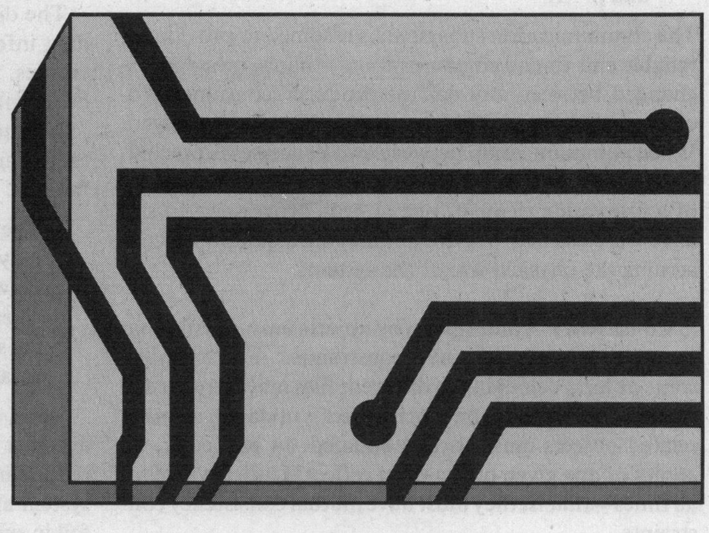
*An experimental system designed as part of INRIA's Project Sirius, Delta implements a distributed executive for real-time transactional systems.*

# A Distributed System for Real-Time Transaction Processing



**Gérard Le Lann**
**Project Sirius**
**INRIA**

The computing systems considered in this article are built from a variety of commonly available hardware components for processing, storage, and communication, such as minicomputers, disks, and buses. Physically distributed over short distances, these systems are usually labeled as multiple-processor computers or local area computer networks.

Target applications considered in this article are transaction-oriented and exhibit some significant real-time constraints. External users activate tasks that do not require large amounts of processing.

A transactional system manages data files that are dispersed over a number of storage elements. Real-time transactional systems exhibit specific requirements that greatly influence the design of a global executive intended for such systems.

We begin by outlining the basic problems that were addressed during the design of Delta, an experimental distributed transactional system built within the framework of Project Sirius. We then discuss some of the advantages of distributed architectures and conclude with a presentation of the basic aspects of Delta's distributed executive mechanisms.

## System requirements and problems

We are interested in those transactional systems that permit many users to access data files concurrently. In addition, we concentrate on physically dispersed systems that provide for access from various locations. Some examples of application areas would be reservation systems, medical information systems, and integrated offices. This section begins with a conceptual representation of a distributed transactional system, followed by a discussion of requirements.

**Model.** The system is accessed by external *agents*—human users, sensors, etc.—that activate transactions. A *transaction* is a set of elementary *actions*—such as DELETE, READ, WRITE, CREATE—that manipulate data objects grouped into files. The executive processes in charge of controlling the execution of transactions are called *producers*. In particular, producers are responsible for the firing of actions. Those executive processes in charge of performing actions are referred to as *consumers*.

Data objects containing vital information are replicated so the system can survive crashes of storage elements. By providing redundant hardware and data, a computing system can survive failures.

There are two ways to replicate data within a given data structure. One approach is to create $p$ partitions and replicate those data objects that are vital to the application over $n$ partitions ($n \leq p$). For two different replicated data objects, their corresponding sets of partitions may be different. On the other hand, all objects in the data structure may be equally vital to the application. In this case, an adequate number of copies of the whole data structure are maintained (full replication). Clearly, partitions containing replicated data objects should not be implemented on the same physical storage elements. For the sake of simplicity, we will assume in the following discussion that each partition is mapped on its own set of storage elements.

Figure 1 shows a *p*-partitioned system in which

- the set of data objects *A* is replicated in partitions 1, $p-1$, and *p*;
- the set of data objects *B* is replicated in partitions 1 and *p*; and
- the set of data objects *C* is replicated in partitions 1 and $p-1$.

The communication subsystem is assumed to provide for reliable end-to-end communcation—that is, messages exchanged between any pair of producer/consumer processes are retransmitted until they are acknowledged. When communication between two processes is precluded because of a crash of the communication subsystem or of a processing element, one or both processors—if still alive—are warned. We are not making assumptions concerning the physical size of the system.

**Consistency.** Values taken by objects must satisfy what are known as consistency constraints.[1] For example, some objects belonging to different files may be related in such a way that when one such object is updated, all other related objects must also be updated. In particular, all copies of any given object must reflect identical values at all times—that is, they must have mutual consistency constraints.

We assume that transactions may be activated at any time (i.e., have concurrent access to the files) from anywhere. Thus, they may possibly interfere with each other, with the consequence that consistency constraints could be violated, as in READ/WRITE conflicts. Every transaction is controlled from a unique producer.

A transaction may ask for an activation of an action at any time. Transactions, delimited by BEGIN and END commands, are viewed by the agents as being *atomic*—that is, indivisible. The system executive is responsible for guaranteeing that such a property holds at all times. Consequently, it is necessary to provide mechanisms at the distributed executive level that guarantee atomicity when faced with concurrent transactions.

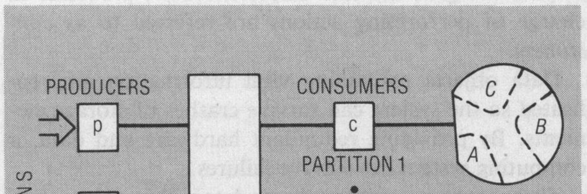
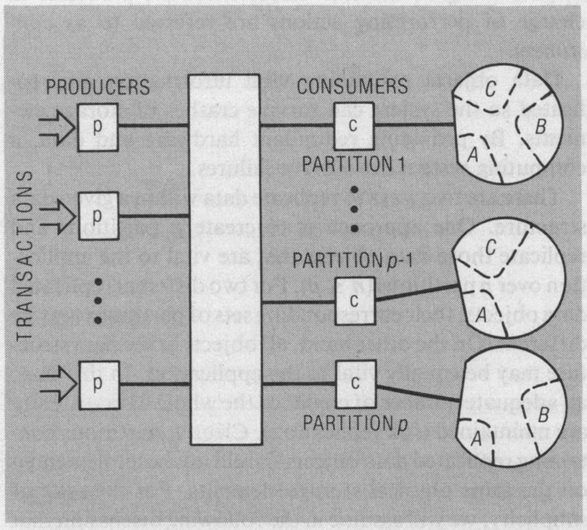



**Figure 1. A partitioned and partially replicated transactional system.**

**Robustness.** We assume that failures may impact the storage elements, the processing elements that host the producers and the consumers, and the communication subsystem. Such failures may preclude the correct completion of transactions. In particular, WRITE actions initialized from a transaction may be only partially performed, contradicting the atomicity requirement.

The distributed executive must maintain some redundant information for surviving or recovering from such crashes. When recovery is impossible, system files must be reinstalled in a state known to be consistent.

If data is replicated, it is also necessary to provide redundant access paths to the data in order to tolerate failures. This means two things:

- The various occurrences of a replicated data object may be accessed via different paths; interferences between concurrent accesses are possible.
- There should be no single point of control for checking such interferences, since it would be equivalent to creating a single point of failure in the system.

Thus, again, there is a need for a decentralized mechanism at the executive level that would take advantage of redundant hardware and data in order to guarantee high system availability. The communication subsystem may fail in such a way that the global system is split into more than one subsystem. Depending on the application requirements, one may decide that all subsystems or only one of them should be kept alive. In either case, a distributed executive mechanism is needed to allow several subsystems to be merged while communication failures are repaired. In particular, objects that were located in different subsystems must have values in agreement with consistency constraints. For example, all occurrences of a replicated object must have identical values.

**Extensibility.** In transaction-oriented environments, system configurations often are constantly being modified as, for example, more terminal equipment is needed, older devices are replaced by more sophisticated ones, or processing and storage elements are added to meet performance requirements. Computing systems based on modular architectures are particularly appropriate in these environments.

Physical modularity allows incremental extension and reduction, hardware redundancy for the sake of robustness, and relaxed maintenance constraints.[2] Plugging an element in and out should be straightforward tasks with modular hardware.

Physical modularity by itself, however, is not sufficient for obtaining dynamically extensible systems. The global executive must also be modular—that is, it should be implemented as a set of cooperative but autonomous local executives embedding decentralized control mechanisms, thus achieving mutual independence among the various system elements.

Extensibility and efficiency are related issues. For example, it is possible to improve the performance of a given system—such as its response time—by adding more processing and storage elements, provided that the executive mechanisms do not create artificial bottlenecks. If this aspect is overlooked, a computing system, although physically extensible, may not satisfy performance requirements.

Recently, a number of designers have been actively involved in developing distributed data-sharing systems that are efficient, robust, and extensible.[3-8] A crucial issue in this area is robustness.

## Basic features of Delta

We restrict our description of Delta to the following basic features which provide solutions for the problems discussed above:

- the decentralized mechanism that maintains the consistency of the files under concurrent access,
- the COMMIT protocol that permits correct termination of transactions in spite of producer/consumer crashes, and
- the recovery algorithm that brings any repaired consumer into a correct state.

**Decentralized mechanism for concurrency control.** Each producer is given a unique, permanent identity—an integer value—that defines its order on the set of producers materialized as a *virtual ring.*[9] Producers are viewed as being sequentially arranged along the virtual ring. Each producer at any time has a unique predecessor and successor. Every producer is required to regularly check its successor, *s,* on the ring by having *life messages* sent to the successor by the communication software. These messages must be acknowledged, which is possible only if some internal checking is successful. Successful internal checking is also a condition for having a producer issue life messages. In this way, checking transitivity is achieved along the virtual ring.

If several successive life messages are not acknowledged, then a reconfiguration is undertaken by the originating producer that sends a special message to the potential successor of its previous successor—$s+1$, $s+2$, etc. This procedure is repeated until a positive response is received. The ultimate situation is a virtual ring with only one producer. Repaired processing elements that host producers should be able to join the system at any time, which is possible via the virtual ring insertion protocol described elsewhere.[10]

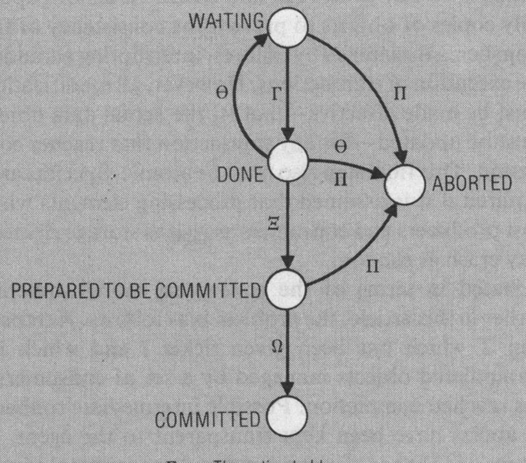
A particular message called the *control token* circulates on the virtual ring. This token carries a *sequencer* that delivers sequential and unique integer values called *tickets.* Each time a producer accesses the sequencer, the current value is delivered and the sequencer is incremented. Once tickets have been selected by a producer, the token is sent to the successor. This mechanism survives failures of both producers and the communication network.

Transmission of the token between adjacent producers is monitored through a positive acknowledgment + retransmission protocol. The token carries with it an integer value called the *cycle number* which is incremented for every complete revolution on the ring. This incrementation is performed by producer $x$ such that $x >$ successor ($x$). At all times, this producer is unique. The cycle number is used for detecting possible duplicates of the token as well as for deciding, upon a reconfiguration of the virtual ring, whether or not the token and the sequencer have been lost. The algorithm can be found elsewhere.[10]

The circulating sequencer mechanism is used to timestamp transactions submitted to producers by agents. Each transaction receives a unique ticket, and all actions invoked for a given transaction carry the value of the ticket allocated to that transaction. Every request for an action received by a consumer is acknowledged. Furthermore, a producer is notified when a requested action's state is changed, in terms of the five states defined in Figure 2.

For such actions as WRITE, CREATE, and DELETE, no modification actually takes place in the system files. These actions are processed within the execution context of the originating transaction; they work on a copy of the objects requested. Only when a producer decodes the END command (END may be either COMMIT or ABORT) from the agent do the actual modifications take place within the system files (in the case of a COMMIT) and only then are execution contexts destroyed.

When two transactions have conflicts at different consumers, the conflicts are resolved identically at all such consumers, and the decision is immediate (move from state *D* to state *A* or to state *W*). When an "older" action is received—that is, an action that carries a smaller ticket—a consumer may decide either to roll back (move to *W*) or abort (move to *A*) the "younger" action. When



| | |
|---|---|
| Γ: | The action holds the smallest ticket |
| Θ: | Reception of an action carrying a smaller ticket |
| Π: | Upon request of the transaction controller |
| Ξ: | The PREPARE TO COMMIT request has been acknowledged |
| Ω: | Reception of the COMMIT request |

**Figure 2. State-transition graph of an action managed by a consumer.**

an action is made to wait, other actions fired by the corresponding transaction are not altered. When an action is aborted, the corresponding initiator will abort all other related actions. Rollbacks and aborts are kept transparent to external agents. In case of an abort, the corresponding transaction will be resubmitted automatically by the original producer.

Tickets define a total ordering on the set of transactions initiated by producers. This condition is sufficient to guarantee consistency. A wide spectrum of decentralized mechanisms intended for solving the consistency problem exists. Our mechanism represents one end of the spectrum, since no information (except ticket values) is necessary for a consumer to correctly schedule competing actions. At the other end of the spectrum are those mechanisms based on actual representations of the system state, such as wait-for-graphs and conflict graphs.[5,11] In between are those mechanisms based on partial knowledge of the system state.[8] The trade-off is between the overhead incurred for collecting state information and savings from unnecessary rollbacks or aborts.

We believe that in real-time transactional systems, transactions do not represent large amounts of processing. Therefore, when acceptable by an application, rollbacks and aborts should not represent high processing costs when compared with the cost of exchanging state information across processing elements in a distributed system. This is one of the reasons we adopted this mechanism for synchronizing conflicting transactions in Delta.

**Atomic transactions under failures.** Transactions initiating WRITE, CREATE; and DELETE actions update only copies of objects to prevent the consistency of files from being threatened by failures, interrupting at random the execution of transactions. However, all modifications must be made effective—that is, the actual data objects must be updated—for any transaction that reaches completion. This final step also must be atomic. Special care is required if it is assumed that processing elements which host producers and consumers as well as storage elements may crash at random.

Stated in terms of the conceptual model presented earlier in this article, the problem is as follows. A transaction $T$ which has been given ticket $t$ and which has manipulated objects managed by a set of consumers $C$ has reached completion. Possible intermediate rollbacks or aborts have been kept transparent to the agent. By means of either a software command or a terminal control key, the agent indicates his willingness to commit $T$.

Let $p$ be the producer in charge of controlling $T$ when the COMMIT command is issued ($p$ need not be the initiator of $T$). The problem is that, although distributed, a COMMIT must be atomic. More precisely, we want to provide $p$ and consumers in $C$ with a COMMIT protocol that is reliable in case of failures and meets the requirement of mutual independence as much as possible.

For coping with the first of these two constraints, a two-phase COMMIT protocol similar to the one that Gray described[5] is convenient. The first phase begins when $p$ sends a PREPARE TO COMMIT message to each consumer in $C$. This message allows a consumer to tell $p$ if a given action can be committed locally before all other consumers are told to commit. It can also be used to provide a consumer with some new values to be committed which have not been computed locally. The first phase ends when every consumer involved has acknowledged this message (corresponding actions move from state $D$ to state $P$) and when all acknowledgments have been received by $p$. If one or more consumers ask for an abort, $T$ is aborted by $p$, and the other consumers are informed about this fact.

**Transactions initiating WRITE, CREATE, and DELETE actions update only copies of objects.**

When all acknowledgments to PREPARE TO COMMIT messages have been received by $p$, this fact is recorded by $p$, and the second phase begins. Producer $p$ sends every consumer a COMMIT message. Upon reception of this message, a consumer makes all changes permanent by deleting, updating, or creating data objects. Locks set on the corresponding local objects are released, and the execution context is destroyed. At any time after the beginning of the second phase, producer $p$ can tell the agent that transaction $T$ has been successfully committed.

This protocol, however, suffers some drawbacks. Because of the failure of $p$ during either the first or the second phase, objects may be kept locked for arbitrarily large time intervals. Consequently, several transactions may be rejected or made to wait. Agents should be prepared to experience arbitrarily large response times before they are told their transactions have been either committed or aborted. This situation violates the principle of mutual independence. What this principle states is that a failure of producer $i$ should impact producer $j$ ($j \neq i$) as little as possible. To meet this requirement, we have modified the original two-phase COMMIT protocol as follows. Every PREPARE TO COMMIT message also includes $C$, the list of all consumers involved in the execution of $T$, which enables any consumer, noticing that $p$ has failed, to get in touch with the other consumers belonging to $C$ to decide whether to commit or abort $T$. Ticket $t$ of $T$ is used as a common reference for such inquiries. If at least one consumer has aborted, all consumers that are up will abort since they are running the first phase of the commit. Likewise, if at least one of them has committed, all consumers that are up will commit.

If $p$ is down and all consumers involved in the execution of $T$ are up and have acknowledged the PREPARE TO COMMIT messages, then transactions $T$ will be committed. The decision will be made by the consumers, based on the knowledge that they have all agreed to commit $T$.

The only undecidable situation is when none of the consumers that are up has either committed or aborted and $p$ as well as some consumers in $C$ are down. In this case, which is not expected to occur frequently, consumers will have to wait until failed consumers come up again. Still, it would be possible to devise a solution which would further reduce the probability of such a situation occurring.

However, since it is not necessary to wait for $p$, processing/storage elements that host producers may behave as memory-less systems. Only consumers are provided with stable storage. It is therefore possible for an agent, using a producer that failed, to access another producer in order to figure out whether a transaction was committed or aborted.

**Recovering from crashes.** For recovering from crashes, a journal must be maintained for each partition. In fact, a journal is part of a partition. That is to say, a partition includes two subsets—one containing the agents' data objects, the other (the journal) containing all information necessary to survive failures which would impact the objects belonging to that partition. For every partition, the two subsets are implemented on different physical storage elements. If the agents' objects and journal are implemented on the same storage element, then the journal should be replicated. Any producer may read or update a journal by submitting actions to the pertinent consumer.

A journal contains four different kinds of information: the agents' table, images, checkpoints, and a list of descriptors.

The *agents' table* has one entry per agent that contains the value of the ticket corresponding to the most recent transaction committed for this agent.

Locally, a consumer may decide at any time to copy the current state of the partition on the journal. This is called an *image*, which has no system-wide significance.

Named markers are called *checkpoints.* A checkpoint corresponds to a state known to be consistent system-wide.

A *descriptor*—created by a consumer for every WRITE action (WRITE includes CREATE and DELETE) that has reached state $P$—contains

- ticket $t$ of the transaction invoking the action,
- the new value of the object,
- the state of the action ($P$, $C$, or $A$),
- a version ID ($V$) if the object is replicated, and
- list $H$ of all other consumers involved in the COMMIT of the transaction.

Since there is no reason why data objects on journals would be more reliable than other data objects, it may be necessary to replicate journals or parts of journals across various partitions. For every transaction, a producer should initiate the explicit actions from an agent that appear between the BEGIN and COMMIT commands, all actions needed to create or update descriptors on journals, and all actions needed to update consistently replicated data objects, which may also belong to journals.

Atomicity should be guaranteed. This is achieved by transforming every WRITE action into a logical WRITE action as follows. A specific subset of the system data in the global data structure is called the *directory,* which includes all information needed to infer to which partitions a given data object belongs. The directory may be either partially or fully replicated over partitions. As is the case for agents' and journals' objects, mutual and internal consistency must be guaranteed for objects that represent the directory. It is expected that WRITEs do not occur too frequently on directories.

As explained earlier, WRITE actions are performed on volatile copies of data objects. When the COMMIT command is decoded, every WRITE action belonging to the corresponding transaction must be finalized. The producer that acts as the COMMIT coordinator (CC) transforms every WRITE into a logical WRITE (L-WRITE) action by looking in a directory to determine which partitions contain an occurrence of the object to be written (this is list $h_1$) and which partitions contain an occurrence of the journal which is to retain the descriptor for this WRITE action (this is list $h_2$). In some cases, it is also necessary to build up a list $h_3$ that includes the names of the partitions containing a copy of the directory—such as when a WRITE is actually a CREATE or a DELETE. Let $h$ be the compound list ($h_1$, $h_2$, $h_3$). List $H$ is the concatenation of all lists $h$ built for every WRITE action for a given transaction. When $H$ has been built, the CC may initiate the COMMIT protocol. From this point, the system behaves as described previously.

**L-WRITEs are atomic, even if processing and/or storage elements crash.**

This procedure will achieve updating of all copies of the object, writing of a new descriptor, updating of the corresponding agents' table entry in all occurrences of the relevant journal, and possibly an updating of directory copies. Most important, L-WRITES are atomic, even if processing and/or storage elements crash.

Since producers may behave as memory-less processes, the removal (or insertion) of a producer that has crashed (or been repaired) does not create any particular problem in data consistency. When a consumer has to join the system, it begins by consulting its journal—any copy if several exist. Storage elements are updated by receiving a copy of the partition known to be correct but late—for example, the most recent image of that partition recorded in the journal. Then, the consumer processes all descriptors "older" than the retrieved image.

Each time it encounters a WRITE action in state $P$, it asks those consumers belonging to the corresponding list $H$ about the current state of that action in order to decide whether to commit or abort this particular action. When this is completed, the consumer sets itself to a state "up." From there on, it behaves as any other consumer.

**Unrecoverable faults and failures.** It is well known that a given recovery mechanism does not guarantee that a system will tolerate all possible failures. For instance, the system described in this article does not tolerate a situation where copies of an object are not identical and the most recent descriptor has been lost. In this case, the system "loses" transactions, leading perhaps to a domino-effect situation.

The probability that such situations will occur can be made as low as possible by choosing an appropriate number of copies for objects and descriptors. However, it is fundamental for the external agents to be aware of the possibility of a catastrophe.

To this end, each time an agent enters the system, it is given the transaction name associated with the ticket corresponding to the transaction committed most recently for this agent. In case the name retained by the system is "younger" than the name remembered by the agent, the agent knows that although the system became silent at the time the COMMIT command was issued, the corresponding transaction was effectively committed. Unnecessary duplicate execution is then avoided.

If the agent reads a name "older" than the one it holds, some unrecoverable failures have occurred since the agent left the system. The agent may then make the appropriate decision based on the knowledge of which transaction has been committed last. To completely present how the system recovers from failures, it would be necessary to explain how the distributed executive performs global checkpoints and how these checkpoints are used to install a past but consistent state when a catastrophe occurs. This subject, however, has to be left to a future article.

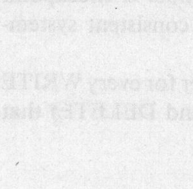
The mechanisms described in this article have been designed by Group Score* of Project Sirius. Delta is an experimental distributed transactional system built on three minicomputers, locally interconnected.[12] The initial interconnection scheme is very straightforward (point-to-point links). We decided to focus our attention on the problem of designing and implementing a distributed executive for real-time transactional systems rather than building another locally distributed communication subsystem. However, the design of the executive is totally independent of the physical topology of the communication subsystem. The interconnection of Delta's elements by some other communication medium, such as Ethernet, should not create any problem at the executive level.

The executive offers higher levels of abstraction primitives for initializing and terminating transactions with the guarantee that transactions remain atomic. Transaction content is transparent to the executive. This approach offers reasonable flexibility regarding the type of application software implemented on such distributed systems. ■

*The members of this group are Pascale Boichot, Janine Boudenant, Guy Ferran, Gérard Le Lann, Pierre Rolin, and Simone Sedillot.

## References

1. K. P. Eswaran et al., "The Notions of Consistency and Predicate Locks in a Database System," *Comm. ACM,* Vol. 19, No. 11, Nov. 1976, pp. 624-633.
2. E.D. Jensen, "The Honeywell Experimental Distributed Processor—An Overview," *Computer,* Vol. 11, No. 1, Jan. 1978, pp. 28-39.
3. P.A. Bernstein and N. Goodman, "Approaches to Concurrency Control in Distributed Database Systems," *AFIPS Conf. Proc.,* 1979 NCC, Vol. 48, pp. 813-820.
4. C.A. Ellis, "Consistency and Correctness of Duplicate Database Systems," *Proc. Sixth ACM SIGOPS,* Vol. 2, No. 5, Nov. 1977, pp. 67-84.
5. J. N. Gray, "Notes on Database Operating Systems," in *Lecture Notes in Computer Science,* R. Bayer et al., eds., Springer-Verlag, 1978, pp. 394-481.
6. B. W. Lampson and H. E. Sturgis, "Crash Recovery in a Distributed Data Storage System," Xerox Report, June 1979; to appear in *Comm. ACM.*
7. D. P. Reed, "Implementing Atomic Actions on Decentralized Data," *Proc. 8th ACM SIGOPS Symp.,* Nov. 1979, pp. 66-74.
8. D. J. Rosenkrantz et al., "System Level Concurrency Control for Distributed Database Systems," *ACM Trans. Database Systems,* Vol. 3, No. 2, June 1978, pp. 178-198.
9. G. Le Lann, "Distributed Systems—Towards a Formal Approach," *Proc. IFIP Congress,* Toronto, Aug. 1977, North-Holland Pub., pp. 155-160.
10. G. Le Lann, "Introduction a l'Analyse des Systèmes Multi-Référentiels," PhD thesis, University of Rennes, May 1977; also in report *INRIA-SIRIUS CTR. I.004,* Project Sirius, Le Chesnay, France.
11. D. A. Menasce and R. R. Muntz, "Locking and Deadlock Detection in Distributed Databases," *Third Berkeley Workshop Distributed Data Management and Computer Networks,* Aug. 1978, pp. 215-232.
12. J. Le Bihan et al., "SIRIUS-Delta: Un Prototype de Système de Gestion de Bases de Données Réparties," *Int'l Symp. Distributed Databases,* Mar. 1980, North-Holland Pub., pp. 137-159.



**Gérard Le Lann** is a member of the coordination team of Project Sirius at INRIA, France. His interests include multiple processor computers, computer networks, and distributed computer systems. Before joining Project Sirius, he was involved in the design and implementation of various multiple computer systems for the French Navy, the Centre Européen de Recherches Nucléaires, and Project Cyclades. European chairman of the First International Conference on Distributed Computing Systems in 1979 and vice-chairman of IFIP WG 10.3, Le Lann holds a Diplôme d'Ingénieur en Informatique from ENSEIHT (Toulouse) and a PhD from the University of Rennes.