North-Holland
Microprocessing and Microprogramming 14 (1984) 9–15

IST. EL. INF.
BIBLIOTECA
ARCHIVIO
APA-20

An Implementation of Optimistic Policy for Concurrency Control in a Computer Network for Real-Time Applications

M. La Manna
Selenia S.p.A., Via S. Maria 83, I-56100 Pisa, Italy
and
L. Simoncini
IEI National Council of Research, Pisa, Italy

In the field of real-time control systems, some attempts are in course to apply basic concepts of networking, data-base and transaction processing in order to provide a robust and efficient environment.

This paper presents a simple implementation of a concurrency control mechanism in a computer network for real-time applications. This mechanism insures data-base integrity and consistency and provides a better performance with respect to locking in particular situations.

The proposed mechanism is based on the detection of existing conflicts and their resolution before the beginning of the committing phase. Each transaction must verify its commit right before committing by performing a control on the statuses of shared objects. Fast transactions which update few objects are generally privileged over slow transactions which update many objects. Adding additional features to the basic mechanism, slow transactions are guaranteed to commit also in presence of concurrent fast transactions.

Keywords: transaction processing, atomic actions, intention lists, locking, optimistic policy.

1. Introduction

In the field of concurrency control design, a great set of policies has been defined to solve the basic problem of common accesses of transactions to shared data objects. This problem is generally solved detecting and handling possible conflicts among transactions. One of possible policies is the two-phase locking method, but other methods have been studied, such as optimistic policy [5]. By this policy, transactions never wait, and conflicting transactions race to the finish: given a set of conflicting transactions, the transaction that first requested commit completes, and the conflicting transactions are forced to restart. Optimistic policy is more efficient than two-phase locking with respect to the system throughput, if the effect of restarting a transaction after a conflict is negligible, while waiting for a locked object is expensive.

Other important properties of optimistic policy has been discussed with regard to the application of transaction processing to real-time systems: that is for those process control systems where system load is not constant, but some situations of congestion occur, followed by situations of relative inactivity. In correspondence of congestion, many fast transactions access few objects, so presenting a low degree of conflictuality among themselves, competing with slow transactions accessing many objects. In these cases, it is convenient to allow committing of fast transactions, while slow transactions are delayed and executed when the load of the system is small. Optimistic policy is used here not to achieve a higher throughput by the system, but to minimize the mean time for executing a transaction.

In this paper optimistic policy is reconsidered, a typical case where its application is useful is showed, then a simple implementation in a distributed environment is outlined.

2. Optimistic Policy

Concurrency control problem in a distributed data-base can be solved using two basic mechanisms: scheduling and aborting transactions. A locking based policy uses the mechanism of scheduling transactions as a function of their accessed objects.

The optimistic policy is based on aborting and automatic restarting of transactions which conflicted with concurrent transactions.

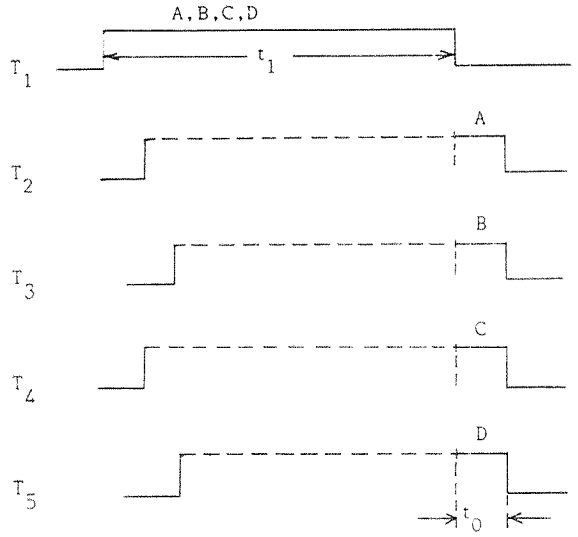

Fig. 1. Concurrent Transactions Using Locking Mechanism.

This latter policy could be very efficient in a distributed environment where most transactions are fast and have a small degree of conflictuality among themselves, while the other transactions are long and address a large set of objects. In fact, using a locking mechanism for concurrency control has some drawbacks in this case: the execution of a long transaction which locks many objects may seriously delay many short transactions attempting to access the same objects. More precisely, optimistic policy in the specificated case is more efficient than locking based policy in terms of mean time to commit (i.e., mean time between starting and committing of a transaction) in a system where situations of congestion frequently occur and are followed by temporary absence of transactions. In these cases the system is as more efficient as more transactions commit in a given time interval.

In order to model such a situation, we consider for simplicity the occurrence of a set of contemporary transactions followed by permanent system inactivity.

A typical case is showed in Fig. 1, where a long transaction T1 which operates on objects A, B, C, D locked these objects, while transactions T2, T3, T4, T5 starting immediately after T1 and attempting to access respectively objects A, B, C, D must wait committing of transaction T1. If transaction T1 lasts t1 seconds, and transactions T2, T3, T4, T5 lasts t0 seconds each, the mean time to execute transactions is approximately

$$[t_1 + 4(t_1 + t_0)]/5 = t_1 + 4/5t_0.$$

In the case of optimistic policy, the situation is showed in Fig. 2. Here transaction T1 must restart because the objects A, B, C, D have been updated

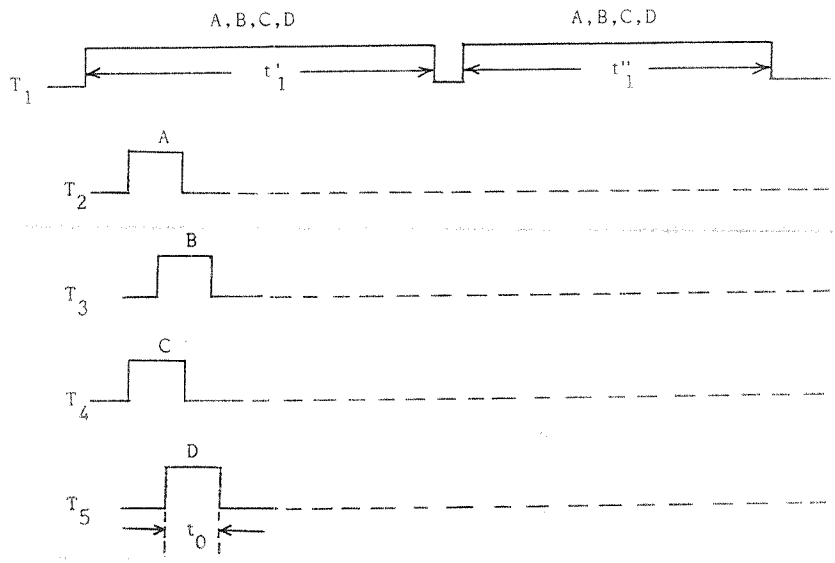

Fig. 2. Concurent Transactions Using Optimistic Policy.

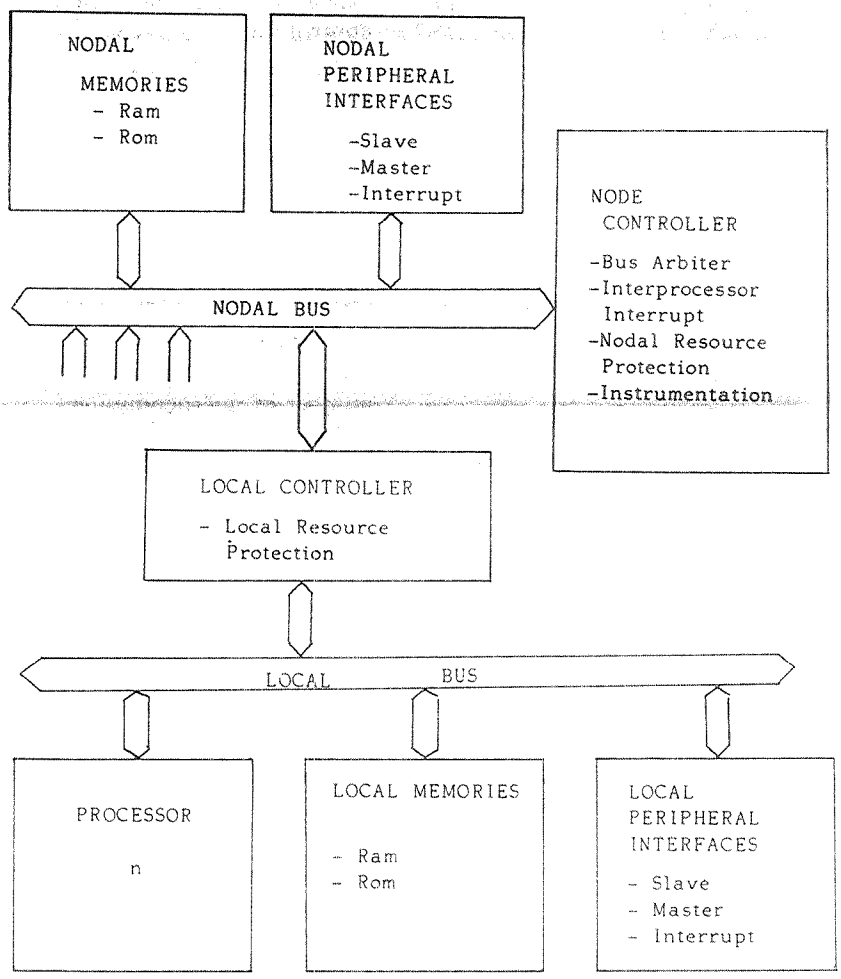

Fig. 3. Structure of a M.A.R.A. Node.

by transactions T2, T3, T4, T5. So transaction T1 lasts globally $(t_1' + t_1'')$ seconds, where $t_1' \leq t_1$ and $t_1'' \leq t_1$ while the other transactions lasts each t_0 seconds. The mean time to execute transactions is then approximately

$$(t_1' + t_1'' + 4t_0)/5 = (t_1' + t_1'')/5 + 4/5t_0,$$

which is less with respect to the previous case. The time t_1' depends on the mechanism used to restart a transaction after discovering a conflicting situation on objects accessed by the transaction itself. In the worst case, transaction T1, after the updating of objects A, B, C, D made by transactions T2, T3, T4, T5, proceeds to operate and discovers the conflicting situation only when attempting to commit: in this case t_1' would be $= t_1$, while generally $t_1' \leq t_1$.

The time t_1'' depends on the capacity of a transaction to store some useful informations about the transaction itself in its local memory, so that the time to commit after a restart is generally less than the time to commit without restart. So in the worst case after a restart $t_1'' = t_1$, while generally $t_1'' \leq t_1$.

3. Basic Mechanism Implementation

The basic mechanism is implemented on a distributed environment constituted by a local network of MARA nodes. MARA is an architecture

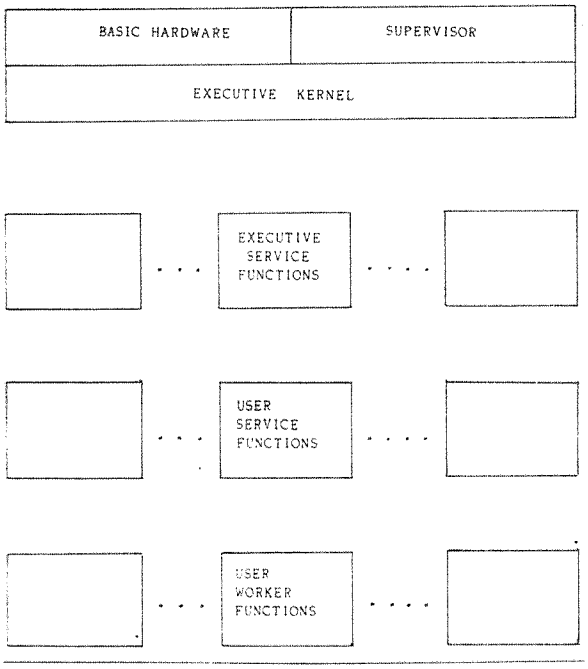

Fig. 4. Organization of M.A.R.A. Software.

developed by Selenia for applications in the process control field.

The structure of a single node is showed in the block diagram in Fig. 3. A high speed nodal bus connects up to 16 microcomputers to nodal memories and peripheral units. Each microcomputer also has a local bus on which may be placed memory and I/O resources.

MARA software is organized as a set of functions. A function represents a capacity to perform a particular type of service, and may be supported on a single microcomputer of a node or on a certain number of microcomputers.

Functions are organized into a hierarchy as indicated in Fig. 4. The functions at the top of the diagram have the highest privilege, while those at the bottom have the lowest privilege. In a typical transactional application, user worker functions perform logical and mathematical operations: they will be called "agent" functions, while user service functions control data-base and peripherals, im-

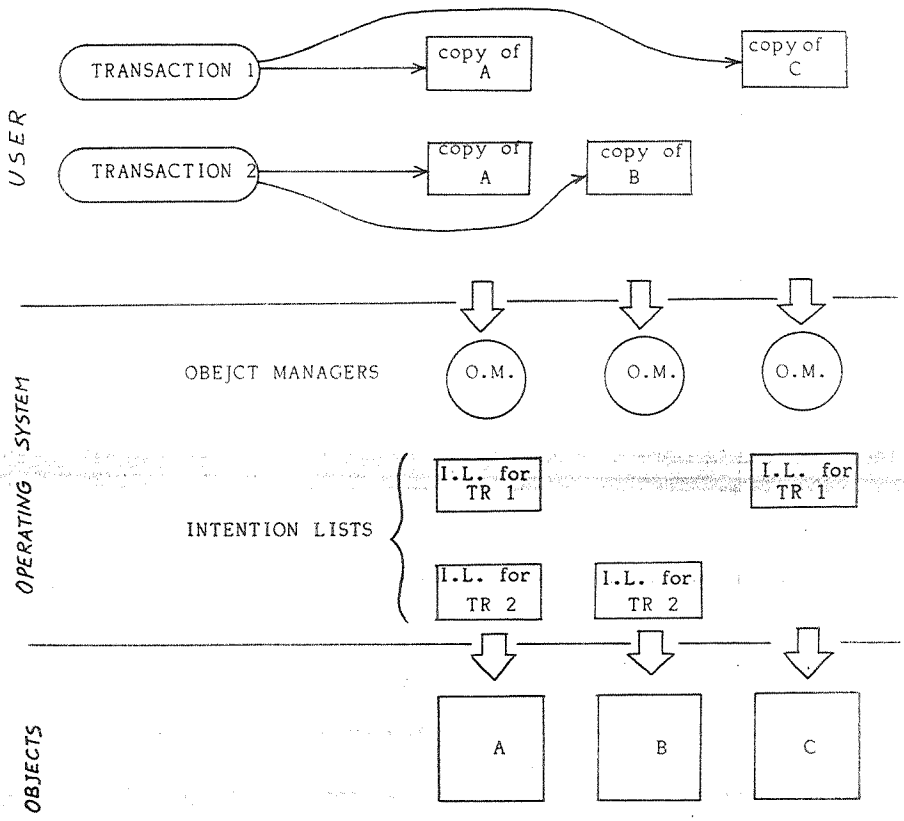

Fig. 5. Mechanism of Updating Objects.

plementing services required by agent functions: they will be called "server" functions. Functions are assigned to physical nodes at the time of system inizialization. We assume, for simplicity, that two different types of nodes are present in the network: agent nodes containing agent functions, which carry out transactions, and server nodes containing server functions. Server functions which control the accesses to objects such as files, peripherals and communication devices will be named "Object Managers".

Such an environment has been conceived for developing systems in the process control field where requirements of availability and survivability are particularly important.

We assume that user transactions manipulate objects by means of Read and Write procedures. The creation and deletion of objects are rather infrequent and are made through special requests to the operating system.

Copies of the needed objects are accessed and updated by transactions. The operating system transforms updating of those copies into intention lists for the objects (Fig. 5). At the end of each transaction, the user issues a Commit request to the operating system, in order to confirm all the actions contained in the intention lists. Transaction management is performed by system functions which are resident on server nodes, and if a transaction accesses objects on different server nodes, the first addressed object manager is also the coordinator during the Commit phase (Fig. 6). When a transaction begins, the user sends a request to the system for the assignment of a transaction identifier. This action is performed by the system in a simple manner, by allowing this assignment to be made by one of the possible servers (for example the coordinator). This assignment has to provide a unique identifier for whatever transaction in the network. Each server gives a number composed of two parts: the first one being a unique identifier of the server and the second one being the contents of a counter increased every time the server itself assign a transaction identifier.

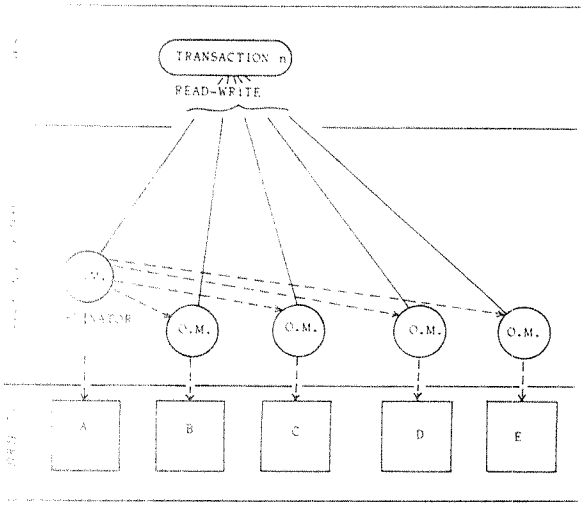

Fig. 6. Outline of Transaction Management.

When an object manager is requested to provide a service for a given transaction for the first time, it communicates to the coordinator of the transaction that it is cooperating to the transaction, through an "Add Server" procedure. By this procedure, the coordinator records the server function in the list of the servers involved in the transaction. Servers never communicate with each other, but only with agent functions and with the coordinator. The requests issued by the agent functions during the body of each transaction are registered in temporary lists, called intention lists, each associated to the proper object manager. As an example, an intention list may contain the updating done during the execution of the transaction on the data-base. This updating is explicitly confirmed at commit time. If a transaction has to update several objects, that is more then one object manager is involved in the transaction, the coordinator as a consequence of the Add Server requests maintains a list of the involved object managers.

At commit time, the coordinator scans this list and asks the object managers to confirm their intention lists. When all servers have confirmed their intention lists, the coordinator asks to erase them. When all servers have erased all intention lists, the coordinator erases its list of servers.

4. Implementation of Optimistic Policy

The implementation of optimistic policy is based on the association to each object (for example a file) of an object status (Fig. 7). When a transaction updates an object, the object status is

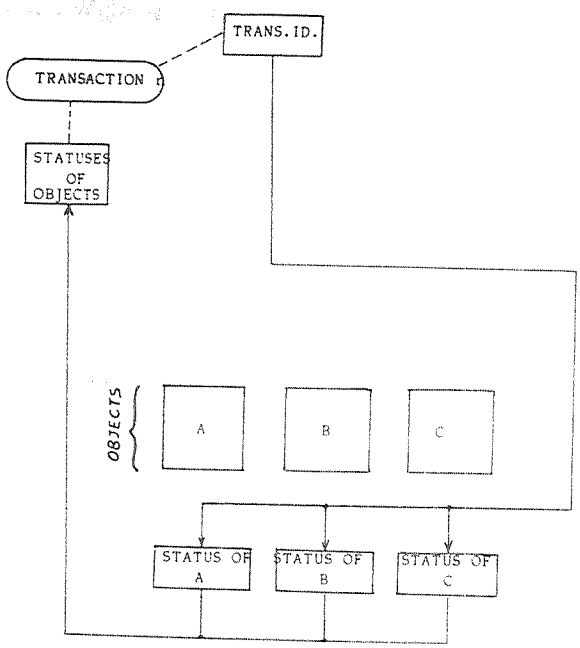

Fig. 7. Outline of the Implementation.

substituted by the identifier of the transaction updating the object. So at every time the status of object O_i is the identifier of the last transaction updating O_i. Object statuses have a default value at system inizialization.

When an agent function executing a transaction performs a Read operation on an object O_i, it reads also the object status and stores the status in its local memory until the end of the transaction. If a transaction reads an object more than once, the status is that one read on the first access to the object. Each transaction, during its execution, builds an intention list on each object it intends to update. Before committing, a transaction controls the actual statuses of shared objects. If all statuses are equal to the ones stored previously, the transaction can commit, else it must restart. To insure correctness of the algorithm, the control on the actual statuses of shared objects must be an atomic action and therefore must not be interrupted by another transaction until the first transaction has committed. In other words, when transaction Tj begins this control on a set of objects $\{O_i\}$, it must lock $\{O_i\}$, mantaining locks until the end of the commit phase. To avoid deadlock of transactions which perform concurrently this control, locking of objects must be made following a predefined serial order.

The committing phase is executed using a two-phase commit algorithm [6], with one of the object managers as a coordinator. One of the actions performed in the committing phase is to change the statuses of updated objects with the identifier of the committing transaction.

Application of optimistic policy automatically removes deadlock problem, as it makes no use of locks, except in the control of the statuses of shared objects. This phase is managed by server functions, thus the problem of deadlocks is solved by operating system. The given implementation is efficient as shown in Section 2 and solves the problem of deadlock, but the problem of starvation of a transaction indefinitely waiting for objects has to be addressed.

5. Adding Priorities to the Basic Mechanism

The problem of starvation of a long transaction accessing many objects can be solved adding to the basic mechanism a set of priorities, in order to allow commit of transactions which restarted after conflicts with other transactions.

Priorities are not assigned to transactions, but to committing of transactions actions on objects. So the system can also support a higher level of priorities assigned directly to transactions. Restarting of a transaction has not the effect of changing its whole identifier, but only a part of it, so to indicate a new version of the same transaction. After restarting, all the intention lists belonging to the old version are updated during the execution of the new version.

When a transaction restarts as a consequence of changing of a set $\{S(O_i)\}$ of object statuses during its execution, on each O_i the transaction gains one level of priority (at inizialization, obviously, all transactions have O priority on all objects).

A transaction may be restarted many times, if many fast transactions accessing the same objects occur contemporarily. When a transaction has reached the maximum level of priority on one object at least (this event should rarely occur), it must lock all its shared objects to be sure to arrive to commit.

6. Conclusions

In this paper a possible implementation of a concurrency control mechanism is presented, for applications in real-time processing. The implemented mechanism doesn't use locks and privileges transactions which first require commit, restarting all conflicting transactions. The use of the proposed mechanism should be very convenient for solving rapidly situations of congestion, where the mean time to commit of a transaction must be minimized.

References

[1] P. Ciompi, M. La Manna, C. Lissoni, I.R. Martin, L. Simoncini: "A proposal for a highly available multimicroprocessor system", 5th International Conference on Control Systems and Computer Science, Bucharest, 8–11 June 1983.

[2] P. Ciompi, M. La Manna, C. Lissoni, I.R. Martin, L. Simoncini: "A highly available multimicroprocessor system for real-time applications", SAFECOMP-83 – Cambridge 20–22 Sept. 1983.

[3] P. Ciompi, M. La Manna, C. Lissoni, L. Simoncini: "A proposal for a distributed fault-tolerant system for real-time applications", AICA-Naples, 28–30 Sept. 1983 (in Italian).

[4] P. Ciompi, M. La Manna, C. Lissoni, L. Simoncini: "A redundant distributed system supporting atomic transactions for real-time control", Real-Time Systems Symposium, Arlington, Virginia, 6–8 Dec. 1983.

[5] H.T. Kung, J.T. Robinson: "On optimistic methods for concurrency control", ACM Trans. Databases Systems 6, 2 (June 1981), pp. 213–226.

[6] B.W. Lampson, H.E. Sturgis: "Design of a Distributed File System", Xerox Palo Alto Research Center, February 1976.

[7] B.W. Lampson, H.E. Sturgis: "Crash recovery in a distributed data storage system", Xerox Palo Alto Research Center, Feb. 1976.

[8] B.W. Bernstein, N. Goodman: "Concurrency control in distributed data-base systems", Computing Surveys, Vol. 13, No. 2, June 1981, pp. 185–221.

[9] J.L. Traiger, J. Gray, C.A. Galtieri, B.G. Lindsay: "Transactions and consistency in distributed data-base systems", ACM Transactions on Data-base Systems, Vol. 7, No. 3, Sept. 1982, pp. 323-342.

[10] C.P. Chou, M.T. Liu: "A concurrency control mechanism and crash recovery for a distributed data-base system (DLDBS)", Distributed Data-bases – North-Holland Publishing Company, INRIA 1980, pp. 201–213.

[11] D.Z. Badal: "On the degree of concurrency provided by concurrency control mechanisms for distributed data-bases", Distributed Data-bases – North-Holland Publishing Company, INRIA 1980, pp. 35–46.

[12] J.M. Ayache, J.P. Courtiat, M. Diaz: "Rebus, a fault-tolerant distributed system for industrial real-time control", IEEE Transactions on Computers, Vol. C-31, No. 7, July 1982.

Mario La Manna received the degree in Electrical Engineering (magna cum laude) from the University of Pisa in 1973. In 1975 he joined Selenia, where he is currently working at the Research and Development Center of Pisa. His research interests are in the area of multiprocessor systems for real-time applications, fault-tolerance and computer networks. His current responsibilities include the implementation of a fault-tolerant system based on the MARA multimicroprocessor developed by Selenia.

Luca Simoncini received the degree in Electrical Engineering (magna cum laude) from the University of Pisa in 1970. In 1970 he joined the IEI, an Institute of National Council of Research, where he is currently working. He is also professor (with contract) at the Dept. of Computer Science of the University of Pisa. His research interests are in the area of fault-tolerance, multiprocessor systems, distributed systems architecture.