FAIL-SAFENESS IN A MULTIPROCESSOR SYSTEM: A DISTRIBUTED STRATEGY BASED ON BACKWARD ERROR RECOVERY

P. Corsini*, L. Lopriore**, L. Strigini***

* Istituto di Elettronica e Telecomunicazioni
Università di Pisa, Pisa, Italy

** Istituto di Elaborazione dell'Informazione
Consiglio Nazionale delle Ricerche, Pisa, Italy

***Selenia, Industrie Elettroniche Associate, S.p.A.
Roma, Italy

FAIL-SAFENESS IN A MULTIPROCESSOR SYSTEM: A DISTRIBUTED STRATEGY BASED ON BACKWARD ERROR RECOVERY

P. Corsini*, L. Lopriore**, L. Strigini***

*Istituto di Elettronica e Telecomunicazioni
Università di Pisa, Pisa, Italy

**Istituto di Elaborazione dell'Informazione
Consiglio Nazionale delle Ricerche, Pisa, Italy

***Selenia, Industrie Elettroniche Associate S.p.A., Roma, Italy

Abstract: A method for fault-handling is presented, designed for multiprocessor systems supporting concurrent processes cooperating through message exchange. The proposal is described in reference to a specific system, i.e., the MuTEAM prototype developed in Pisa: our requirements was that no erroneous output be generated by the system under a single fault hypothesis. The fault-handling model adopted is based on backward error recovery: the set of all the application processes is partitioned into disjoint subsets (called families), which represent the atomic unit of recovery. Recovery points are established on communications among families. A single consistent recovery line is maintained, thereby avoiding the domino effect. The model does not rely on the usage of mass storage devices: rather, the recovery information pertinent to all the processes is kept in the distributed main memory of the system.

1. INTRODUCTION

Fault-tolerance is becoming a qualifying requirement for computing systems in an increasing range of applications. Indeed, up to now, most fault-tolerant systems have been designed for critical environments, justifying the high cost of developing sophisticated ad-hoc solutions /1-5/. It is felt, however, that the cost of fault-tolerance may be substantially reduced if certain general fault-handling techniques are devised that can be easily tuned to the cost and performance constraints of the specific application /6/.

Distributed systems are a suitable base for implementing such techniques. The salient potential advantages are:

1) redundancy without full duplication. Due to the modularity obtainable, the adjunctive hardware for fault-tolerance may be provided on a per-module basis, according to the requirements for minimal system performance and soft degradation.
2) effective hardware support for error confinement. It is possible to enforce the separation between protection domains by mapping it onto the physical partition of hardware modules.
3) fault-tolerance through the distribution of control /7/. Distributed algorithms have a potential for higher robustness, to be ascribed to the absence of singularity points /8/. The spreading of errors due to faults may be prevented by checks being autonomously performed by interacting processes on communications with other processes /9/.

In this paper, a proposal is made for fault-tolerance, designed for multiprocessor systems supporting concurrent processes cooperating through message exchange. The approach chosen ensures that no erroneous output is generated under a single fault hypothesis. The fault-handling model adopted is based on backward error recovey /10-11/: the set of

all the application processes is partitioned into disjoint subsets (called families), which represent the atomic unit of recovery. Recovery points are established on communications among families. A single consistent recovery line is maintained, thereby avoiding the domino effect /12/.

The proposal is described in reference to a specific system, i.e., the MuTEAM prototype developed in Pisa /13/. The MuTEAM architecture /14/ features a clusterized /15-18/ multiprocessor organization. It is provided with a kernel /19/ representing the run-time support for a message-passing concurrent language based on CSP /20/. The activity of the system consists of running a set of cooperating processes, distributed among the clusters and, in a given cluster, among the computer elements constituting that cluster.

The MuTEAM hardware configuration is described in the following section. The subset of the MuTEAM kernel dealing with synchronous communications is summarized in Section 3. Then, in Section 4, the fault-handling strategies adopted are described in detail. Lastly, the specific implementation of these strategies on the MuTEAM system is reported in Section 5.

2. CONFIGURATION OF THE HARDWARE OF THE MuTEAM SYSTEM

MuTEAM is a multiprocessor system consisting of a set of clusters loosely connected via serial links. The link topology is application specific, but must satisfy the constraint that each pair of nodes is connected by at least one communication path despite any single link failure.

A standard cluster consists of up to 16 computer elements (or

nodes) connected by means of two parallel buses, namely the cluster bus and the signalling bus (Fig. 1). The cluster bus allows the processor of a node to access the shared memory blocks of the other nodes in the cluster. The signalling bus, on the other hand, is utilized for inter-node asynchronous signal transmission.

Fig. 2 shows the block diagram of the generical node, say node N. It consists mainly of: i) an Extended Central Processing Unit ECPU; ii) a Switching and Arbitration Logic SAL; iii) a Shared Memory Subsystem SMS; and iv) a Private Memory and I/O Subsystem PMS.

The Extended Central Processing Unit ECPU consists essentially of: i) a main processor MP with segmentation facilities (actually a Zilog Z8001 microprocessor); ii) an Address Translator AT; iii) a communication control unit CCU (actually a Z80-based microcomputer); and iv) hardware used for implementing both the normal I/O space and certain segments of the supervisory memory address space of the Z8001 microprocessor. As will be detailed in the following section, the main purposes of CCUs are inter-process message notification and process scheduling. The normal I/O space essentially consists of the I/O-mapped interfaces of serial controllers, utilized for intercluster communications and for exchange of information with an external development system. Finally, the supervisory memory segments contain the code and working areas for: i) bootstrap routines; ii) a local debugging monitor communicating with the external development system; and iii) routines for intercluster message routings. As well as with other clusters and with the development system (via the serial controllers), MP communicates with: i) SMS and PMS (via AT and SAL); ii) the Shared Memory Subsystems of the other nodes in the same cluster (via AT, SAL, and the cluster bus); and iii) CCU (via FIFO buffers). In its turn, CCU is connected to the other CCUs in the same cluster by the signalling bus: if communicating processes run on different nodes, the

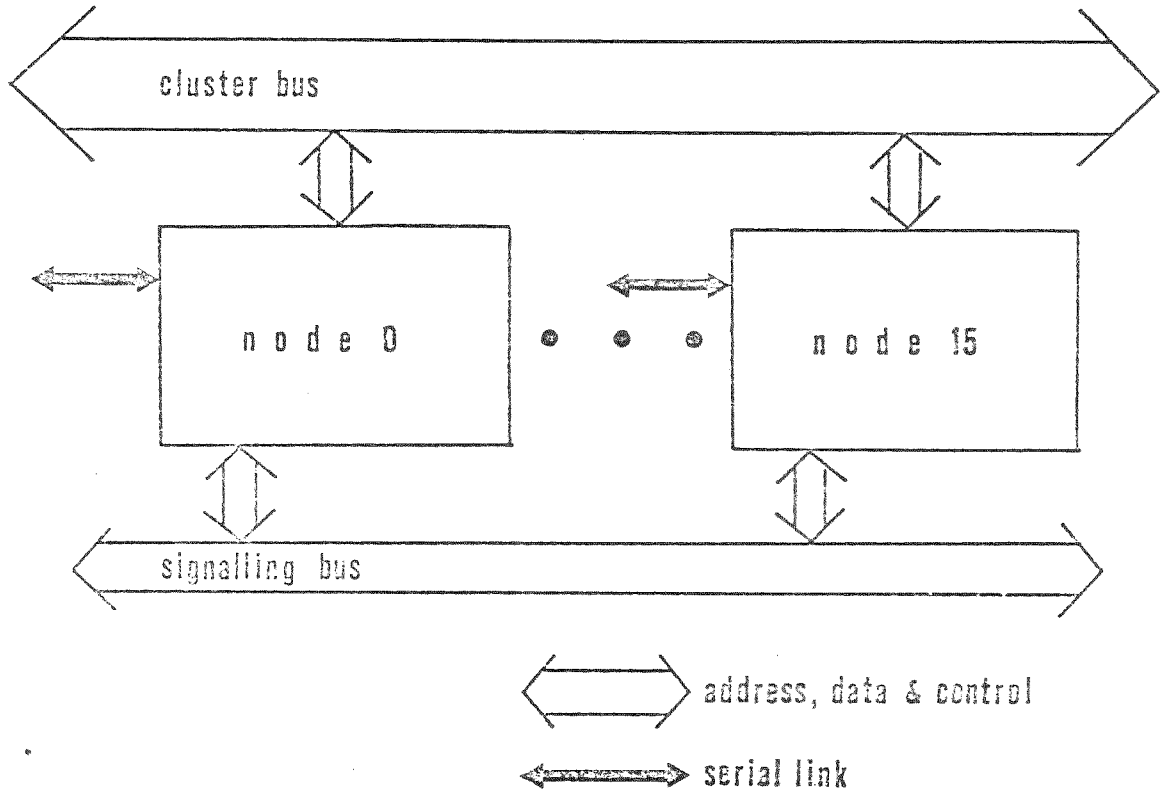

Fig. 1. Configuration of a cluster

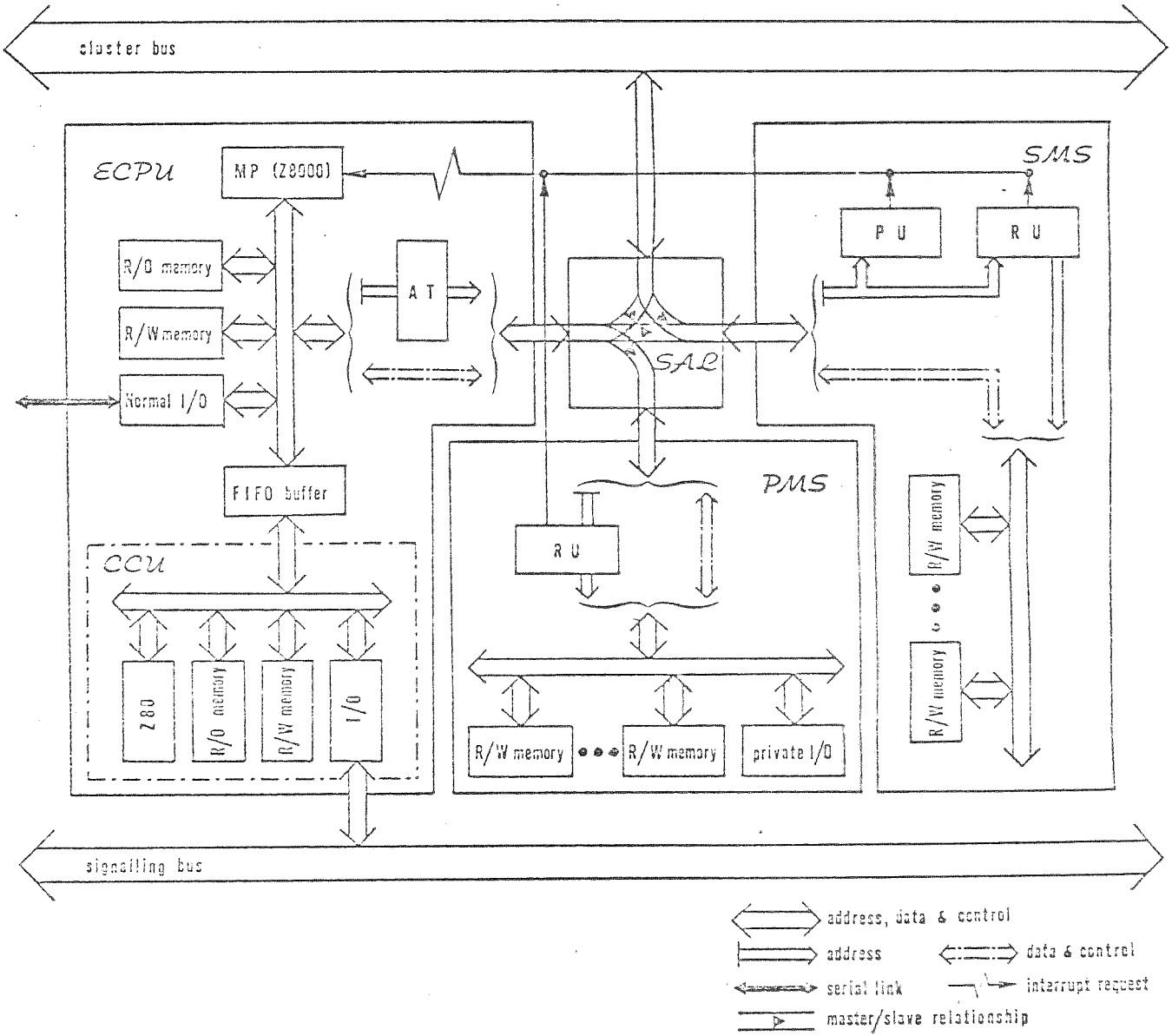

Fig. 2. Block diagram of the generical node

pertinent message notifications are performed by the CCUs of those nodes via the signalling bus.

A local space of 128 segments of up to 64K bytes each is associated to every process. When a given process P (running on MP) generates a local name S of a segment, the Address Translator AT converts the pair {P,S} into a 15-bit virtual segment name relative to a cluster global virtual space. Such a space is composed of 2^{15} segments, statically allocated to the nodes of the cluster. To be more precise, the first bit (namely Private/Shared Bit) of a given virtual segment name specifies if that segment is shared or private, i.e., it is implemented in the Shared Memory Subsystem or in the Private Memory and I/O Subsystem of a node in the cluster. Four further bits (namely Node Bits) specify the node where that segment is allocated. The ten remaining bits represent the name of the segment within the pertinent subsystem. The configuration of the node hardware and the allocation mechanism described above imply that process P can access: i) the 2^{14} shared segments implemented in the cluster of node N; and ii) the 2^{10} private segments implemented in N. On the other hand, it cannot refer to private segments implemented in a different node. Note that a complete allocation of the cluster global space requires the cluster to be actually made up of 16 nodes.

The Switching and Arbitration Logic SAL receives a virtual segment name from AT and analyzes the four Node Bits. If they specify that the segment is allocated in a different node, the logic establishes a path from the main processor MP to the cluster bus. Instead, if the segment is allocated in the same node, the logic analyzes the Private/Shared Bit, and establishes a path from MP to SMS or PMS, as required. A further task of SAL is monitoring the cluster bus for detecting segment names pertinent to SMS. When one such name is detected, a path is established by the logic from the cluster bus to

SMS. An arbitration function is also performed, both to assure a single mastership of the cluster bus and to maintain the required mutual exclusion between contemporary access attempts to SMS coming from the cluster bus and from MP. The organization described features a high degree of parallelism in memory accesses. Indeed, a process running on N: i) accesses segments in PMS without interfering with processes running on other nodes, and accessing to segments in SMS; and ii) accesses segments in SMS without interfering at the cluster bus level with any other process working on segments in a different memory subsystem.

The Shared Memory Subsystem SMS is made up of: i) a physical memory bank of up to 1 Mbyte; ii) a Relocation Unit RU; and iii) a Protection Unit PU. The Relocation Unit RU receives the 10-bit virtual name of a segment allocated in SMS and a 16-bit offset, and translates such a pair into the 20-bit address of the pertinent memory location in the physical memory bank. The unit mainly consists of 2^{10} registers, one register for each segment of SMS. A register is 32 bits wide, and is partitioned into two 16-bit fields, namely the Base and the Limit fields: the quantity contained in the Base field of the register addressed by the virtual segment name multiplied by 16 represents the segment base in the bank. The offset is added to the base to obtain the required physical memory address. An ad hoc logic verifies that the offset does not exceed the segment length, as specified by the Limit field of the register above.

The Protection Unit PU enforces an access control list technique /21/ for protecting the segments of SMS. The possible access rights are READ and WRITE, and the active entities, whose accesses to the segments must be validated, are the main processors of the nodes of the cluster (this implies that the virtual address transmitted on the cluster bus must be paired with the 4-bit name of the processor generating that

address). The unit mainly consists of 2^{10} registers, one register for each segment allocated in SMS. Each register is 32 bits long, and implements the access list for the pertinent segment. To be more precise, the {k-th, (k+16)-th} bits specify the access rights held on that segment by MP_k, k=0,1,...,15. A dedicated status line of the cluster bus specifies the type of the access attempt on the segment actually addressed. A proper access violation checker compares the access type with the rights of the accessing processor on the specific segment. As will be shown in Section 5, the protection mechanisms described are utilized for logically insulating the faulty node from the other nodes in the cluster after a fault has been detected.

Note that all registers contained in both the Relocation and the Protection Units of SMS are accessible only to the Main Processor of the same node, and only if it is running in the supervisor state. This guarantees both protection from erroneous user processes and confinement of errors due to faults in different nodes.

The protection environment is completed at the hardware level by a feature of SAL which considers requests for the cluster bus from MP only if MP is running in the supervisor state. This implies that communication among processes allocated on different nodes can only be obtained by calls of proper kernel routines.

The structure and internal organization of the Private Memory and I/O Subsystem PMS is similar to that of SMS. However, no protection logic is provided, that is, both the access rights READ and WRITE on all private segments implemented in PMS are permanently assigned to MP. Certain segments of PMS may be used to contain the memory-mapped interfaces of any possible input/output device private to node N.

3. LOW LEVEL MECHANISMS FOR MESSAGE EXCHANGE

Two sophisticated procedures are provided at the kernel level, namely SECURE_SEND and SECURE_RECEIVE, which are utilized by processes for interprocess message transmission. These procedures implement secure communications featuring fault-tolerant characteristics (such as message validation and fail-safeness) and the establishment of recovery points. They make wide use of two lower level procedures, namely SEND and RECEIVE, performing synchronous message passing with no provision for fault-handling. In turn, these two procedures rely on a set of commands, implemented by CCUs, and mainly utilized for synchronization among processes. In this section, these commands and a simplyfied version of the SEND and RECEIVE procedures will be briefly described, whereas the SECURE_SEND and SECURE_RECEIVE procedures will be described only in Section 5. From now on, we will refer to a process S allocated on node N_i wishing to send a message of type T to a process R allocated on node N_j.

3.1. CCU Commands

Each CCU interfaces to the Main Processor of its own node by providing a specialized set of commands. The most important of these, namely CCU_SUSPEND_SENDER, CCU_SUSPEND_RECEIVER, CCU_AWAKE_SENDER and CCU_AWAKE_RECEIVER, will now be described.

The CCU_SUSPEND_SENDER command is as follows:

command CCU_SUSPEND_SENDER(R:in PROCESS_NAME; T:in MESSAGE_TYPE);

An actual invocation of this command performed by process S to CCU_i causes S to be suspended and a new process to be dispatched to the main processor MP_i of N_i. Process S is registered as waiting for the event:

R receives a message of type T from process S.

The CCU_SUSPEND_RECEIVER command is as follows:

command CCU_SUSPEND_RECEIVER(S:in PROCESS_NAME; T:in MESSAGE_TYPE);

An actual invocation of this command performed by process R to CCU_j causes R to be suspended and a new process to be dispatched to the main processor MP_j of N_j. Process R is registered as waiting for a message of type T from process S.

The CCU_AWAKE_SENDER command is as follows:

command CCU_AWAKE_SENDER(S:in PROCESS_NAME; T:in MESSAGE_TYPE);

An actual invocation of this command performed by process R to CCU_j causes CCU_j to signal to CCU_i of node N_i (where S is allocated) that a message of type T coming from S has been received by R (consequently, CCU_i will schedule S as ready again).

Finally, the CCU_AWAKE_RECEIVER command is as follows:

command CCU_AWAKE_RECEIVER(R:in PROCESS_NAME; T:in MESSAGE_TYPE);

An actual invocation of this command performed by process S to CCU_i causes CCU_i to signal to the CCU_j of node N_j (where R is allocated) that a message of type T coming from S is available for R (consequently, CCU_j will schedule R as ready again).

3.2. Data Base for Low Level Message Exchange

Let us now describe the data base provided in the kernel for managing low level message exchanges among processes. It consists essentially of a set of tables, called INCTABLEs, each one relevant to a specific process. $INCTABLE_R$ pertinent to process R consists of a set of entries. Each entry is associated to a pair {S,T} , where S is the name of the process which is allowed to utilize that entry for sending

messages of type T to R. To be more precise, the generical entry consists of: i) the specification of the process name S and the message type T to which the entry is associated; ii) a flag W, specifying if R is waiting for a message of type T from S; and iii) a pointer BF to (i. e., the virtual segment name of) a buffer for the message. In turn, a buffer consists of: i) a Lock flag L to prevent simultaneous access to the buffer by R and S; ii) a Full flag F specifying if the buffer actually contains a message; and iii) a memory space M wide enough to contain the body of a message of type T. The declarations for these objects in a self-explanatory notation are as follows:

```
type BUFFER is
  record
    L,F: BOOLEAN;
    M: MESSAGE_BODY;
  end record;

type INCTABLE_ENTRY is
  record
    S: PROCESS_NAME;
    T: MESSAGE_TYPE;
    W: BOOLEAN;
    BF: access BUFFER;
  end record;

type INCTABLE is
  array(NATURAL range <>) of INCTABLE_ENTRY;
```

$INCTABLE_R$ is stored in a shared memory segment of the node N_j to which R is allocated. The access right READ on such a segment is granted to all processors in the cluster. The access right WRITE is only granted to the main processor MP_j in node N_j. Buffers are stored in separate segments of N_j. Both READ and WRITE access rights on each one of these segments are granted to MP_j and to main processor MP_i of the node to which process S is allocated.

A directory, called the INCTABLE Directory, is stored in the

private memory of each node. It has one entry for each process: the k-th one contains a pointer to the INCTABLE of the k-th process. A declaration for these directories is as follows:

type INCTABLE_DIRECTORY is
array(1..PROCESS_NUMBER) of access INCTABLE;

where PROCESS_NUMBER is the actual number of processes.

3.3. The SEND and RECEIVE Procedures

The SEND and RECEIVE procedures allow a sender process S and a receiver process R to perform synchronous message exchange. Each message is a pair message type, message body , where the message body is always an array of bytes, and the message type qualifies the contents of the array.

A heading for the SEND procedure is as follows:

```
procedure SEND(S,R: in PROCESS_NAME; T: in MESSAGE_TYPE;
               B: in MESSAGE_BODY);
```

Execution of the procedure causes the buffer pertinent to process S and message type T (as specified by $INCTABLE_R$) to be filled with the message body B. Process S is suspended at the occurrence of either of the two following events: i) the buffer is full (that is, it contains a message previously inserted by S and not yet completely absorbed by R); and ii) process R is not waiting for a message of type T from S.

Let us now consider the Receive procedure. A heading is as follows:

```
procedure RECEIVE(R,S: in PROCESS_NAME; T: in MESSAGE_TYPE;
                  B: out MESSAGE_BODY);
```

Execution of the procedure causes the contents of the buffer pertinent to the commmunication to be copied into the variable B local to R. Process R is suspended only on the occurrence of a condition of buffer empty.

It should be clear that process blockings and consequent awakenings involved in both the procedures mentioned above are obtained by utilizing the CCU commands previously described.

4. RECONFIGURATION AND RECOVERY

Our requirement for system fault tolerance was that the system must tolerate any single fault.

By single fault we mean: i) a node fault, i.e., one or more faults localized in the same node; or ii) any set of simultaneous node faults on nodes each of which belongs to a different cluster; or iii) a link fault i.e., any set of simultaneous faults on intercluster serial links that do not split the system into non-connected subsystems. In this paper, we only deal with single faults of the first two kinds: link failures are dealt with by means of strategies based on successive message routings among clusters through non-faulty links /22-24/.

By tolerate we mean that outputs generated by the system towards its environment are guaranteed to be correct. We model the system environment by means of a set of specialized processes (namely actuator processes) dedicated to driving external actuators. We do not hypothesize any ability of these specialized processes to deal with lost or repeated messages. Indeed, by correct output we mean that if a message should be generated towards an actuator process, then it is actually generated, and only once.

The chosen strategy is based on: i) validating system outputs by

means of autotest procedures /25-26/; ii) reconfiguring the system, in order to cut out the node which has failed; and iii) the backward recovery /23-24/ of the set of processes affected by the fault. A single consistent recovery line is maintained, thereby avoiding the domino effect /25/. The strategy has to work in a system which is not provided with any mass storage device. This implies that the number of past states maintained for each process must be kept minimized to one. However, in order to ensure the availability of at least one safe past state for each process after any fault, a back-up copy and an auxiliary back-up copy of the state must be maintained, stored in two different nodes. Moreover, efficiency requires that the frequency of updating these copies must be reduced as far as possible, while, in order to avoid too much degradation in the performance of the system, the back-up copies should be updated for a few processes at a time.

The problem of partial update is twofold. At any given time: i) each copy must contain correct state information for the pertinent process; and ii) the entire contents of all the copies must be consistent, i.e., they must represent a state of the whole system admissible when no faults are present, and congruent with the past interaction between the system and its environment. The two requirements stated above represent a central issue in fault-tolerance for distributed systems.

4.1. Process Families

The goals mentioned previously are achieved by partitioning the set of all the processes into disjoint subsets, called families. The processes in the same family cooperate closely to achieve a common functionality. They communicate with each other frequently, and, much less frequently, with the processes of the other families, in a

controlled fashion. To be more precise, interfamily interactions are performed sequentially, so that a family can be assimilated from its outside to a single powerful process. Furthermore, if a data flow occurs across the boundary of a family, the information involved is certified as correct: this is obtained by validating such information through the execution of the autotest procedures.

This suggests a simple criterion satisfying the two requirements for partial update: corresponding to each interfamily communication, a new back-up copy is put in the place of the old one, for each process in the two families involved. This copy is: i) correct, as it is validated by autodiagnosis; and ii) congruent with the copies of the other processes in the system, since rolling back all the processes of the two families involved to this back-up state, would not undo any interfamily communication.

The actual composition of each family, i.e., the names of the processes belonging to that family, is stated by the applicative programmer, and represents a further input for the compiler. In order to guarantee the correctness of the messages to the actuator processes, the constraint should be followed whereby each of them is a family on its own. Moreover, the resulting frequency of interfamily communication should be the outcome of a tradeoff between two opposite requirements: i) the system must not be overburdened with too frequent activities of back-up state updating; and ii) the recovery line available in case of failure must not be too remote from the present state of the system.

4.2. Family arbiters

In order to guarantee the required serialization of interfamily communications performed by processes belonging to the same given family, each family is provided with an arbiter process. Before starting

an interfamily communication, each process must obtain a grant from the arbiter of its own family. This grant is returned only after the arbiter has communicated properly with the other arbiters, so as to avoid the enforced serialization of communications becoming a source of deadlock /27/.

The arbiters are also responsible for starting a diagnostic and fault-handling (DFH) session corresponding to each interfamily communication. Indeed, a request from one single arbiter is sufficient to trigger off a DFH session: proper software mechanisms embedded in DFH algorithms guarantee that multiple requests for system diagnosis issued during a DFH session do not cause useless repetition of the session. In order to ensure that at least one arbiter starts a DFH session at each given interfamily communication even when a faulty node is present, arbiters of different communicating families are allocated in different nodes. Moreover, a further restriction is that any two communicating processes belonging to different families must be allocated in different nodes. In such a way, we are assured that at least one of the two processes makes a request for interfamily communication to its arbiter.

4.3. Cluster Reconfiguration

After a fault has occurred and has been diagnosed in node N_f, proper reconfiguration activities are carried out reallocating all the processes previously allocated in N_f to non-faulty nodes. In actual fact, in order to avoid possible inefficiencies in the resulting workload distribution among nodes, we provide a mechanism which allows reallocation of the processes allocated in every node. To be more precise, according to our model for faults, in each cluster as many different faults may occur as there are nodes in that cluster: for each

fault, we store proper information about where to reallocate every process in the cluster. However, efficiency considerations for reconfiguration suggest that, in order to generate the lowest information flow: i) a process allocated on a non-faulty node should be reallocated on the node containing its back-up copy, or on its own node (storing the auxiliary back-up copy of the process); and ii) a process allocated on the faulty node should be reallocated on the node containing its back-up copy. These criteria should also be kept in mind by the applicative programmer when distributing back-up copies and auxiliary back-up copies among nodes.

5. IMPLEMENTATIVE ISSUES

Let us consider a process P allocated on node N_u. We designate: i) by CD_P and WA_P the two sets of segments containing the code and the working area of P, respectively (these segments are stored in the private memory of N_u); and ii) by $INCT_P$ and BUF_P the two sets of segments containing $INCTABLE_P$ and message buffers for P (these segments are stored in the shared memory of N_u). In order to allow recovery after a failure, it is sufficient to maintain a single back-up copy of the process code: the pertinent set of segments is called BCD_P, and it is stored in a node N_v different from N_u. On N_v we also maintain the back-up copy of the past state of process P, that is, of WA_P, $INCT_P$ and BUF_P: the pertinent sets of segments are called BWA_P, $BINCT_P$ and $BBUF_P$, respectively. Moreover, let AWA_P, $AINCT_P$ and $ABUF_P$ be the set of segments storing the auxiliary back-up copies of the state of P: in order to minimize the utilization of the cluster bus during their updating, these copies are maintained in the private memory of node N_u.

5.1. Data-Base for Fault-Handling

All the configuration information pertinent to process P is contained in a table, called Process Fault Handling Table $PFHT_P$. Besides the name of the process and of the node where the process is allocated, the table contains an array of node names, the f-th one specifying where P must be relocated after a failure has occurred in node N_f. Moreover, the table contains pointers to the process code area, working area, INCTABLE area and buffer area, and to all the back-up and auxiliary back-up copies of these areas, as described above: in order to reduce the size of the table, we make it obligatory for each area to be stored in a set of consecutive virtual memory segments, so that it is completely addressed by the name of the first segment, together with the number of the segments constituting that area. A structure for $PFHT_P$ is given by the following declaration:

```
type PROCESS_FAULT_HANDLING_TABLE is
  record
    P_NAME: PROCESS_NAME;
    N_NAME: NODE_NAME;
    NEW_ALLOCATION: array(1..NODE_NUMBER) of NODE_NAME;
    CD, BCD: access CODE_SEGMENT;
    CD_LENGTH: NATURAL range 1..MAX_CODE_AREA_LENGTH;
    WA, BWA, AWA: access WORKING_AREA_SEGMENT;
    WA_LENGTH: NATURAL range 1..MAX_WORKING_AREA_LENGTH;
    INCT, BINCT,AINCT: access INCTABLE_SEGMENT;
    BUF, BBUF, ABUF: access BUFFER_SEGMENT;
    BUF_LENGTH: NATURAL range 1..MAX_BUFFER_AREA_LENGTH;
  end record;
```

$PFHT_P$ is stored in the private memory of node N_u. A back-up copy of $PFHT_P$, called $BPFHT_P$, is stored in the private memory of the same node N_v where the back-up copy of the process state is stored. A directory in each node, called the Fault-Handling Directory, contains pointers to all PFHTs and BPFHTs stored in that node. A definition for these directories is as follows:

```
type FAULT_HANDLING_DIRECTORY is
   array(NATURAL range <>) of access PROCESS_FAULT_HANDLING_TABLE;
```

where the range is stated according to the actual number of PFHTs and BPFHTs stored in the specific node.

The data base for fault-handling is completed at the node level by two tables, called Family Configuration Table FCT and Arbiter Table ART, both stored in the private memory of the node. The first has one entry for each process: the contents of the r-th entry specifies the family the r-th process belongs to. A null family name specifies that the pertinent process must be considered as belonging to every family: as will be shown later, the only processes characterized in this way are family arbiters and diagnostics and fault-handling processes (these processes, whose concurrent execution produces a DFH session, will be described later in this section). The Arbiter Table has one entry for each family: the contents of the s-th entry specify the name of the arbiter associated to the s-th family. Declarations for these tables are as follows:

```
type FAMILY_CONFIGURATION_TABLE is
   array(1..PROCESS_NUMBER) of FAMILY_NAME;

type ARBITER_TABLE is
   array(1..FAMILY_NUMBER) of ARBITER_NAME;
```

5.2. Secure Message Passing

As stated in Section 3, a process wishing to send or receive a message should not utilize the SEND or RECEIVE procedures. Instead, it should make use of two higher level procedures, namely SECURE_SEND and SECURE_RECEIVE, whose aim is to enforce the discipline on message passing described in the previous section. Indeed, these procedures in

turn actually call the SEND and RECEIVE procedures for obtaining both the effective transfer of information and the required serialization and validation of interfamily communications.

Let us now refer to the SECURE_SEND procedure. A program is as follows:

```
procedure SECURE_SEND(S,R: in PROCESS_NAME; T: in MESSAGE_TYPE;
                      B: in MESSAGE_BODY) is
   ARS: PROCESS_NAME;
begin
   if FCT(S)=FCT(R)
   then SEND_MESSAGE(S,R,T,B);
   else ARS:=AT(FCT(S));
        SEND(S,ARS,transmit_request,R);
        RECEIVE(S,ARS,transmit_grant,G);
        SEND(S,R,T,B);
        SEND(S,ARS,exit_request,G);
        RECEIVE(S,ARS,exit_grant,G);
   end if;
end;
```

Consider an actual invocation of the procedure, performed by a process S (allocated on node N_i) wishing to send a message of a specified type T and body B to a receiver process R (allocated on node N_j). The procedure execution firstly causes proper accesses in the Family Configuration Table of node N_i for obtaining the names F_S and F_R of the families of S and R, respectively. If $F_S=F_R$ (that is, the communication is an intrafamily one), message transmission is simply obtained by a proper call of the SEND procedure. Instead, if the communication is an interfamily one, the Arbiter Table is firstly accessed for obtaining the name AR_S of the arbiter of family F_S. A message is sent to AR_S, requesting to enter the critical section constituted by the interfamily communication, and specifying the name of the intended receiver process. A message containing the grant is then waited for: the grant is accorded by the arbiter only after proper syncronization activities have taken place, involving arbiter AR_R of family F_R and, possibly, arbiters of other families communicating with F_S. When the grant has

been obtained, the actual transfer of the message to process R is accomplished by utilizing the SEND procedure. Afterwards, a message is sent to arbiter AR_S, asking permission to exit the critical section. A message containing permission is then waited for: permission is accorded by the arbiter only after a diagnostic and fault-handling session has taken place, validating message transmission and either updating back-up states (as required) or possibly causing a backward error recovery. It should be noted that a call of the SECURE_SEND or SECURE_RECEIVE procedure in communications between arbiters simply turns into a call of the SEND or RECEIVE procedure, respectively: this happens because each arbiter belongs to all families.

Let us now refer to the SECURE_RECEIVE procedure. A program is as follows:

```
procedure SECURE_RECEIVE(R,S: in PROCESS_NAME; T: in MESSAGE_TYPE;
                         B: out MESSAGE_BODY) is
  ARr: PROCESS_NAME;
begin
  if FCT(R)=FCT(S)
  then RECEIVE(R,S,T,B);
  else ARr:=ART(FCT(R));
       SEND(R,ARr,receive_request,S);
       RECEIVE(R,ARr,receive_grant,<>);
       RECEIVE(R,S,T,B);
       SEND(R,ARr,exit_request,<>);
       RECEIVE(R,ARr,exit_grant,<>);
  end if;
end;
```

As in the case of SECURE_SEND, consider an actual invocation of the procedure, performed by a process R wishing to receive a message of a specified type T from a process S into a local variable B. The procedure execution firstly checks if R and S belong to the same family. If so, message receiving is simply accomplished by a proper call of the RECEIVE procedure. Otherwise, the same call is a critical section. So it must be preceded and followed by proper message exchanges with arbiter AR_R of the family F_R. These message exchanges

are similar to those described above for the SEND procedure: their aim is to request and obtain permission to enter and leave the critical section, respectively (the behaviour of arbiter AR_R is similar to that of AR_S: it allows the critical section to be entered only after synchronizing with AR_S and, possibly, with other family abiters; moreover, it allows the critical section to be abandoned only after triggering off a DFH session, and waiting for session termination).

5,3. Fault Handling

Let us now analyze a diagnostic and fault-handling session in more detail. It is supported by a set of highest priority processes, called Diagnostic and Fault-Handling Processes (DFHPs), one for each node. DFHPs may communicate with each other; moreover, each of them may communicate with all the family arbiters in its own node. In all cases, as DFHPs and arbiters are considered to belong to every family, these communications are carried out as intrafamily ones. No back-up information about the states of DFHPs is maintained; indeed, after a fault, these processes are neither reallocated nor rolled back.

As long as no DFH session is being carried on, all DFHPs are suspended, waiting for a message from any of the respective potential partners in a communication. When a DFHP receives a request from an arbiter for a DFH session, it extends the request to all other DFHPs: this triggers off the session, which always begins with system diagnosis. During this phase, DFHPs execute a parallel algorithm characterized by the consistency of results obtained by each DFHP running on a non-faulty node, and by syndrome decoding being local to each DFHP. This algorithm, detailed in /26/, guarantees a satisfactory

degree of fault coverage at low cost, if properly supported by specialized test and error detection hardware.

Let us now consider a no-fault situation. In this case, upon termination of the diagnostic activity, each DFHP autonomously enters a back-up phase: concurrent execution of this phase by all DFHPs produces updating of the back-up and auxiliary back-up copies of all processes belonging to the families involved in the interfamily communication which caused the DFH session. Let us refer to $DFHP_m$ allocated on node N_m: firstly, it looks up the Family Configuration Table FTC_m stored in N_m for obtaining the names of those processes which are both allocated in N_m and belong to the families above. Then, $DFHP_m$ updates the back-up and auxiliary back-up copies of each of these processes by exchanging proper messages with the DFHPs in the node storing the copies. Lastly, $DFHP_m$ concludes its own elaboration by sending an acknowledgment message to the arbiters possibly allocated in N_m and associated to the families involved in interfamily communication.

Instead, let us consider the case in which a fault exists in the system, localized in node N_f. In this case, each DFHP terminates the diagnostic phase after having identified and localized the fault. In this way it enters a fault-handling phase: concurrent execution of this phase by all DFHPs on non-faulty nodes produces global fault-handling activity consisting of three steps. In the first step, node N_f is logically disconnected from non-faulty nodes. In the second step, processes are redistributed among these nodes, and backward recovered. Lastly, in the third step, consistency of the data base for message exchange is restored.

We now refer again to $DFHP_m$ allocated on node N_m and shall describe its activities pertinent to the three steps above. In the first step, it deletes from the registers of the Protection Unit PU_m of N_m all access rights owned by N_f on shared memory segments implemented

in N_m. Moreover, it notifies CCU_m that any communication coming from CCU_f must be ignored.

In the second step, $DFHP_m$ carries out two interleaved activities, that is: i) it receives recovery information pertinent to processes to be reallocated on N_m as a consequence of reconfiguration; and ii) it transmits recovery information pertinent to processes whose back-up copies are stored in N_m and which are reconfigured onto other nodes. In both cases, information transfer is performed one segment at a time by a message containing: i) the name of the process to which the segment is associated; ii) the length of the segment; iii) the specification whether the segment belongs to the process code area, working area, INCTABLE area or buffer area; iv) the global segment name; v) the segment name local to the process; and, finally, vi) the actual contents of the segment.

To be more precise, the receiving activity is organized as follows: i) all segments which are part of the working areas and INCTABLE areas of processes allocated on N_m are retrieved, and the pertinent memory portions are considered to be free; ii) messages are received from other DFHPs running on non-faulty nodes, each relevant to one segment of a process to be reallocated in N_m; iii) for each message, the length of the segment (as specified by the message itself) is considered; memory portions made available as a consequence of point i) above and of transmission activity are searched for a free memory space of that length (this space must belong to the shared memory, if the segment is relevant to the process INCTABLE area or buffer area; otherwise, it must belong to the private memory); the free memory space is loaded with the segment contents (as specified by the message); v) a global segment name among those pertinent to N_m (i.e., among those whose Node Bits codify the quantity m) is associated to that memory

space by updating the pertinent register of the Relocation Unit RU_m; this global segment name is then mapped into the segment name local to the process (as specified by the message) by updating the pertinent register of the Address Translator AT_m.

The transmitting activity, on the other hand, is organized as follows: i) each Process Fault Handling Table stored in N_m is considered, as well as each back-up copy of a Process Fault Handling Table stored in N_m and pertinent to processes allocated in N_f, and its NEW_ALLOCATION field is accessed for obtaining the name of the node where the pertinent process must be reallocated as a consequence of a fault in N_f; ii) of the processes mentioned above, each one is considered which must be reallocated onto a node containing neither the process back-up state, nor the auxiliary back-up state: all segments constituting the back-up state are transmitted (one at a time) to the DFHP of the new node; and iii) physical memory areas pertinent to transmitted segments are made available for the receiving activity.

In the third step, $DFHP_m$ updates the INCTABLEs of all the processes reallocated in N_m, by writing into the BF field of each entry the global name of the segment where it has reallocated the pertinent buffer. Afterwards, proper messages are transmitted by $DFHP_m$ to all other DFHPs, specifying the name of each process reallocated in N_m, together with the virtual name of the first segment of the INCTABLE pertinent to that process. Then, $DFHP_m$ utilizes the same kind of information (received from others DFHPs) for: i) updating the INCTABLE Directory in node N_m (by writing into each entry the global name of the segment where the pertinent INCTABLE has been reallocated); ii) updating the Protection Unit PU_m (by assigning proper access rights to each processor, as stated in Section 3); and iii) transmitting to CCU_m the names of the processes allocated in each node (this information is hen autonomously utilized by CCU_m for updating its own data base: for

this purpose, CCU_m accomplishes proper information transfers with the other CCUs via the signalling bus).

After the node which has failed has been repaired, proper actions are performed, restoring the configuration of the whole system as it was previous to the fault. These reconfiguring actions are similar to those described above, so they will not be described again in detail.

6. CONCLUSIONS

Fault-handling strategies may be roughly divided into two classes, tailored to two different models for system architecture. In the first model the architecture is physically partitioned into processors and mass storage devices. Each processor is subject to crash: however, a crashing processor may only corrupt the data involved in the device access possibly in progress. Moreover, erroneous system outputs possibly generated by that processor are tolerated by the external environment. After repair, the processor resumes operations using the state information available in mass storage. In its turn, a mass storage device may fail, which implies the loss of all data stored in that device: however, a device failure causes no harm to the contents of other devices. In these kinds of architecture, fault-tolerance is typically pursued either by stable storage methods /28/ or other consistency techniques /29/. Typical applications concern scientific environments, data-base management and personal computing. In the architecture in the second model, there is no inherent separation between the process working areas and the state information relevant to recovery. A low-end example is a microcessor-based controller made up of few tightly interacting components, so that a fault in one of them causes both a processor crash and the loss of state information. A high-end example is a multiprocessor system in a

critical environment: here, the presence of interactions among processors outside mass storage makes it possible for a processor error to cause unacceptable disruption of operations in other processors. Fault-tolerance is usually pursued either by software replication /2,6/, or by system-wide ad hoc hardware /30,31/.

In this paper, we have been concerned with multimicroprocessor system architectures for real-time applications. The fault-handling approach adopted was intended to take into account the following hardware and software characteristics:

1) The system does not need to be provided with mass storage devices. This implies that the amount of recovery information available must be as small as possible. Moreover, the storage medium is volatile, and a failure in one processor may imply the loss of the whole contents of the associated memory banks. Processors are subject to non-crash faults: no protection is enforced between processors and the storage containing system state information by the physical separation of the respective hardware environments.
2) The system must not generate erroneous outputs towards its external environment even when a fault is present. However, the hardware is not provided with system-wide fault-handling facilities, nor is the application software replicated.
3) The software model does not define passive object entities, but only active processes whose global states represent the state of the system.

The characteristics described led to the fault-handling strategy reported in the present paper. It is believed that such a strategy may be a valid alternative to more classical methods in the case of similar hardware/software system specifications.

REFERENCES

/1/ A.L. Hopkins et al., "FTMP - A Highly Reliable Fault-Tolerant Multiprocessor for Aircraft", Proceedings of the IEEE, Vol. 66, No. 10, Oct. 1978, pp. 1221-1239.

/2/ J.H. Wensley et al., "SIFT: Design and Analysis of a Fault-Tolerant Computer for Aircraft Control", Proceedings of the IEEE, Vol. 66, No. 10, Oct. 1978, pp. 1240-1255.

/3/ D.D. Burchby et al. "Specification of the Fault-Tolerant Spaceborne Computer (FTSC)", Proceedings of the Sixth International Symposium on Fault-Tolerant Computing, Pittsburg, Penn., June 1976, pp. 129-133.

/4/ A. Avizienis et al., "The STAR (Self-Testing-And-Repairing) Computer: An Investigation of the Theory and Practice of Fault-Tolerant Computer Design", IEEE Transactions on Computer, Vol. C-20, No. 11, Nov. 1971, pp. 1312-1321.

/5/ W.N. Tog, "Fault-Tolerant Design of Local ESS Processors", Proceedings of the IEEE, Vol. 66, No. 10, Oct. 1978, pp. 1126-1145.

/6/ F. Farber, "Taskspecific Implementation of Fault Tolerance in Process Automation System", Proceedings of the Workshop on Self-Diagnosis and Fault-Tolerance, Tubingen, July 1981, pp. 84-102.

/7/ E.D. Jensen, "Distributed Systems", Advanced Course on Distributed Systems Architectures and Implementation, Lecture Notes in Computer Science, Springer-Verlag, 1980.

/8/ J.M. Jaffe, "Parallel Computation: Synchronization, Scheduling and Schemes", Ph.D. Thesis, Massachussetts Institute of Technology, August 1979.

/9/ A.K. Jones, P. Schwarz, "Experiences Using Multiprocessor Systems - A Status Report", Computing Surveys, Vol. 12, No. 2, June 1980, pp. 121-165.

/10/ B. Randell, P.A. Lee, P.C. Treleaven, "Reliability Issues in Computing System Design", Computing Surveys, Vol. 10, No. 2, June 1978, pp. 123-165.

/11/ T. Anderson, P.A. Lee, S.L. Shrivastava, "A Model for Recoverability in Multilevel Systems", IEEE Transaction on Software Engineering, Vol. SE-4, No. 6, 1978, pp. 486-493.

/12/ D.L. Russel, "State Restoration in Systems of Communicating Processes", IEEE Transactions on Software Engineering, Vol. SE-6, No. 2, Mar. 1980, pp. 183-194.

/13/ F. Grandoni et al., "The MuTEAM System: General Guidelines", Proceedings of the Eleventh Annual International Symposium on Fault-Tolerant Computing, Portland, June 1981, pp. 15-16.

/14/ G. Cioffi, P. Corsini, G. Frosini, L. Lopriore, "MuTEAM: Architectural Insights of a Distributed Multimicroprocessor System", Proceedings of the Eleventh Annual International Symposium on Fault-Tolerant Computing, Portland, June 1981, pp. 17-19.

/15/ R.J. Swan, S.H. Fuller, D.P. Siewiorek, "Cm* - A Modular, Multimicroprocessor", Proceedings of the AFIPS 1977 National Computer Conference, AFIPS, Vol. 46, pp. 637-644.

/16/ M. Ajmone Marsan, G. Conte, D. Del Corso, F. Gregoretti, "Architecture, Communication Procedures and Performance Evaluation of the μ^* Multimicroprocessor System", Proceedings of the First International Conference on Distributed Computing Systems, Huntsville, Alabama, 1979.

/17/ J. Archer Harris, D.R. Smith, "Hierarchical Multiprocessor Organizations", Proceedings of the Fourth Annual Symposium on Computer Architecture, 1977, pp. 41-48.

/18/ G. Mazarè, "MSC - A Symmetric Multi-Micro-Processor System", Proceedings of the Second Euromicro Symposium on Microprocessing and Microprogramming, Venice, 1976, pp. 135-140.

/19/ F. Baiardi, A. Fantechi, A. Tomasi, M. Vanneschi, "Mechanisms for a Robust Multiprocessing Environment in the MuTEAM Kernel", Proceedings of the Eleventh Annual International Symposium on Fault-Tolerant Computing, Portland, June 1981, pp. 20-24.

/20/ C.A.R. Hoare, "Communicating Sequential Processes", Communications of the ACM, Vol. 21, No. 8, Aug. 1978, pp. 668-679.

/21/ J.H. Saltzer, M.D. Schroeder, "The Protection of Information in Computer Systems", Proceedings of the IEEE, Vol. 63, No. 9, Sept. 1975, pp. 1278-1308.

/22/ C.H. Sequin, A.M. Despain, D.A. Patterson, "Communication in X-TREE, A Modular Multiprocessor System", Proceedings of the ACM 1978 Annual Conference, Washington, D.C., Dec. 1978, pp. 194-203.

/23/ H. Sullivan, T.R. Bashkow, "A Large Scale, Homogeneous, Fully Distributed Parallel Machine", Proceedings of the Fourth Annual Symposium on Computer Architecture, 1977, pp. 105-117.

/24/ F. Baiardi, N. De Francesco, G. Vaglini, "A Remote Process Communication Facility for the MuTEAM System", ISI Technical Report, University of Pisa, 1982.

/25/ L. Simoncini, F. Sahebam, A.D. Friedman, "Design of Self-Diagnosable Multiprocessor Systems with Concurrent Computation and Diagnosis", IEEE Transactions on Computer, Vol. C-29, No. 6, June 1980, pp. 540-546.

/26/ P. Ciompi, F. Grandoni, L. Simoncini, "Distributed Diagnosis in Multiprocessor Systems: The MuTEAM Approach", Proceedings of the Eleventh Annual International Symposium on Fault-Tolerant Computing, Portland, June 1981, pp. 25-29.

/27/ L. Lamport, "Time, Clocks, and the Ordering of Events in a Distributed System", Communications of the ACM, Vol. 21, No. 7, July 1978, pp. 558-565.

/28/ B.W. Lampson, H.E. Sturgis, "Crash Recovery in a Distributed Storage System", unpublished paper, Computer Science Lab., Xerox Paolo Alto Research Center, Palo Alto, Calif., 1976.

/29/ W.H. Kohler, "A Survey of Techniques for Synchronization and Recovery in Decentralized Computer Systems", Computing Surveys, Vol. 13, No. 2, June 1981, pp. 149-183.

/30/ R.E. Glaser, G.M. Masson, "The Containment Set Approach to Crash-Proof Microprocessor Controller Design", Proceedings of the Twelfth International Symposium on Fault-Tolerant Computing, Santa Monica, CA, June 1982, pp. 215-222.

/31/ D.P. Siewiorek et al., "A Case Study of C.mmp, Cm* and C.vmp: Part I - Experiences with Fault Tolerance in Multiprocessor Systems", Proceedings of the IEEE, Vol. 66, No. 10, Oct. 1978, pp. 1178-1199.