces one to wait while the other recovers.
2. The group already exists. After checking permissions, the system adds the joining process to the group as a new member, transferring the state of some operational member as of *just before* the join took place. The transfer is done by repeatedly calling user-provided routines that encode the state into messages and then delivering these to user-provided routines that decode the messages in the joining process. The entire operation is a single virtually synchronous event. All the group members see the same set of events up to the instant of the join, and this is the state that they transfer. After the transfer, all the members of the group (including the new member) see the membership change to include the new member, and subsequently all see exactly the same sequence of incoming requests (subject to the ordering constraints of the protocol used to send those requests).
3. The group previously existed but experienced a total failure. The handling

of this case depends on whether or not the group is maintaining non-volatile logs and, if so, whether or not this process was one of the last to fail and consequently has an up-to-date log; Skeen (1985) gives an algorithm for deciding this. The former case is treated like case (1). In the latter, a recovery is initiated out of the log file. If the process is not one of the last to fail, the system delays the recovery until one of the last group members to fail has recovered, and then initiates a state transfer as in case (2).

In ISIS, a log file consists of a checkpoint followed by a series of requests that modify the state. The checkpoint itself is done by performing a state transfer (see above) into a log file. Thus, recovery out of a log looks like a state transfer from a previously operational member, followed by the replay of messages that were received subsequent to the checkpoint and prior to the failure. Management and recovery from logs in a virtually synchronous setting has been examined by Kane (1988).

ISIS obtains its join mechanism by composing several of the tools described earlier. For example, the state transfer is done using the coordinator cohort tool, described above. To ensure that this is done at a virtual instant in time, a GBCAST is used to add the new member to the existing process group, and the transfer is triggered just before reporting the membership change to the group members (including the new member). The mechanism is not trivial to implement, but is still fairly simple. Similarly, solutions to the other aspects of the problem are constructed out of broadcast protocols and reasoning such as what we described above for the token passing algorithm.

The ISIS recovery tool is illustrative of an interesting aspect of virtual synchrony. On the one hand, the designer thinks about recovery as a series of simple steps: a site restarts, the recovery manager executed the client's program, the program requests that it be added, the request is authenticated, a series of state transfer messages arrive and finally a new view becomes defined showing the new member. The sequence is always the same, and no other events ever occur while it is underway. On the other hand, the same designer treats state transfer as an atomic event (a sort of "transaction") when writing software that may interact with a group while a recovery may be taking place. The recovery either has not happened yet or it is done, seen from outside there are no other possibilities. Because this eliminates a huge number of possible race conditions and cases to deal with, a complex mechanism is rendered simple enough for a novice to use correctly.

15.9. Orthogonality issues

It was observed that for a set of tools to be of practical value they must permit a step-by-step style of programming. For example, if we build a distributed program using some set of tools, and then extend it in a way that requires an additional replicated variable, the only code needed should be for managing and synchronizing access to the new variable. It should not be necessary to reexamine all the previous code to ensure that no unexpected interaction will creep in and break some preexisting algorithm. We say that a set of tools are orthogonal to one another if they satisfy this property.

A desirable characteristic of the virtually synchronous environment is that orthogonality is immediate in algorithms that require just a single broadcast event, because these broadcasts are virtually synchronous with respect to other events in the system. For example, since updates to a replicated variable appear to be synchronous, introducing a coordinator-cohort computation for some [illegible] purpose in a program that uses such updates should not "break" the replicated data mechanism. More complex mechanisms, such as the ISIS recovery mechanism, are made to look like a single synchronous event, even when they involve several distinct subevents. A consequence is that one can build software in ISIS by starting with a non-distributed program that accepts an RPC-style of interaction, then extending it into a distributed solution that uses a process group and replicated data, introducing a dynamically changing distributed configuration mechanism, arranging for automated recovery from failure, and so forth. Each change is virtually synchronous with respect to the prior code, hence no change will break the pre-existing code. The same advantage applies in a setting like the Linda system: software here can be developed by debugging a single process that uses the tuple-space primitives, and once it is operational replicating this process to the extent desired.

15.10. Scaling, synchrony and virtual synchrony

We observed that a genuinely synchronous approach to distributed computing will have scaling problems. Above, only HAS implements such an approach, and its performance degrades quite explicitly as a function of the number of sites in the network, because a larger network has larger expected delays over its communication links. This increases the minimum delay before a broadcast can be delivered, and because HAS does not support an asynchronous broadcast the performance of application software is directly impacted.

A system like ISIS has a slightly different problem. Here, the basic protocols are essentially linear in the size of process groups (see Birman, Joseph, and Schmuck (1989)). However, several parts of ISIS involve algorithms that scale with the number of sites in the network. To address this issue, a recent version of ISIS introduced a notion of scope into the system. The idea is to restrict these algorithms to small collections of sites in a way that does not compromise the correctness of the overall system. The resulting system has been scaled up to more than one hundred sites without imposing a severe load on any machine, although process *groups* must not grow to include more than 20 or 30 members (here, we assume a 10 Mbit network and 2—5 MIPS workstations). Current research is now focused on introducing a notion of hierarchy for use when process groups get very large. These figures are based on current experience with those aspects of the ISIS system that will not change when [illegible]er algorithms are installed. Thus, ISIS potentially [illegible]es to moderately large networks, but is unlikely to scale up into geographically distributed settings with tens of thousands or millions of sites. An open question is whether there exists some other architecture that would yield virtual synchrony and high levels of concurrency, as does ISIS, but would scale without limit.

Finally, consider the quorum schemes, which [illegible] achieve virtual synchrony. These degrade in a way that is completely determined by the quorum size and the number of failures to be tolerated. While process groups stay [illegible], one would expect bounded performance limited by RPC bandwidths, and poorer than what can be achieved using asynchronous protocols. The quorum approach is clearly [illegible]table for systems that replicate data at very large numbers of locations.

To summarize, there seems to be good reason to view virtual synchrony as an effective programming tool for small and medium size ne[illegible]ks, and with the use of hierarchical structuring techniques should even be able to encompass a typical medium-size factory or company. In much larger settings, other approaches yielding weaker correctness guarantees would be needed.

It should also be noted that our collection of tools focuses on programming "in the [illegible]". The design and implementation of software for a factory requires something more: a methodology for composing larger systems out of smaller components, and perhaps a collection of tools for programming in the large. The former would consist of a formalism for describing the behavior of system components (which could themselves be substantial distributed systems) and how components interact with one another, independently of implementation. The latter would include software for cooperative application development, monitoring dependencies between components of a large system and triggering appropriate action when a change is made, file sys[illegible] with built in replication, and mechanisms with which the network can be asked to monitor for arbitrary user-specified events and to trigger user-specified actions when those events occur. These are all hard problems, and any treatment of them is beyond the scope of this [illegible]ussion. Moreover, the current state of the art in these areas is painfully deficient. Substantial progress is needed before it becomes practical to talk about building effective and robust network solutions to large-scale problems.

15.11. An example of ISIS software and performance

It might be interesting to see a sample of a typical ISIS program. The program shown below solves the drilling problem in ISIS. In contrast to the Linda solution, the method is fault-tolerant and supports dynamic process recovery. As before, the code will be in two parts: the code for a process that issues the original work request to the cell controller, and the distributed algorithm run in parallel by the control processes. We start with the code for making a request:

```
/* Define a type called hole_t for describing holes */
typedef struct
{
    /* Description of hole */
    int     h_x, h_y, ....;          /* Description of the hole */

    /* Runtime variables set by algorithm */
    address h_drill;                 /* Process that will drill it */
    int     h_state;                 /* Status, see below */
} hole_t;

#define     H_NULL       0           /* Initial state */
#define     H_ASSIGNED   1           /* h_drill has been set */
#define     H_DRILLING   2           /* Drilling underway */
#define     H_DONE       3           /* Hole completed */

main()
  {
    address driller;
    int nholes, nreplies, checklist[MAXHOLES], ntocheck;
    hole_t holes[MAXHOLES];

    ... initialize nholes and holes[0..nholes-1] ...

    /* Lookup address of drill service */
    driller = pg_lookup("/bldg14/cell22-a/driller");

    nreplies = cbcast(driller, WORK_REQ,
                      /* Message to broadcast */
                      "{%d,%d,...,%a,%d}[]", holes, nholes,
                    1 /* One reply wanted */,
                      /* Reply format */
                      "{%d}[]", checklist, &ntocheck);

    if(nreplies != 1)
        panic("Drill service is not available\n");
    if(ntocheck != 0)
    {
        printf("Job requires manual recheck.  Please check:\n");
        for(i = 0; i < ntocheck; i++)
        {
            hole_t *h = &holes[checklist[i]];
            printf("Hole at %d,%d ..\n", ...);
        }
        printf("Type <cr> when finished rechecking: ");
        while(getchar() != '\n') continue;
    }
    ... etc ...
  }
```

program imports the list of entry points from the drill service, which d es the **WORK_REQ** entry to which the work request is being transmitted. To a reader familiar with the C programming language, the code will be self-explanatory except for the arguments to `cbcast`, which are the group to transmit to (a long form accepting a list of groups is also supported), the entry point to invoke in the destination processes, the format of the data to transmit (here, an array of structure elements), the array itself and its length, the number of replies desired (1), the format of the expected reply (an array of integers), a place to copy the reply, and a variable that will be set to the length of the reply array.

The cell controller requires more code:

```
/* A typical drill controller */

# include "hole-desc.h"

main()
  {
    /* Bind the two entry points to handler routines */
    isis_entry(WORK_REQ, work_req);
    isis_entry(DRILLING, drilling);
    /* Start ISIS lightweight task subsystem */
    isis_mainloop(restart_task);
  }

/* Task to restart this process group member */
restart_task()
  {
    /* Join or create group, obtain current state */
    driller = pg_join("/bldg14/cell22-a/driller",
        /* On first time create, call first_time_init */
            PG_INIT, first_time_init,
        /* On joining an existing system, do state transfer */
            PG_XFER, state_xfer_out, state_xfer_in,
        /* Call monitor_routine on membership changes */
            PG_MONITOR, monitor_routine,
        0);
   }

/* Global variables */
int    checklist[MAXHOLES], ntocheck;

/* Reception of a new work request (WORK_REQ entry) */
work_req(msg)
  message *msg;
  {
    int nholes;
    hole_t holes[MAXHOLES];
    pgroup_view *pgv = pg_getview(driller);

    msg_get(msg, "{%d,%d,...,%a,%d}[]", holes, &nholes);
    for(n = 0; n < nholes; n++)
    {
        hole_t *h = &holes[n];
```

```
            h->h_who = schedule(h, pgv);
            if(<first hole assigned to this process>)
                h->h_state = H_DRILLING;
            else
                h->h_state = H_ASSIGNED;
        }
        t_fork(drill_task);
        ntodrill = nholes;
        ntocheck = 0;
        cur_req = msg;
        send_rep();
      }

  send_rep()
    {
      t_wait(&work_done);
      if(pg_rank(my_address, driller) == 1)
            /* Oldest process replies for group */
            reply(cur_req, "{%d}[]", checklist, ntocheck);
      cur_req = (message*)0;
    }

  int drill_task_active;

  /* How many failures we can tolerate at a time */
  #define   N_FAULTS_TOLERATED    1

  drill_task()
    {
      int done_with, n;
      char answ[N_FAULTS_TOLERATED];
      ++drill_task_active;
      n = next_hole(my_address, holes);
      while(n != -1)
      {
          hole_t *h = &holes[n];
          drill_hole(h);
          done_with = n;
          n = next_hole(holes);
          /* Async. broadcast to inform others of my next action */
          cbcast(driller, DRILLING,
                    "%a,%d,%d", my_address, n, done_with,
                N_FAULTS_TOLERATED+1,
               "%c", &answ);
      }
      --drill_task_active;
    }

 /* Invoked when a DRILLING cbcast is done */
 drilling(msg)
   {
     msg_get(msg, "%a,%d,%d", who, &next, &done);
```

```
      /* Update status of holes list */
      holes[done].h_state = H_DONE;
      if(next != -1)
          holes[next].h_state = H_DRILLING;

      /* When done, awaken send_rep() */
      if(--ntodrill == 0)
          t_signal(&work_done);

      /* Confirm that we got the message */
      reply(msg, "%c", '+');
    }

  /* When a process fails, reassign its remaining work */
  monitor_routine(pgv)
    pgroup_view *pgv;
    {
         int must_drill = 0;
         if(pgv->pgv_event != PGV_DIED)
             return;
         for(h = holes; h < &holes[nholes]; h++)
             if(h->h_who == pgv->pgv_died)
             {
                    if(h->h_state == H_ASSIGNED)
                    {
                        h->h_who = schedule(h, pgv);
                        if(addr_ismine(h->h_who))
                            ++must_drill;
                    }
                    else if(h->h_state == H_DRILLING)
                    {
                        h->h_state = H_DONE;
                        checklist[ntocheck]++ = h-holes;
                        if(--ntodrill == 0)
                            t_signal(&work_done);
                    }
             }
         if(must_drill && drill_task_active == 0)
              t_fork(drill_task);
    }
```

The above code is certainly longer than for the Linda example, and it looks far more more complex [illegible] the Linda example. However, the Linda example was not fault-tolerant and did not address the scheduling aspects of the problem. Moreover, our solution is ac[illegible]y quite simple. It works as follows.

Each controller [illegible]ess joins a driller process group. The group as a whole receives each request by accepting a message to the `work_req` entry point. In parallel, all members sched[illegible] the work, noting which hole each of [illegible] other processes is currently drilling and marking all others as assigned. A lightweight task is forked into the background to do the actual drilling; it will share the address space cell controller with [illegible] task running `work_req`, using a non-preemptive "monitor" style of mutual exclusion under which only one task is executing at a time, and context switching occurs only when a task pauses to wait for something. The `work_req` task now waits for drilling to be completed.

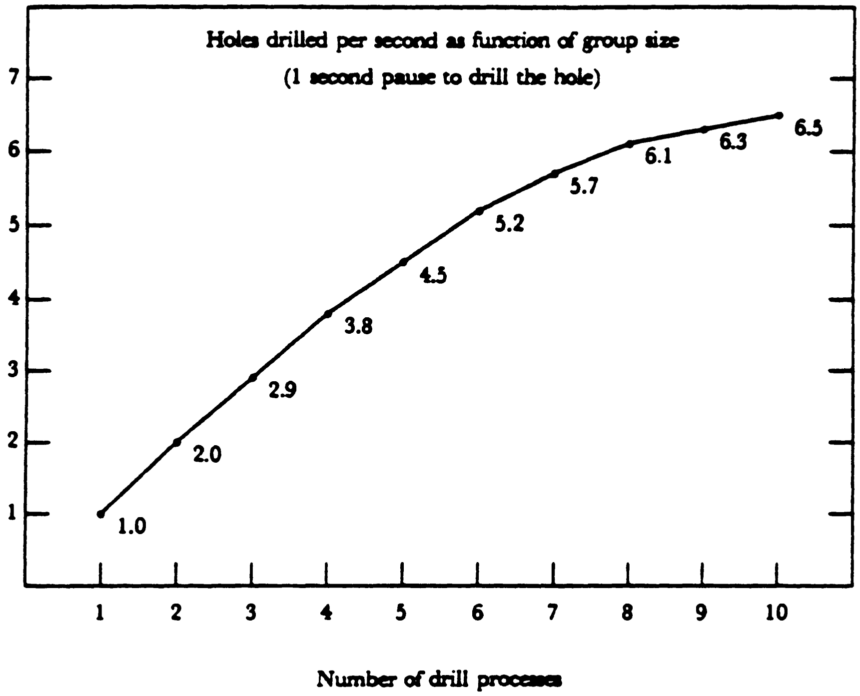

FIGURE 15.10. Holes drilled per second with a one-second per-hole delay.

The `drill_task` operates by drilling the next assigned hole, then broadcasting to all group members when it finishes this hole and moves on to the next one. The broadcast must be done synchronously, waiting until enough replies are received to be sure that the message has reached at least `N_FAULTS_TOLERATED` remote destinations (because the sender will receive a reply to its own message, we actually wait for one more reply above this threshold). The point here is to be sure that even if `N_FAULTS_TOLERATED` drill processes crash, the broadcast will still be completed because some operational process will have received it. Each group member marks the previous hole as `H_DONE` and the next one as `H_DRILLING` when this broadcast arrives.

If a process fails, the other group members detect this when their monitor routines are invoked by ISIS. They reassign work, moving any hole that the failed process was actually drilling to the check list. Any process that has ceased drilling (and hence no longer has an active `drill_task` spawns a new one at this time.

A normal ISIS application would also include code for initializing the group at cold-start time and for transferring the state of the group to a joining member, by encoding it into one or more messages. We have omitted this code above.

What about performance? FIGURE 15.10 and FIGURE 15.11 graph the performance of this application program, in holes-per-second drilled by the entire group as a function of the number of members. We generated these figures on a network of SUN 3/60 workstations, otherwise idle, running release 3.5 of the SUN operating system and communicating over a 10Mbit ethernet. FIGURE 15.11-a was based on a control program for which the simulated delay associated with moving the drill units and drilling holes was 1-second per hole. FIGURE 15.11 used a delay of zero.

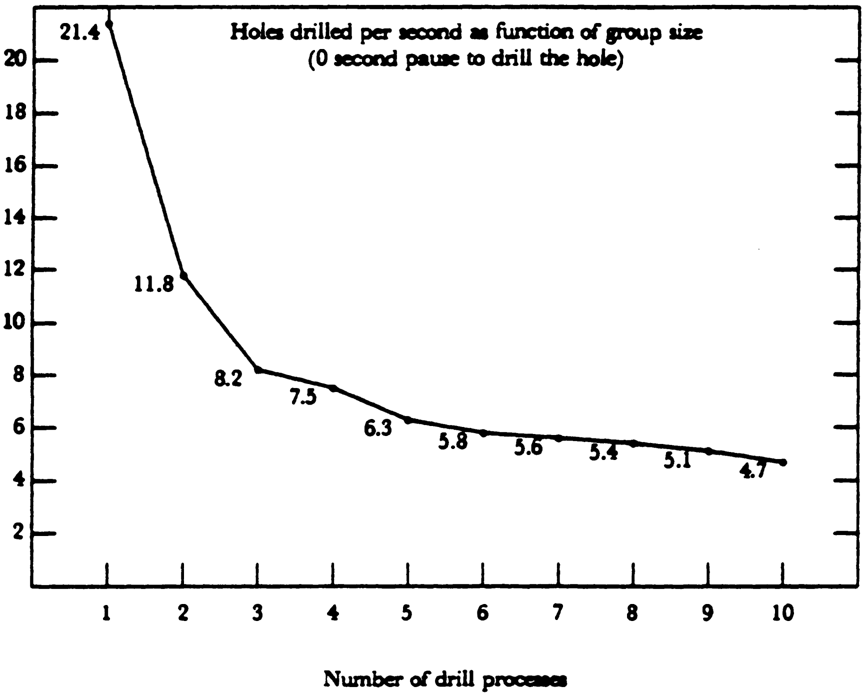

FIGURE 15.11. Holes drilled per second with a zero-second per-hole delay.

In the absence of any ISIS overhead, the first graph would show a linear speedup and numbers would all be infinite in the second graph. Thus, the communication overhead imposed by this version of ISIS becomes significant when the group reaches six members, limiting the attainable speedup for drilling holes with this delay factor. Since the number of messages sent per second grows as the square of the size of the group in this example, these curves are not unreasonable ones. More detailed performance figures for ISIS are available in Birman and Joseph (1987a), and Birman, Joseph, and Schmuck (1989).

15.12. Theoretical properties of virtually synchronous systems

We conclude the chapter with a review of some theoretical results relevant to the behavior of virtually synchronous systems.

15.12.1. How faithful can a virtually synchronous execution be to the physical one?

A system like ISIS seeks to provide the illusion of a synchronous execution while actually executing asynchronously. Moreover, unlike Linda or HAS, failures are "events" in the virtually synchronous execution model used by ISIS. This leads to limits on the extent to which the model can be faithful to reality. For example, it is impossible to ensure that a virtually synchronous execution will present failures in the precise sequence that physically occurred with respect to other events. Specifically, in a situation where the system is about to deliver a broadcast, it cannot prevent a physical failure from occurring just as the broadcast delivery is taking place. At one destination, the failure has occurred "after" delivery, but at the other it is "before" delivery. From this we see that a system like ISIS might sometimes be forced to claim that a message was delivered to a process that had actually crashed before delivery took place.

A relevant theoretical result (Fisher, Lynch, and Patterson, 1985) shows that it is impossible to reach distributed agreement in an asynchronous system subject to failures. Additional work along these lines was done by Hadzilacos and reported in Hadzilacos (1984). These results limit what is achievable in a virtually synchronous system. In particular, this establishes that ISIS cannot avoid all risk of incorrectly considering an operational site to have failed.

On the other hand, it is possible for a system to avoid claiming anything inconsistent with the *observable* world. This is done by introducing agreement protocols to decide what picture of a fundamentally uncertain event to provide in its synchronous world model, and then present this to its users in a consistent manner. This is what ISIS does. Unless a failed site or process recovers and can be queried about what it observed just before failing, code that runs in ISIS can never encounter an inconsistency. Moreover, when there is some question of ensuring that all processes have really observed a broadcast or other event, this can be arranged by briefly running the system synchronously — for example, by asking those processes to reply after they have seen the event and waiting for the replies. This is comparable to deferring external actions in a transactional system until the transaction has reached the prepared-to-commit stage.

15.12.2. State machine approach

Above, we suggested that virtual synchrony be viewed as an extension of transactional serializability that introduces process groups and atomic group addressing while eliminating the transactions. Virtual synchrony is at least as closely related, however, to work that was done on a theoretical abstraction called the *state machine* approach to distributed computing. State machines were originally introduced by Lamport, and work in the area is surveyed by Schneider (1986). In this approach, a (static) set of processes interact through a logically centralized service termed the *state machine*; the machine choses an order in which requests should be executed and delivers them to the partici[illegible]ts. In the terms of this chapter, a state machine implements a closely synchronous environment. In most theoretical treatments, state machines are used as a fault-tolerance mechanism, and described in terms of a Byzantine failure model. [illegible] may be one reason that their practical value was not immediately perceived. ISIS can then be understood as a state machine impl[illegible]tation that uses a series of optimizations to make the approach viable in an environment subject to a less difficult class of failures. To the best of our [illegible]ledge, this [illegible] was never [illegible] directly in a state machine context.

15.12.3. Representing and using IPC context information

There has been other work on communication mechanisms that preserve some form of "con[illegible] information", which CBCAST does by constraining the order in which messages are delivered. For example, Jefferson's *virtual time* approach implements a causal delivery ordering constraint using rollback (Jefferson, 1985). The mechanism operates in a point-to-point communication setting where messages are timestamped and must be delivered to each process in increasing timestamp order (this is common in event-based simulation systems). The problem that arises is that although processes are constrained to use monotonically in[illegible] timestamps for their [illegible]ions, it is impossible to predict just when any given process will need to transmit to any other. If a message is received with timestamp t, it should be delayed if some other process might send a message with timestamp $t' < t$ to the same destination. But, without waiting for all processes to advance beyond time t, how can this property be insured? A virtual time system o[illegible]tes by by [illegible] the "optimistic" assumption that no [illegible] earlier message will be sent, permitting the delivery of interprocess messages as soon as they arrive. After delivering a m[illegible] at time t, if a message does arrive with timestamp $t' < t$, the system simply rolls the destination process back to a time prior to t' ("unsending" any m[illegible] it issued during this period, which may trigger further rollbacks), then delivers t' and t in the correct order. The scheme requires that the system be able to [illegible] checkpoints and that rollback be cheap. The system of Strom and Yemeni (1985) implements a closely related programming language.

Notice that the timestamping scheme described above is really intended to represent time during a discrete simulation, and hence has a different purpose than the $\rightarrow$ o[illegible]tor introduced earlier. In contrast, Peterson (1987) (Peterson, Buchholz, and Schlichting, 1989) has developed a communication mechanism that rep[illegible]nts $\rightarrow$ explicitly and then uses this to enforce ca[illegible] delivery orderings. His system, Psync, includes a s[illegible]l amount of event ordering [illegible]rmation in messages that are sent. On reception of a message, a process can invoke simple primitives to test whether there may be outstanding prior messages, or to compare the orders in which two messages were sent. In effect, they permit the interrogation of $\rightarrow$. Peterson has completed an implementation of these primitives in the X-kernel, and used this to build several Psync applications. These include reliable broadcast protocols with the ordering properties of CBCAST, ABCAST and GBCAST, although lacking dynamic process group addressing.

15.12.4. When can a problem be solved asynchronously?

Schmuck has looked at the question of when a system specified in terms of synchronous broadcasts can be run correctly using asynchronous ones (Schmuck, 1988). He defines a system to be *asynchronous* if it admits an implementation in which every broadcast can be delivered immediately to its initiating process, with remote copies of the message being delivered sometime later. Failure broadcasts are not considered, although they could be added to the model without changing any of the results.

Define S_{async} to be the class of all system specifications describing problems that can be implemented in this efficient, asynchronous manner. Schmuck introduces the concept of a *linearization operator*, a function that maps certain partially ordered sets of events to legal histories. In a theorem he shows that for all specifications S:

$$S \in S_{async} \Leftrightarrow \exists a \textit{ linearization operator for } S.$$

He proves the only-if direction by showing how to construct an implementation for a specification S, based on its linearization operator, using a communication primitive similar to CBCAST. The other direction is proved by contradiction. The result establishes that Schmuck's implementation method is *complete* for the class S_{async}, i.e. the method yields a correct implementation for all specifications $S \in S_{async}$.

Schmuck's construction method depends on finding a linearization operator for a given specification. Unfortunately, whether a specification S is in S_{async} is undecidable. It is immediate that there exists no general method for finding a linearization operator for S. However, Schmuck does propose methods for solving this problem for certain subclasses of S_{async}. The basic characteristic of these subclasses is that they have linearization operators determined entirely by commutativity properties of the broadcasts done in the system. Moreover, Schmuck shows how to construct [illegible] specifications, in which CBCAST is used as often as possible, but ABCAST is still available for situations in which CBCAST cannot be used. These results can be used to "automatically" construct a linearization operator, and hence an optimal asynchronous broadcast protocol, for a problem like the token-passing ones described above. Interestingly, when we showed that token request, token passing and replicated updates could be totally ordering them along the path the token follows, we essentially described the construction of a linearization operator for that problem. Thus, Schmuck's work formalizes a style of [illegible]t of important practical relevance.

Herlihy and Wing have also looked at the [illegible] of achieving "locally" ordered behavior in distributed systems. This work develops a theory of linearizability, a pro[illegible] similar to serializability, but observed from the perspective of the objects performing o[illegible]tions rather [illegible] from the perspective of the processes acting upon those processes (Herlihy and Wing, 1987).

15.12.5. Knowledge in virtually synchronous systems

Some recent work applies logics of knowledge to protocols similar to CBCAST and ABCAST. The former problem was examined by Taylor and Panangaden (1988), who develop a formalism for what they refer to as *concurrent common knowledge*. [illegible] kind of knowledge is obtained when an asynchro[illegible] CBCAST is performed by a process that subsequently behaves as if all the destinations received the message at the [illegible]t it was sent. In ISIS, such a process will never encounter evidence to contradict this assumption. Taylor and Panagaden formally characterize the power of this style of computation, and then use their results to analyze algorithms like the concurrent update discussed above.

Neiger and Toueg have examined the relationship between the t[illegible] ordering of events in an ABCAST protocol and the total ordering that results from incorporating a shared realtime clock into a distributed system (Neiger and Toueg, 1987). [illegible] characterize the settings under which a broad[illegible] algorithm written to use a distributed clock could be implemented using an ABCAST protocol and a logical clock (Lamport, 1978).

15.13. Acknowledgements

We are grateful to Frank Schmuck for detailed comments and suggestions that lead to substantial [illegible]ons to this manuscript. Nick Carriero's comments about the discussion of Linda and Ray Strong's comments about [illegible]t work on the HAS systems were extremely valuable. The above treatment also benefited from discussions with Ajei Gopal, Ken Kane, Keith Marzullo, Gil Neiger, Pat Stephenson, [illegible] Taylor, Sam Toueg, Doug Voigt, and many others.

15.14. Refer[illegible]

Apollo (1985).
The Apollo Domain File System.
Apollo Computers, Inc., 1985.

P. A. Bernstein and N. Goodman (1981).
Concurrency Control in Distributed Database Systems.
ACM Computing Surveys 13 (2) : 185-221, June 1981.

K. P. Birman and T. Joseph (1987a).
Exploiting virtual synchrony in distributed systems.
Proceedings Eleventh Symposium on Operating System Principles : 123-138, Nov. 1987.

K. P. Birman and T. Joseph (1987b).
Reliable Communication in an Unreliable Environment.
ACM Transactions on Computer Systems : 47-76, Feb. 1987.

K. P. Birman, T. Joseph, and F. Schmuck (1989).
Issues of scope and scale in the ISIS distributed toolkit.
1989, In preparation.

A. D. Birrell and B. J. Nelson (1984).
Implementing Remote Procedure Calls.
ACM Transactions on Computer Systems 2 (1) : 39-59, February 1984.

N. Carriero and D. Gelernter (1986).
The [illegible] S/Net's Linda Kernel.
ACM Transactions on Computer Systems 4 (2) : 110-129, May 1986.

D. R. Cheriton and W. Zwaenepoel (1985).
Distributed Process Groups in the V Kernel.
ACM Transactions on Computer Systems 3 (2) : 77-107, May 1985.

E. C. Cooper (1985).
Replicated distributed programs.
Proceedings Tenth Symposium on Operating System Principles : 63-78, Nov. 1985.

F. Cristian, H. Aghili, R. Strong, and D. Dolev (1986).
Atomic Broadcast: From Simple Message Diffusion to Byzantine Agreement.
IBM Research Report RJ 5244 (54244), July 1986.

F. Cristian (1988).
Reaching Agreement on Processor Group Membership in Synchronous Distributed Systems.
IBM Research Report RJ 5964 (59426), March 1988.

M. Fisher, N. Lynch, and M. Patterson (1985).
Impossibility of distributed consensus with one faulty process.
J. ACM 32 (2) : 274-382, Apr. 1985.

V. Hadzilacos (1984).
Issues of fault tolerance in concurrent computations.
Ph.D. Dissertation, Harvard University, June 1984.
Available as Technical Report 11-84.

M. Herlihy (1986a).
Optimistic Concurrency Control for Abstract Data Types.
Proceedings 5th Symposium on Principles of Distributed Computing: 206-217, Calgary, Alberta, Canada, 1986.

M. Herlihy (1986b).
A quorum-consensus replication method for abstract types.
ACM Transactions on Computer Systems 4 (1) : 32-53, Feb. 1986.

M. Herlihy and J. Wing (1987).
Linearizable abstract types.
Proceedings ACM Symposium on Principles of Programming Languages, Munich, Germany, Jan. 1987.

D. R. Jefferson (1985).
Virtual Time.
ACM Transactions on Programming Languages and Systems 7 (3) : 404-425, July 1985.

T. Joseph and K. Birman (1986).
Low cost management of replicated data in fault-tolerant distributed systems.
ACM Transactions on Computer Systems 4 (1) : 54-70, Feb. 1986.

K. Kane (1988).
Log-based recovery in asynchronous distributed systems.
Ph.D. Dissertation. , Dept. of Computer Science, Cornell University, expected Dec. 1988.

N. P. Kronenberg, H. M. Levy, and W. D. Strecker (1986).
VAXclusters: A Closely-Coupled Distributed System.
ACM Transactions on Computer Systems 4 (2) : 130-152, May 1986.
Presented at the Tenth Symposium on Operating System Principles, Orcas Island, Washington, December, 1985.

L. Lamport (1978).
Time, Clocks, and the Ordering of Events in a Distributed System.
Communications of the ACM 21 (7) : 558-565, July 1978.

B. Lampson (1986).
Designing a global name [illegible]ce.
Proceedings 5th ACM Symposium on Principles of Distributed C[illegible]ing: 1-10, Calgary, Canada , Aug. 1986.
1985 Invited Talk.

B. Liskov, D. [illegible], P. Johnson, and R. Scheifler (1987).
Implementation of Argus.
Proceedings of the Eleventh Symposium on Operating System Principles: 111-122, A[illegible] TX, 8-11 November 1987.
In *ACM Operating Systems Review 21*:5.

N. Lynch, B. Blaustein, and M. Siegel (1986).
Correctness conditions for highly available replicated databases.
Proceedings 5th ACM Symposium on Principles of Distributed Computing: 11-28, Calgary, Canada, Aug. 1986.

G. Neiger and S. Toueg (1987).
Substituting for [illegible]time and common knowledge in asynchronous distributed systems.
Proceedings 6th ACM Symposium on Principles of Distributed Computing: 281-293, Vancouver, Canada, Aug. 1987.

L. L. Peterson (1987).
Preserving Context Information in an IPC Abstraction.
Proceedings 6th Symp. on Reliability in Distributed Software and Database Systems: 22-31, Wil[illegible]burg, VA, March 1987.

L. L. Peterson, N. Buchholz, and R. Schlichting (1989).
Preserving and Using Context Information in Interprocess Communication.
ACM Transactions on Computer Systems, 1989.
Conditionally accepted.

R. Rashid et al. (1987).
Machine-Independent Virtual Memory Management for Paged Uniprocessor and Multiprocessor Architectures.
Proceedings 2nd ASPLOS: 31-39, Palo Alto, Ca., Oct. 1987.

F. Schmuck (1988).
The use of efficient b[illegible]cast protocols in asynchronous distributed systems.
Ph.D. Dissertation, Aug. 1988.

F. B. Schneider (1986).
The State Machine Approach: A Tutorial.
Technical Report 86-600, Cornell University, December 1986.

P. M. Schwarz and A. Z. Spector (1984).
Synchronizing Shared Abstract Types.
ACM Transactions on [illegible] Systems 2 (3) : 223-250, August 1984.
Also available in Stanley Zdonik and David Maier (editors), *Readings in Object-Oriented Databases*. Morgan Kaufmann, 1988. Also available as Technical Report CMU-CS-83-163, Carnegie Mellon University, November 1983.

D. Skeen (1985).
Determining the last process to fail.
ACM Transactions on [illegible] Systems 3 (1) : 15-30, Feb 1985.

R. Strom and S. Yemeni (1985).
Optimistic Recovery in Distributed Systems.
ACM Transactions on Computer Systems 3 (3) : 204-226, April 1985.

K. Taylor and P. Panangaden (1988).
Concurrent Common Knowledge: A New Definition of Agreement for Distributed Systems.
Proceedings Seventh ACM Symposium on Principles of Distributed Computing, Toronto, Canada, Aug. 1988.