A HIERARCHICAL CONTROLLER FOR CONCURRENT ACCESSING OF DISTRIBUTED DATABASES

Mohamed G. Gouda
Information Science
Honeywell Systems and Research Center, Minneapolis, MN

ABSTRACT: An access controller for a distributed database is a (central or distributed) structure which routes access requests to the different components of the database. Such a controller is also supposed to resolve the conflicts between concurrent requests, if any, such that deadlock situations never arise.

In this paper, some architectures for distributed access controllers of distributed databases are investigated. In particular, three controllers with hierarchical architectures are considered. The controllers are evaluated based on three criteria: (i) freedom of deadlocks, (ii) robustness, and (iii) parallelism. The third criterion implies that the added redundancy to increase the controller robustness against failure conditions should also contribute to the amount of achieved parallelism during the no-failure periods. We then define a controller architecture which satisfies all the three criteria.

1. Introduction

The problem of concurrent accessing of distributed databases has received much attention in recent years (Mullery 75), (Grapa 75), (Thomas 76), (Peebles 77), and (Peebles 78). The problem seems difficult since all the already known schemes to solve it require "too much" overhead in terms of additional memory, processing, and (unfortunately) communication. Neverthless, and regardless of its difficulty, it is a fundamental problem that has to be solved if truly distributed databases are to be realized in an efficient way.

In this paper, we consider the problem for some distributed database system. The considered system is outlined in Figure 1. It has a number of user processes which access some data sets via an access controller. The user processes generate requests for some data sets in the system. It is the function of the access controller to route these requests to the appropriate data sets without causing a deadlock situation between any number of "conflicting" requests.

When a data set D receives a request from some user process U, the data set D sends an acknowledgement to U. The acknowledgement serves as a permission for U to directly access the data set D. In this case, U is said to be holding the data set D. When the user process U finishes with the data set D, it releases D which then waits for the next request. If D receives the next request while it is being held by U, it stores the request in some back up store until it is released by U.

A user process U can request, at any instant, to access any number of data sets in the system provided that U is not holding any data sets at this instant. Thus, it is a one step allocation scheme; and user processes should be designed to request data sets assuming worst case conditions. In some cases, this may prove to be a severe assumption; and further research is still needed in such cases.

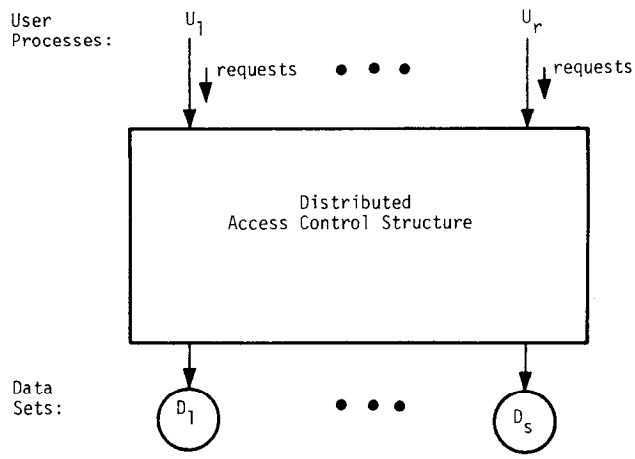

Fig. 1 The Outline of a Distributed Database System

The format of a user U request to allocate the data sets $D_{i_1},\ldots,D_{i_n}$ for an access session is shown in Figure 2. When user U sends this request.

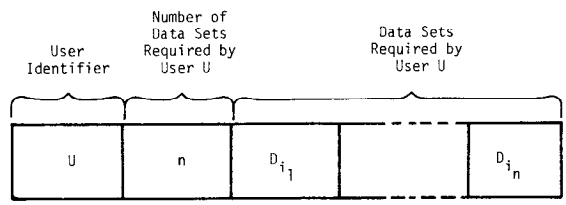

Fig. 2 The Format of a User Request

it should wait to receive acknowledgements from all the requested data sets, before it can access any of them. When U finishes with one of the held data sets, it releases the set. Only when U releases all the held data sets that it can request another collection of data sets for the next access session.

To improve the system robustiness and throughput, the access controller should be distributed (Jensen 78); i.e., it should consist of a number of autonomous communicating processes that do not share any common memory (Gouda 77a, 77b). This seems an attractive choice specially since new technologies have lowered the cost of hardware in recent years. A distributed controller should satisfy the following three requirements:

(i) No deadlock situation can arise between any set of concurrent conflicting requests (deadlock free controller).

(ii) Proper operation should continue even if some processes in the controller fail (robust controller).

(iii) Highly parallel operation is achieved (parallel controller).

In section 2, a hierarchical deadlock-free controller is specified. The specified controller is pipelined; thus its operation is reasonably parallel. However, this controller has a single failure point; i.e., it is not robust. In section 3, the controller architecture is modified to increase its robustness. The modification consists of adding extra processes to the controller without affecting its freedom of deadlocks. Unfortunately, the additional processes do not improve the system parallelism. Therefore, in section 4, the architecture is modified such that the additional processes do improve the controller parallelism during the no-failure periods.

2. A Deadlock-Free Controller

Consider the distributed access controller in Figure 3. It has four control processes C, C_1, C_2, and C_3 which form a tree structure. The user processes U_1, U_2, and U_3 are connected to the root process C. And the data sets D_1, D_2,..., and D_7 are connected to the leaf processes C_1, C_2, and C_3. Since the control processes are sequential, it is straight forward to show that any conflicting requests (i.e., requests directed to a common collection of data sets) will flow within the controller tree in sequence. Thus, deadlock situations can never arise.

The distributed database system shown in Figure 3 represents a virtual system rather than a physical system. Actually, it can be mapped into many different physical systems. For example, it can be mapped, as shown in Figure 4, into a distributed system of three processes P_1, P_2, and P_3 and two communication channels $c_{1,2}$ and $c_{2,3}$.

To explain the different interactions within the access controller, the structure and activities of a typical control process in the controller are specified. In the controller tree, each control process (except the root and the leaves) has one parent control process and some son control processes. A control process with one parent and r sons, denoted son_1, son_2,..., and son_r, is shown in Figure 5a.

Also shown in Figure 5a is the different messages exchanged between the control process and its parent and sons. The control process receives a request from its parent; and answers by sending back an ack (for an acknowledgement). The received request is partitioned into a number (less than or equal to r) of requests. These requests are then sent to different sons and acknowledgements are received as an answer.

The data structure of the control process is shown in Figure 5b. It consists of one input register, a "map" for the controller tree, and r output registers. The input register is used to store the input request after receiving it from the parent process. The controller tree map is used in the partitioning of the input request into a number of output requests, and in the assignment of each output request to a different son process. The output registers are used to store the output requests prior to sending them to the corresponding son processes.

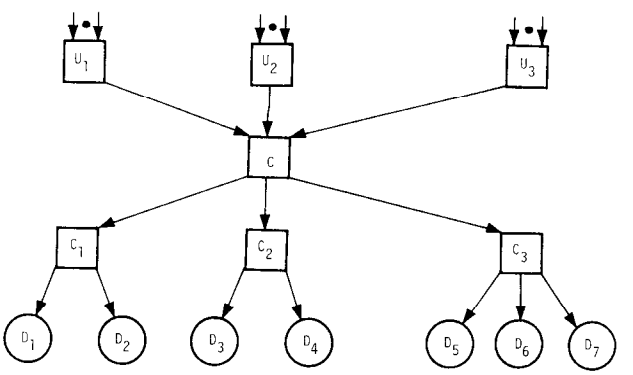

Fig. 3 A Tree-like Distributed Access Controller

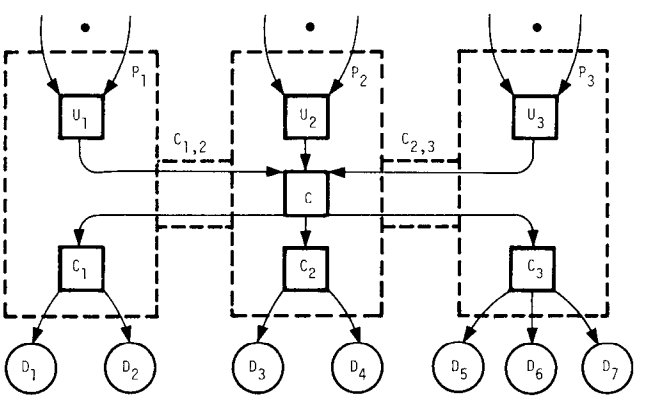

Fig. 4 Mapping the Virtual Controller to a Physical Architecture

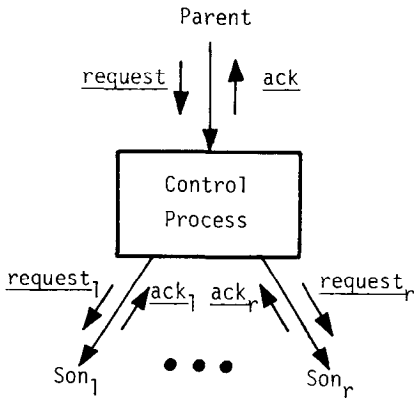

Fig. 5(a) Input/Output

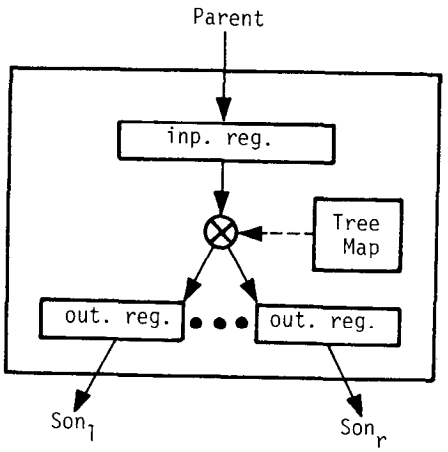

Fig. 5(b) Data Structure

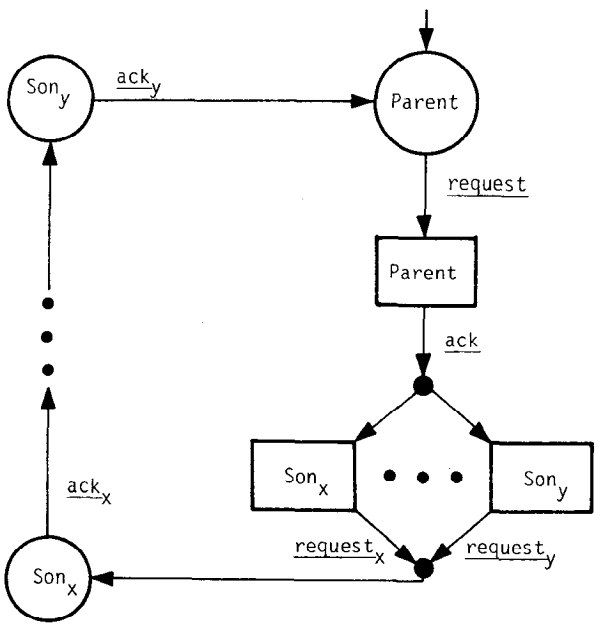

Fig. 5(c) Control Structure

The control structure of the process is shown in Figure 5c as a directed graph with two types of nodes, named receiving and sending nodes (Gouda 77a, 77b). A receiving node has a circular shape, and it represents a receiving operation in which the process waits until some message is received from another process. Similarly, a sending node has a rectangular shape; and it represents a sending operation in which the process sends some message to another process. To specify the other process and the message associated with each node, the following notation is adopted. The other process name is written as a label inside the node; and the message name is written as a label on the node output edge.

At the beginning, the control process waits to receive a request from its parent process. On receving the request, the process sends an ack to its parent; then it sends simultaneously a number of requests (denoted $request_x$,..., and $request_y$) to different son processes (denoted son_x,..., and son_y). Then, the process waits to receive ack's from these sons. On receiving the ack's, the process returns to its initial state waiting for another request from its parent; and the sequence repeats.

There are two apparent advantages to the above access controller. First, the operation is deadlock-free since conflicting requests are handled in a sequential order. This sequential order is imposed by the sequential root process which can only handle one request at a time. The following theorem is straight forward using induction on the number of levels in the controller tree.

Theorem 1:

Let r_1 and r_2 be two incoming requests to the contoller tree. If both r_1 (or any request generated from r_1), and r_2 (or any request generated from r_2) reach a control process in the controller tree, then they reach this process in the same order they reach the root process in the tree.

The second advantage of the controller is its parallel operation. The parallelism is achieved by pipelining. On the average, each level on the controller tree has one request at a time. Thus, the rate with which requests enter (or leave) the controller equals to the rate with which one request is processed in one level in the controller tree.

Despite the above advantages, the controller is not robust against process failures. For example, if the root process fails, the whole controller fails; thus, the controller has a single failure point. In the next section, we discuss how to modify the controller architecture to increase its robustness.

3. A Robust, Deadlock-Free Controller

The above controller architecture is not robust since there is only one copy of each process in it. When one process fails, all its descendant processes in the controller tree become useless since requests cannot reach any of them. Thus, one way to increase the controller robustness is by adding multiple copies of each process in the controller.

To synchronize between the different copies of one

process, the ring mechanism of Lelann (Lelann 77) can be used. The process copies are connected to a virtual ring; and there is one token circulating the ring in one direction from one process copy to the next, as shown in Figure 6a. A process copy performs its operation only when it has the ring token. After the process finishes its operation , it sends the token to the next process copy on the ring.

In this architecture, when one process copy fails, the other process copies on the ring continue to perform the same operations. The virtual ring is reconfigured to exclude the failed process copy. Some error-tolerant schemes (Lelann 77), (Chang 77), and (Kain 78) can also be implemented to handle the case when the ring token is lost due to some failures.

As duplicate copies of all control processes are added to the controller, many virtual communication lines are generated between them. As an example, Figure 6b shows the generated communication lines as duplicate copies of a process C, its parent process, and its two son processes are added to the system.

The above approach does not require that the number of added copies be the same for all control processes in the controller. This feature can be utilized by adding more copies to the processes near the tree root (where one process failure can lead to a global system failure), and less copies to the processes near the tree leaves (where process failures cause only local failures). As an example, Figure 6c shows a controller tree with three control processes C_1, C_2, and C_3, and three data sets D_1, D_2, and D_3. Assume that all data sets are of the same importance, then process C_1 is more important than C_3 which is more important than C_2. Therefore, the system is modified, as shown in Figure 6c, to have three copies of C_1, two copies of C_3, and only one copy of C_2.

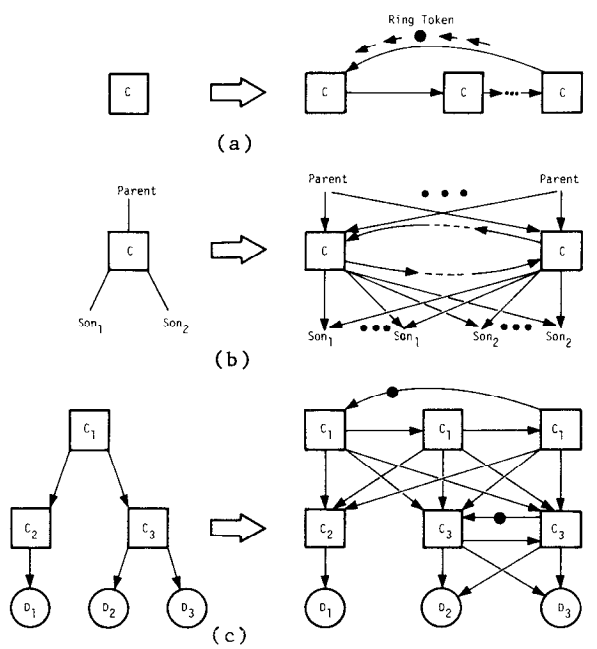

Fig. 6 Some Examples of Adding Redundancy to the Tree Controller

The specification of a control process in this controller is shown in Figure 7. The input/output specification (Figure 7a) is exactly the same as in Figure 5a except that now the process is connected to two identical process copies, called $side_1$ and $side_2$ in Figure 7a. The process receives the ring token from $side_1$ and sends it to $side_2$. The process data structure (Figure 7b) is self explanatory. The process control structure (Figure 7c) is exactly like before (Figure 5c) except of two extra operations. After receiving a request from its parent process, the process should wait to get the token of its ring. The token is released only after handling the request and sending it to the son process. This action ensures a sequential order in handling conflicting requests.

This controller is robust and its operation is deadlock-free. Freedom of deadlocks is achieved as conflicting requests are handled by different processes in the same sequential order. It is the same order with which these requests "meet" the token of the root process ring. Specifically, the following theorem can be verified using induction on the number of levels in the controller tree.

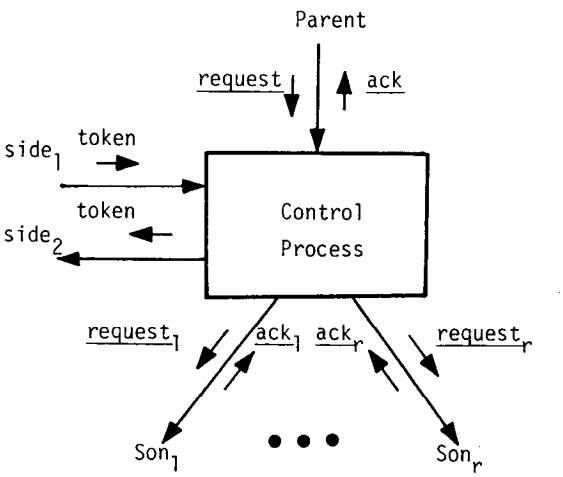

Fig. 7(a) Input/Output

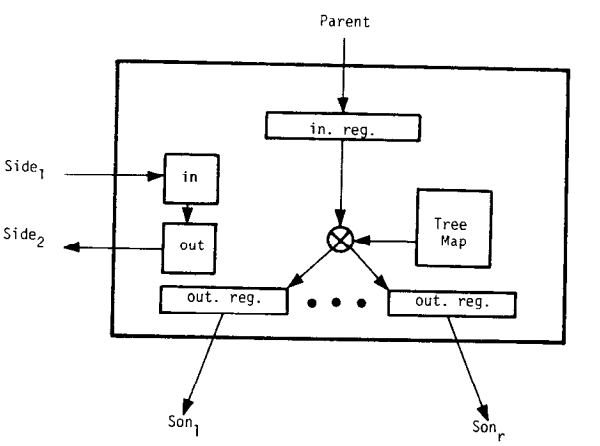

Fig. 7(b) Data Structure

Fig. 7 The Specification of the Control Process

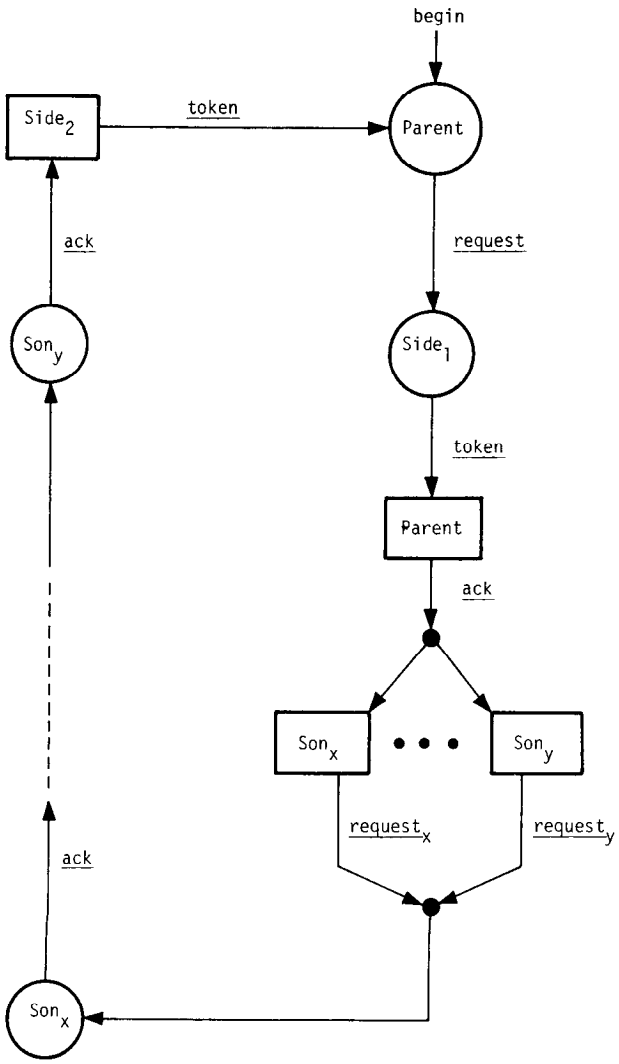

Fig. 7(c) Control Structure

Theorem 2:

Let r_1 and r_2 be two imcoming requests to the controller tree. If both r_1 (or any request generated from r_1) and r_2 (or any request generated from r_2) reach a process ring in the controller tree, then they meet the token of this process ring in the same order they meet the token of the root process ring.

The amount of parallelism in this controller equals the amount of parallelism in the controller of section 2. Thus, although the redundancy in this controller serves to increase the controller robustness against process failures, it does not improve the throughput during the no-failure periods. A mechanism is needed to utilize the controller redundancy during the no-failure conditions. Such a mechanism is discussed in the next section.

4. A Parallel Robust Deadlock-Free Controller

Figure 8 shows the path of some request r in the controller tree. The request is passed as a single piece request via the controll processes C_1, C_2, C_3, and C_4. But it is partitioned by process C_5 into two smaller requests r_1 and r_2 which are then sent to the data sets D_1 and D_2 respectively.

So long as the request remains a one piece request, it can never cause a deadlock situation with any other set of requests. Therefore, it is not necessary to order the handling of single piece requests with respect to the handling of other conflicting requests (in order to prevent deadlocks) since deadlocks cannot be created anyway. In other words, if a single piece request arrives to a control process, and if this request does not require partitioning at this process, then there is no need for the process to wait for its token ring before it sends the request to one of its son processes. In the example of Figure 8, this can be done in C_1, C_2, C_3, and C_4, but not in C_5, since C_5 does partition the request into two requests.

To add this feature, the control structure of each process should be modified to become as shown in Figure 9. When the process receives a request, it examines the request:

- Is the received request a part of some bigger request?
- Does the received request require partitioning?

Only if the answer to both questions is no is the request sent to its destination son process without waiting for the process ring token. Otherwise, the process should wait to get the token before sending the request.

By adding this feature, all the process copies on a virtual ring can be operating at the same time handling single piece requests, i.e., more parallelism. Also, if the distributed database is structured such that the majority of requests are not partitioned until they reach the bottom levels of the controller tree (see Figure 8), then a large amount of parallelism can be achieved by adding more process copies on each virtual ring. Hence, both robustness and parallelism can increase while freedom of deadlocks is preserved at the same time.

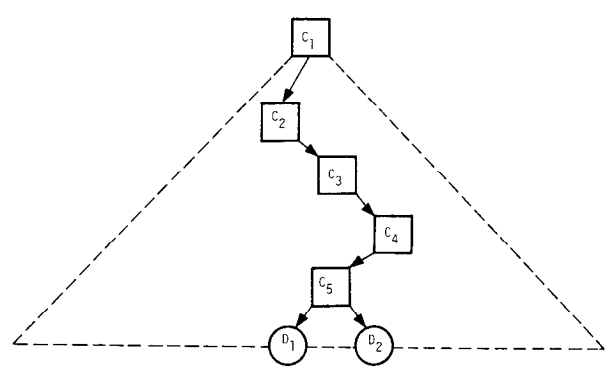

Fig. 8 A Request Path in the Controller

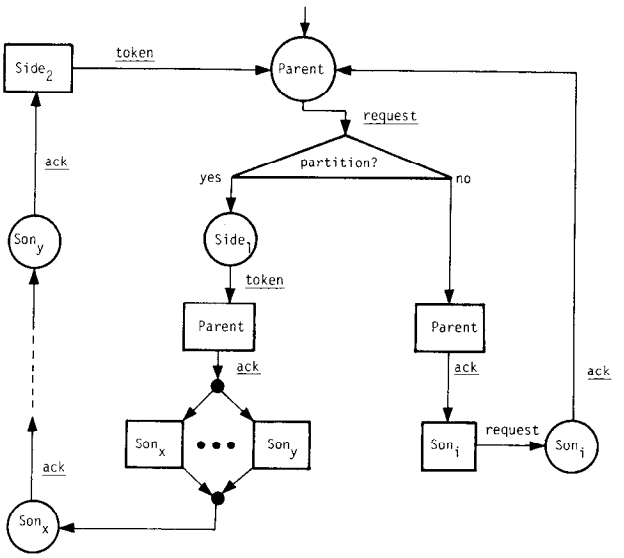

Fig. 9 The Control Structure for a Process in a Robust, Deadlock-free Controller

5. Conclusions

In this paper, three distributed architectures for access controllers of databases are investigated. The first architecture is free of deadlocks and supports parallel handling of incoming requests, however, it is very sensitive to the failure of its processes. The second architecture has evolved from the first by adding redundancy to increase the system robustness such that freedom of deadlocks and parallelism are preserved. Finally, the third architecture has evolved by modifying the second architecture so that the added redundancy can be used to increase the system parallelism during the no-failure periods.

References

(Chang 77) E. Chang, et al. Unpublished Manuscript. Computer Communications Networks Group, University of Waterloo, Ont., Canada, 1977

(Gouda 77a) M.G. Gouda. Protocol Machines: Towards a Logical Theory of Communication Protocols. Ph.D. Thesis, Computer Science Dept., University of Waterloo, Waterloo, Ont., Canada, Oct. 1977

(Gouda 77b) M.G. Gouda. Communicating Processes as a Tool for Concurrent Programming. SAI Technical Memo 2-77, Honeywell Systems and Research Center, Minneapolis, Dec. 1977

(Grapa) E. Grapa, et al. Techniques for Update Synchronization in Distributed Data Bases. Center for Advanced Computation

(Jensen 78) E.D. Jensen. The Honeywell Experimental Distributed Processor - An Overview. Computer, Vol. 11 No. 1, Jan. 1978

(Kain 78) D. Kain. Private Communication. Feb. 1978

(Lelann 77) G. Lelann. Distributed Systems - Towards a Formal Approach. Information Processing 77, G. Gilchrist edr, North - Holland Publishing Co., August 1977

(Mullery 75) A. Mullery. The Distributed Control of Multiple Copies of Data. IBM Tech. Rep. RC5782, Dec. 1975

(Peebles 77) R. Peebles. Concurrent Access Control in a Distributed Transaction Processing System. Prepared for the Brown University Workshop on Distributed Processing, Aug. 1977

(Peebles 78) R. Peebles, et al. System Architecture for Distributed Data Management. Computer, Vol. 11 No. 1, Jan. 1978

(Thomas 76) R. Thomas. A Solution to the Update Problem for Multiple Copy Data Bases Which Uses Distributed Control. Prepared for the Brown University Workshop on Distributed Processing, Aug. 1976