Architectural Support for Designing Fault-Tolerant Open Distributed Systems

Salim Hariri and Alok Choudhary, Syracuse University
Behcet Sarikaya, Bilkent University

A distributed voting algorithm and a two-level hierarchy for permanent memory are key elements in this scheme for supporting fault tolerance in open distributed systems.

A distributed system consists of autonomous computing modules that interact with each other using messages. Designing distributed systems is more difficult than designing centralized systems for several reasons. Physical separation and the use of heterogeneous computers complicate interprocessor communication, management of resources, synchronization of cooperating activities, and maintenance of consistency among multiple copies of information. The main advantages of distributed systems include increased fault-tolerance capabilities through the inherent redundancy of resources, improved performance by concurrently executing a single task on several computing modules, resource sharing, and the ability to adapt to a changing environment (extensibility).[1]

Distributed systems cover a wide range of applications. Recent advances in VLSI devices and network technology will further increase the use of distributed systems. As the complexity of these systems increases, so does the probability of component failure, which can adversely affect the performance and usefulness of such systems. Thus, reliability, availability, and fault tolerance become important design issues in distributed systems. Fault tolerance is the system's ability to continue executing despite the occurrence of failures. Increasing the reliability and fault tolerance of a system involves a trade-off between the cost of failure (for example, costs incurred by incomplete or incorrect computations) and the cost of incorporating redundancy and recovery mechanisms.

Because of their inherent redundancy, distributed systems provide a cost-effective way to apply fault-tolerance techniques. Open distributed systems provide universal connectivities among their components because their designs are based on the standard protocols adopted by the International Standards Organization (ISO). In this computing environment, interacting processes communicate through messages that traverse a stack of software layers. Consequently, applying fault-tolerance techniques to execute critical tasks can be costly in terms of execution time.

In this article, we first provide an overview of the main techniques for designing

0018-9162/92/0600-0050$03.00 © 1992 IEEE

fault-tolerant software and hardware systems. We identify the important features of the building blocks (computers, memories, buses, etc.) that can support an efficient implementation of fault-tolerant open distributed systems (FTODS). Taking into account the features of these building blocks, we propose an organization for FTODS. In FTODS, the algorithms needed for transferring files and synchronizing the concurrent activities of the computing modules — and for recovery — are ISO standard protocols. We propose the use of low-level voting and recovery algorithms that can run as a layer of software above the operating system to make the open distributed system an attractive environment for applying fault-tolerant techniques.

Glossary of acronyms

AAT — Atomic action tree
ACSE — Association control service element
ASE — Application service element
CCR — Commitment, concurrency, and recovery
DVA — Distributed voting algorithm
FTAM — File transfer and management
FTMP — Fault-tolerant multiprocessor
FTODS — Fault-tolerant open distributed systems
HPM — Hierarchical permanent memory
JTM — Job transfer and manipulation
MPM — Magnetic permanent memory
MTTF — Mean time to failure
ODP — Open distributed processing
ODS — Open distributed systems
OSI — Open Systems Interconnection
RDA — Remote database access
SIFT — Software-implemented fault tolerance
SPM — Semiconductor permanent memory
TP — Transaction processing
TR — Transaction reliability
VTP — Virtual terminal protocol

Design considerations for fault tolerance

Fault tolerance, a system's ability to continue executing its tasks despite the occurrence of failures, can be achieved by fault masking. Masking (also called static redundancy) is incorporated into the design to concurrently mask faults and prevent their propagation to other modules. The most common example of static redundancy is the triple modular redundant system. Another approach for providing fault tolerance is dynamic redundancy, which uses spare components to replace faulty modules once they are detected. Still another approach — a combination of these two, called hybrid redundancy — applies static and dynamic redundancy to achieve fault tolerance. In general, the design of a fault-tolerant computer involves one or more of the following strategies: fault masking, fault detection, fault containment, fault diagnosis, repair/reconfiguration, and fault recovery[2] (see the sidebar "Strategies for designing fault-tolerant computers").

Designing a fault-tolerant distributed system is more involved than designing a fault-tolerant centralized system. Two main problems must be addressed during design:

(1) Concurrency control, which involves scheduling concurrent execution of tasks on different nodes such that their results are identical to a serial execution of the tasks (serializability requirement).

(2) Redundancy management, which involves preserving consistency among replicated resources and maintaining the state information with backup modules to support recovery.

Transactions are an important programming paradigm for simplifying the design of reliable distributed applica-

Strategies for designing fault-tolerant computers

Many techniques have been used to build fault-tolerant computers. They include

Fault masking: Concurrent masking and correction of generated errors.

Fault detection: Use of hardware and software mechanisms to determine the occurrence of a failure. Fault detection mechanisms include concurrent fault detection, stepwise comparison, and periodic testing to determine whether computers or communication links are operating correctly.

Fault containment: Prevents propagation of erroneous or damaged information in the system after a fault occurs and before it is detected.

Fault diagnosis: Locates and identifies the faulty module responsible for a detected error.

Repair/reconfiguration: Eliminates or replaces the faulty module, or provides means to bypass it.

Fault recovery: Corrects the system to a state acceptable for continued operation.

Most of these techniques have been used to build such computers as the Tandem 16 NonStop system, the Stratus computer system, the VAXft 3000, the Teradata and Sequoia systems, the fault-tolerant multiprocessor (FTMP), the software-implemented fault-tolerance (SIFT) system, and AT&T's Electronic Switch System (ESS).[1,2] The effectiveness of fault-tolerance techniques can be measured by the "coverage," defined as the conditional probability of recovering from a fault once it occurs.[3] It is difficult to measure this parameter because it involves evaluating the probability that fault detection, fault diagnosis, repair/reconfiguration, and recovery algorithms are operating correctly.

References

1. D.P. Siewiorek and R.S. Swarz, *The Theory and Practice of Reliable System Design*, Digital Press, Bedford, Mass., 1982.
2. D.P. Siewiorek, "Fault Tolerance in Commercial Computers," *Computer*, Vol. 23, No. 7, July 1990, pp. 26-37.
3. J.B. Dugan and K.S. Trivedi, "Coverage Modeling of Fault-Tolerant Systems," *IEEE Trans. Computers*, Vol. C-38, No. 6, June 1989, pp. 775-787.

Transactions

A transaction can be defined as a collection of operations having the following three properties[1]:

Failure atomicity: Either all operations are performed successfully or their results are undone when a failure occurs.

Permanence: The results of committed transactions will not be lost.

Serializability: The results of executing transactions concurrently are the same as if they were executed serially.

Use of the transaction concept to model distributed computations provides a convenient means to solve the concurrency control and redundancy management problems.[1] The concurrency control problem consists of three tasks: assigning an order to all transactions, identifying conflicting transactions, and synchronizing transactions to resolve the identified conflicts. Basically, there are three approaches to concurrency control: time-stamp-based schemes, locking protocols, and optimistic techniques.[2]

References

1. *Distributed Systems*, S. Mullender, ed., Addison-Wesley, Reading, Mass., 1989.
2. P.A. Bernstein, V. Hadzilacos, and N. Goodman, *Concurrency Control and Recovery in Database Systems*, Addison-Wesley, Reading, Mass., 1987.

tions (see the "Transactions" sidebar). Techniques for managing redundancy and maintaining consistency of replicated objects are broadly characterized as centralized- and decentralized-control algorithms. The centralized-control approach supports strong consistency requirements and prevents deadlocks, but it is susceptible to single points of failure. The decentralized-control approach supports weak consistency requirements (when it is permissible to have the state of some replica out of date for a short period of time), and therefore it can potentially increase a system's throughput. The primary-copy algorithm[3] applies the centralized-control strategy to ensure the consistency of replicated resources. In this scheme, one node is designated as the primary node and made responsible for serializing updates. When the update values have been computed, the primary node broadcasts them to all other nodes in the system. The primary node then waits to receive acknowledgments from all nodes before processing the next transaction. The main problem with this scheme is that it permits no parallelism among transaction executions.

Voting algorithms have also been used to ensure consistency of replicated resources. In this scheme, managers of replicated resources use a common set of rules to determine whether an update can be made. The algorithm's control can be centralized or decentralized, depending on whether the voting is done at one site or multiple sites.[3] In addition to maintaining consistency of replicated resources, redundancy management is responsible for system recovery in the presence of node crashes and communication link failures.

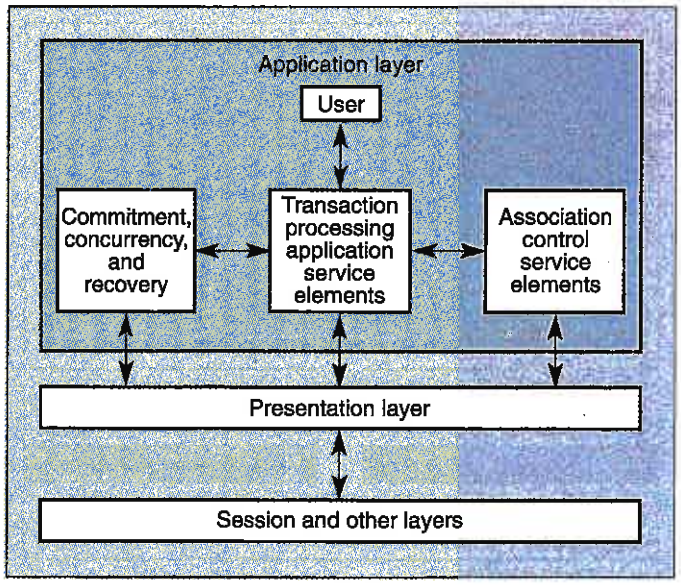

Figure 1. The structure of an application layer.

Open distributed systems

In this article, we investigate techniques for providing architectural support to improve the execution of distributed applications that use the Open Systems Interconnection standards. The main goal of the OSI reference model is to provide universal connectivity among heterogeneous computers. The reference model is designed to structure communication hardware and software in a layered architecture.[4] ISO committees are working on an architecture in line with the reference model for open distributed processing (ODP). This effort aims to combine the OSI model with a database model to arrive at a global framework for designing distributed systems. In such an environment, any computer would be open for communication and could be integrated easily with the existing distributed systems to perform certain tasks. Implementation of the communication protocols as layered software tends to be very slow and consequently limits the scope of applications for open distributed systems.

The application layer is implemented as several application service elements (ASEs), with one ASE serving them all. This element is called the association control service element (ACSE), and it provides association (connection) establishment/disconnection service to other ASEs. In open distributed systems, distributed applications are implemented by the services that the ASEs provide. The application layer services can be in the form of file transfers using the FTAM (file transfer and management) protocol, remote database access using

the RDA protocol, job transfers using the JTM (job transfer and manipulation) protocol, a virtual terminal using the VT protocol, and transaction processing using the TP protocol.

To achieve reliable and fault-tolerant computing in open distributed systems, the ASEs use the commitment, concurrency, and recovery (CCR) services provided by a special ASE called the CCR protocol.[4] CCR is a standard two-phase commit protocol that provides the services needed to achieve concurrency control and recovery during execution of application layer tasks such as FTAM, TP, VTP, etc. Figure 1 shows the OSI communication model and the interactions among the ASEs of the application layer.

Architectural support for FTODS

In this section we identify features that should be supported by the computing modules of open distributed systems to facilitate an efficient implementation of fault-tolerant algorithms. On the basis of this criteria, we propose an organization for fault-tolerant open distributed systems, the architecture of its building blocks, and the required algorithms and protocols. The architecture of the computing modules should support reliable broadcasting, self-repair/recovery, selective fault tolerance, and permanent memory (see the sidebar "FTODS computing module capabilities").

Organization of the FTODS. An FTODS comprises a set of computing modules we refer to as nodes. Nodes communicate and interact with each other by broadcasting their messages on a redundant broadcast medium. A

FTODS computing module capabilities

The architecture of the computing modules should facilitate the efficient implementation of fault-tolerant algorithms. This architectural support can be provided by the following capabilities:

Reliable broadcasting. Reliable broadcasting provides means for a set of processes to communicate in spite of failures and is used frequently as a primitive operation to implement reliable distributed applications.[1] It has been shown that reliable broadcasting provides an efficient solution to many problems — for example, distributed consensus, distributed synchronization, replicated update, and transaction management in database systems.[2] Furthermore, these reliable protocols will run efficiently on the underlying architecture if its communication network has a broadcast capability.

Self-repair/recovery. Recovery in distributed systems with replicated resources, computations, and database systems is a nontrivial task. Moreover, the overhead of recovery can degrade system performance significantly.[2] Hardware recovery blocks have been proposed to reduce overhead during the save operations of system state and to speed up recovery when faults are detected in a multiprocessor system.[3] The literature is rich with techniques that can be used to support self-repair and recovery. For example, the use of static redundancy to achieve fault masking has been used in the c.vmp (computer-voted multiprocessor) computer.[4] Also, Kuhl and Reddy[5] have addressed fault diagnosis at the system level and the conditions under which nodes can diagnose the failure of other nodes to achieve self-test.

The architecture of the computing modules should support a hierarchical approach for recovery such that most of the time-consuming tasks are executed at a lower level of this hierarchy. The use of static (masking) redundancy and diagnostic routines simplifies the tasks involved in fault detection, self-reconfiguration and repair, and recovery. Providing the computing modules with these features could significantly reduce the complexity of recovery at the application level. With the proliferation of VLSI chips, I/O processors, controllers, and memory, it is now reasonable to use redundant components in designing the computing modules.

Selective fault tolerance. Since not all task operations require fault tolerance, it is desirable to run only the critical operations in a fault-tolerant mode and the rest in a normal mode. This will lead to a significant improvement in performance without compromising the fault-tolerance requirements. Consequently, the architecture of the computing modules should support dynamic reconfiguration such that the processors within a node can be configured for use as a masking redundancy during critical operations and as a multiprocessor system during noncritical operations. This capability has been supported by the c.vmp, which contains three processor-memory pairs that can operate independently and can also provide fault-tolerant operations.[4]

Hierarchical permanent memory system. Most recovery algorithms needed to achieve fault-tolerant computing rely on permanent storage.[1] Stable storage is used to store the checkpoints of a system state; these checkpoints will be used to restore the system to the previous fault-free checkpoint state when a failure occurs during normal operation. Stable storage is normally constructed using dual magnetic disks. Performance of fault-tolerant algorithms can be improved significantly if stable storage is implemented in a two-level hierarchy in which semiconductor permanent memory is used in the first level and magnetic permanent memory is used in the second. The SPM acts as a buffer between the processor and the MPM.

References

1. *Distributed Systems*, S. Mullender, ed., Addison-Wesley, Reading, Mass., 1989.
2. J. Chang and N.F. Maxemchuk, "Reliable Broadcast Protocols," *ACM Trans. Computer Systems*, Vol. 2, No. 3, Aug. 1984, pp. 251-273.
3. Y.-H. Lee and K.G. Shin, "Design and Evaluation of a Fault-Tolerant Multiprocessor Using Hardware Recovery Blocks," *IEEE Trans. Computers*, Vol. C-33, No. 2, Feb. 1984, pp. 113-124.
4. D.P. Siewiorek and R.S. Swarz, *The Theory and Practice of Reliable System Design*, Digital Press, Bedford, Mass., 1982.
5. J.G. Kuhl and S.M. Reddy, "Fault-Diagnosis in Fully Distributed Systems," *Proc. 11th Int'l Symp. Fault-Tolerant Computing*, IEEE CS Press, Los Alamitos, Calif., Order No. 350 (microfiche only), June 1981, pp. 100-105.

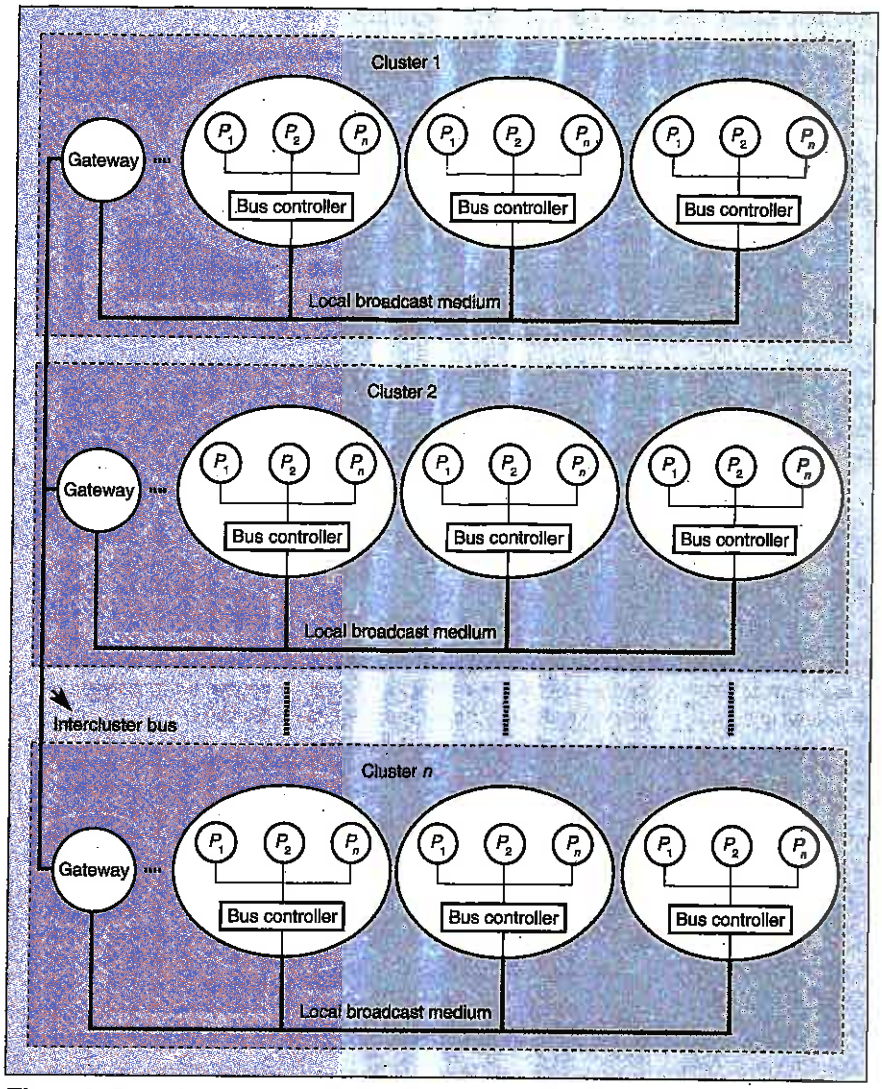

Figure 2. Organization of the fault-tolerant open distributed system.

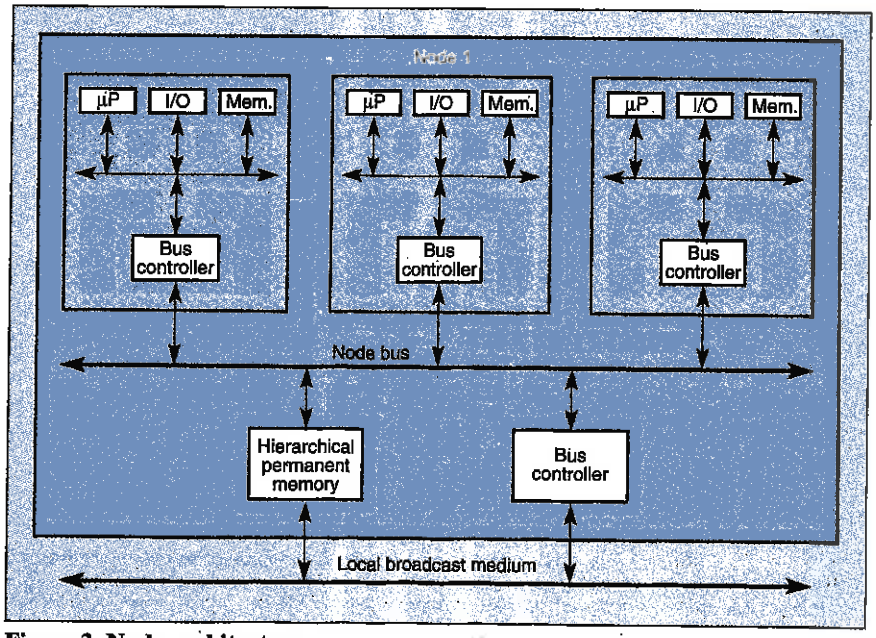

Figure 3. Node architecture.

set of nodes forms a cluster. A cluster (C_i) communicates with another cluster (C_j) through the gateway associated with each cluster (see Figure 2).

Node architecture. Each node has several processing elements which can be configured dynamically to form either a redundant computing module or a shared-bus multiprocessor system. The processing elements communicate with each other via a redundant node bus. Components of a node (as shown in Figure 3) include a general-purpose microprocessor, an input/output system, and bus controller subsystems. The number of processing elements needed at each node depends on the reliability and fault-tolerance requirements. A node has two operational modes: fault tolerant and multiprocessing. For noncritical tasks, a node's processors can be configured as a shared-bus multiprocessor system. For critical tasks, the node's processors execute the same task synchronously and use a voting procedure to mask out the effect of faulty processors. The coordinator processor (P_c), which is chosen from the set of fault-free processors according to a predefined selection procedure, supervises the voting algorithm and communication with other nodes.

Hierarchical permanent memory. Permanent memory provides secure data storage for the state of a node and any other information relevant to the execution of a transaction. Consequently, it is possible to commit the transaction atomically or undo all its actions should that transaction be aborted. Permanent memory can be implemented using magnetic or semiconductor technology. Figure 4 shows the organization of the hierarchical permanent memory (HPM), which uses a semiconductor permanent memory (SPM) at the first level and magnetic permanent memory (MPM) at the second level. The SPM contains two battery-backup RAM units, a comparator unit, and several bus interface units. The MPM consists of dual magnetic disks and a comparator.

In the proposed HPM, the SPM acts as a buffer between the coordinator processor of a node and the MPM; as a result, the HPM unit's effective access time is reduced. The coordinator processor of a node updates the SPM atomically according to the following procedure:

(1) After obtaining a majority consensus on the data to be committed, the coordinator processor places the data on the node bus with a write signal.

(2) The values from the bus are written into the two semiconductor memories simultaneously.

(3) The comparator module immediately reads back and compares the updated locations.

(4) If the values differ, an abort signal is sent to the coordinator processor via the node bus indicating that the values need to be rewritten. This process can be repeated up to a predefined number of times before an error signal is generated. If the comparison operation produces a match, then the updated values are committed and an appropriate signal is sent to the coordinator processor.

Figure 5 is a flowchart describing an atomic write operation. Error detection and correction codes can also be used to increase the reliability and simplify the diagnosis of the memory system. However, coding techniques alone cannot provide fault tolerance against crashes of memory devices. A similar procedure is used for a read operation. In addition to the fault-tolerance capability hierarchical permanent memory provides, its use also improves the performance of recovery protocols.

Cluster coordinator and gateway. For each set of nodes forming a cluster (C_i), there is a node designated as the cluster coordinator (C_c). The nodes of a cluster are ordered in a predetermined priority list so that any fault-free node knows the procedure for selecting the C_c node. The C_c node periodically receives status messages from the nodes in its cluster. Also, the C_c supervises the recovery procedure when one of its nodes is in a crashed state. A cluster's gateway forwards all messages routed to nonlocal nodes through the gateways connected to the intercluster communication link. The remote gateways pick up the messages addressed to one of their nodes.

Selective fault-tolerance capability. Redundancy and fault-tolerant algorithms are used to ensure atomic execution of critical tasks and system recovery when faults occur. For example, updating a bank account is a critical task, and its execution should be controlled by a commit protocol. In this case, the critical operations are those that update the bank accounts. However, there are other operations that do not affect the system consistency requirements (reading a set of records, searching the database, etc.), so they do not need a commit protocol to control their execution.

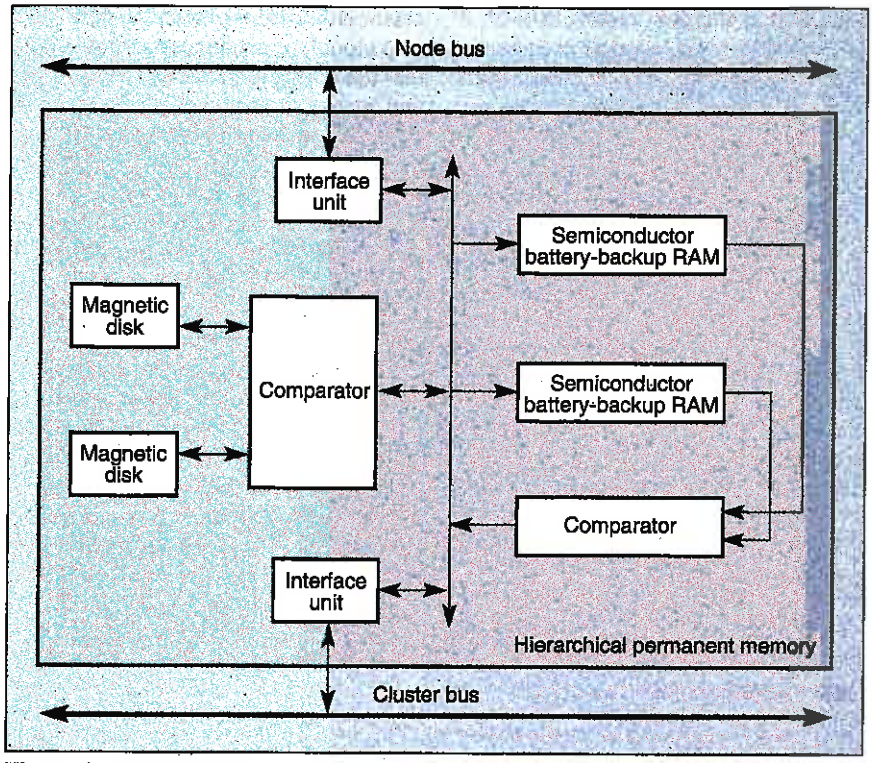

Figure 4. Hierarchical permanent memory.

Since critical operations constitute only a small part of all the operations, redundancy in the architecture can be exploited to improve performance through parallel processing. However, for a node to operate in two modes — redundant mode and multiprocessing mode — the system should provide techniques for reconfiguration.

Support for two processing modes is provided by monitors. A monitor is a layer of software embedded above the operating system. The tasks are represented using a graph whose nodes represent computational structures and whose arcs represent the dependency constraints between the computations. Critical tasks are distinguished from noncritical tasks using system primitives and semantics of the computation. The monitor maintains a queue of ready tasks that can be executed concurrently as soon as a processor is available. The monitor schedules the tasks in a first-in, first-out manner. However, scheduling must incorporate execution of critical tasks, since they require all the processors on a node.

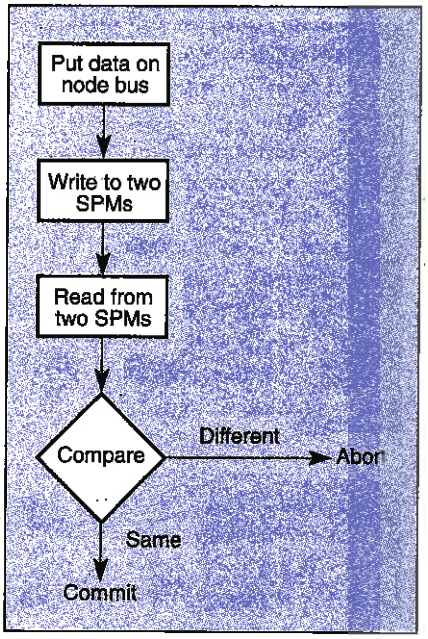

Figure 5. An atomic write operation.

Two scheduling schemes can be used for scheduling critical tasks. The first uses preemptive scheduling in which a critical task to be scheduled preempts all other tasks. When the monitor rec-

ognizes that the next task is critical, it preempts the tasks on other processors. In the second scheme, the monitor waits for all current tasks to complete before scheduling a critical task, and it does not schedule any noncritical tasks during that period, even if processors are available. The first scheme has the advantage that critical tasks are completed as soon as possible. But the overhead of preempting tasks can be significant because the state of all the current processes must be saved. The second scheme does not require saving the states of the current processes, but the processors may remain idle for a long period, reducing utilization and throughput.

Distributed voting algorithm. In our analysis, we assume that a faulty processor either stops producing data (fail-stop model) or produces corrupted data that the voting algorithm can recognize and use as a symptom of a faulty processor. Processor faults can be caused by a malfunction of its hardware and/or software. A processor can assume only two states: faulty or fault free. During the fault-tolerance mode of operation, a node's processors are configured to execute the same task (static redundancy) and the system memory is reconfigured as a hierarchical permanent memory. A coordinator processor (P_c) is associated with each node. The P_c supervises the distributed voting algorithm and commits the tasks' execution. Selection of the P_c follows a predefined procedure. For example, if each processor is identified by a number (ID), then P_c can be selected as the fault-free processor with the maximum ID value.

Figure 6 describes the voting algorithm and related procedures for distributed voting in the FTODS environment. Each processor P_i computes a result (or a set of results) that must be voted on before it can be committed. The functions used to compute these results depend on the application transactions. In phase 1, these results are computed and each P_i participating in the distributed computation broadcasts its results on the node bus using the broadcast primitive. Phase 2 involves voting on the result and determines whether the result can be committed. Each P_i receives the values from all other P_i's and independently determines the confidence in the values by comparing them.

```
Phase 1:
  Each P_i in a node does the following, (1 ≤ i ≤ n)
    1. vote = function ("parameters") /*computation depends on the application*/
    2. broadcast ("node", vote, P_name) /*broadcast "vote" to all processors (P_name) in "node"*/

Phase 2:
  /*Each P_i does the following, (1 ≤ i ≤ n)*/
    vote[j] = recv_msg (P_j, vote, P_name) 1 ≤ j ≤ n, j ≠ i /*rec. vote from all processors*/
    If (vote[i] = vote[j]) ∀_{i,j}, 1 ≤ i,j ≤ n
      begin
        result = vote[i]; /*result contains vote to be committed*/
        vote_commit ("all"); /*vote commit with param "all"*/
      end
    else if (vote[i] = vote[j]) for at least k votes s.t. k ≥ ⌈n/2⌉
      begin
        result = vote of majority; /*result contains value of the majority*/
        if ((P_i = P_c) .and. (vote[i] ≠ result)) /*if current coordinator not in majority*/
          select_coordinator (P_k ∈ majority); /*select new coordinator*/
        vote_commit ("majority"); /*commit the majority value*/
        if (P_i = P_c)
          local_recovery (for all P_j not in majority); /*start recovery of processors not in majority*/
      end;
    else /*no majority */
      begin
        P_i (status) = local_diagnostic (P_i), 1 ≤ i ≤ n /*start local diagnostics*/
        if (P_i (status) = "okay")
          begin
            select_coordinator ("node"); /*select new coordinator from "okay" processors*/
            if (P_i = P_c) /*new coordinator does the following*/
              vote_commit (new_majority); /*commit new majority*/
          end;
        else if (P_i (status) ≠ "okay") ∀_i , 1 ≤ i ≤ n /*all processors have failed*/
          print ("complete node failure, external recovery required");
      end.
```

Figure 6. The distributed voting algorithm.

There can be three possible outcomes of this comparison. First, all values match with each other — a complete consensus. In this case, the coordinator processor P_c broadcasts the result to all P_i's. Each P_i acknowledges by sending a broadcast acknowledge message, and the result is written in the permanent memory.

In the second outcome, only a majority is obtained on the result and the values of some P_i's do not agree with that of the majority. In this case, there are two groups of processors, those belonging to the majority and those that don't. Since a majority is sufficient to commit a new value, the distributed voting algorithm (DVA) is executed such that it uses the majority result as the correct one. If the current coordinator is part of the majority, it coordinates the current DVA and initiates a recovery procedure for the P_i's in the minority. If the current coordinator is not in the majority, then the P_i's belonging to the majority select a new coordinator using the "select_coord()" procedure. This new coordinator now coordinates commit and recovery for the minority processors.

In the third outcome, a majority is not obtained. This triggers the "self-diagnostic()" procedure associated with each P_i. The self-diagnostic procedure returns the status of each processor as either "okay" or "failed" (actually, obtaining anything other than "okay" implies "failed"). If none of the processors return a status "okay," the node (all n processors in the node) is considered "failed." This requires external recovery, which the cluster coordinator will perform as part of the distributed node recovery algorithm. If some P_i's are okay after the self-diagnostic procedure, they broadcast their status and then select a new coordinator from this new set. Voting is repeated for the new set, and the recovery procedure is initiated for other P_i's.

Distributed node recovery algorithm. One node is designated as the cluster coordinator (C_c) for each cluster. Selection of the C_c follows a predefined procedure similar to that used in selecting the P_c of a node.

Figure 7 describes the distributed node recovery algorithm. Each P_c of a node periodically broadcasts a status message on the local broadcast medium. The current C_c checks the status of all node coordinators. If any node is crashed and does not send a status message to the C_c, the C_c copies the state of that node as well as the node's task_queue. It then assigns these tasks to other nodes in the cluster, choosing nodes with the minimum load. If a node that crashed earlier has recovered and sends an "okay" message to the C_c, the C_c updates its own record to reflect this

```
Each Pc_i (node coordinator) in a cluster does the following, (1 ≤ i ≤ m)
Forever do /*periodically*/
        broadcast ("cluster", status, P_c) /*broadcast "status" to all node coordinators in the cluster*/

/*The cluster coordinator is one of the node coordinators*/

    recv_msg (Pc_j, status, P_name) 1 ≤ j ≤ m, j ≠ i /*rec. status from all node coordinators*/
    If (Pc_i = C_c) /*if I am the cluster coordinator*/
      begin
        For (i = 1 to m) do /*check status of all nodes*/
          If (status (Pc_i) ≠ "okay") /*any failed node?*/
            begin
              state = read (Pc_i, state_block) /*read the state of Pc_i from its HPM*/
              task_queue = Read (Pc_i, task_queue) /*Obtain the tasks of Pc_i*/
              Pc_j = select (minload, cluster) /*choose a node with minimum load currently*/
              Send (Pc_j, state) /*copy state of crashed node to the chosen node*/
              Send (Pc_j, task_queue) /*assign tasks to the new node*/
              rec_status (Pc_i) = failed /*record this with C_c*/
              recover (Pc_i) /*recover the crashed node by copying updated information*/

/*The recovery may involve replacement of the crashed node's hardware needs to be replaced (i.e., if catastrophic
failure occurred)*/

            end
          else if (status (Pc_i) = okay) and (rec_status (Pc_i) = failed)) /*Pc_i was repaired but record was not*/
            rec_status (Pc_i) = okay /*update C_c record*/
      end;
    else if (Pc_i ≠ C_c) /*if I am not the cluster coordinator*/
      if (status (C_c) ≠ "okay") /*the cluster coordinator failed*/
        select_cluster_coordinator(); /*select new cluster coordinator*/
    end
```

Figure 7. The distributed node recovery algorithm.

```
Procedure vote_commit (parameter: set_processor);
/*set_processor: a list of processors that are fault-free, e.g. all, majority etc.*/
  if (P_i = P_c) /*if I am the coordinator*/
    begin
      broadcast ("set_processor", result, P_name); /*reliable broadcast result to processors in "set_processor"*/
      k = || set_processor ||; /*cardinality of set_processor*/
      for j = 1 to k
        recv_msg (P_j ∈ set_processor, bcast_ack, P_name); /*rcv. acknowledgment from processor in
set_processor*/
      exit;
    end;
  else if ((P_i ≠ P_c) .and. (P_i ∈ set_processor)) /*other than coordinator processor*/
    begin
      recv_msg (P_c, result, P_name); /*rec. result from coordinator*/
      bcast_ack (P_c, P_name); /*acknowledge to the coordinator*/
    end;
    return;
end vote_commit.

Procedure select_coordinator (parameter: set_processor);
  if (my_node (status) = "okay")
    begin
      broadcast (set_processor, status, P_name); /*broadcast status*/
      recv_msg (set_processor, status, P_name); /*rec. status from other processor*/
        if (my_node = max (set_processor)) /*e.g. largest node_id */
          mynode = P_c; /*I am new coordinator*/
    end;
    return;
end select_coordinator
```

Figure 8. Some procedures used in the distributed voting algorithm.

change. If other node coordinators do not receive a status message from the C_c, that is, if the C_c itself failed, then node coordinators select a new C_c following a procedure similar to that for selecting a node coordinator (see Figure 8). Once a new C_c is selected, it repeats the above procedure to check for node failures.

Implementation issues. The architectural support provided by the computing modules of fault-tolerant open distributed systems supports the trend toward open distributed systems. In FTODS, each computing module of a node has its own operating system and a runtime system that includes the distributed voting algorithm, the distributed node recovery algorithm, and the monitor (to schedule tasks and switch them between the two modes of operation, and to do other housekeeping tasks). The fault-tolerance, concurrency control, and redundancy management algorithms use standard protocols and are implemented at the application layer as application service elements. In this environment, development of reliable applications is significantly easier because they are not concerned with implementing the fault-tolerance, concurrency, and recovery techniques; these techniques are provided to the applications as services by an ASE such as the CCR protocol.

We can better understand this architectural support by studying the main steps incurred during execution of a standard two-phase commit protocol (such as the CCR protocol). For example, to execute a transaction atomically, the master node running this transaction broadcasts a message (C-Begin) to all nodes involved in the transaction execution, indicating the beginning of an atomic execution. Since the underlying communication structure of FTODS supports broadcasting, we expect the transfer of the C-Begin message to be efficient. Once the C-Begin message is received at each slave node, the monitor switches to the fault-tolerant mode of operation, stores the system state in the hierarchical permanent memory, and checks the possibility of running the actions associated with the transaction. If an action can run successfully, the slave node sends an "okay" message (C-Prepare); otherwise, it sends a "failure" status message (C-Refuse).

Redundant execution of actions in the fault-tolerant mode, use of the distributed voting algorithm with provision to recover by itself, and use of two-level permanent memory will all contribute to improved performance, reliability, and fault tolerance. In the second phase, if the master node receives a C-Prepare message from all the slave nodes, it commits the transaction by broadcasting the C-Commit message; otherwise, it broadcasts the C-Rollback message. Also, tasks in this phase will complete quickly because the

proposed architecture supports the broadcast capability and the rollback procedure.

We believe that providing architectural support at the node level and using standard protocols will significantly simplify the development of reliable distributed applications, thereby making open distributed systems an attractive computing environment.

Reliability analysis of FTODS

Node reliability. Assume that r represents the reliability of each processor for a given period of time T. This reliability measure should take into account the failures caused by hardware as well as by software. Detailed Markovian methods can be used to predict the combined reliability measure that takes into consideration hardware faults and software errors — for example, design errors related to system overloads, overflow/underflow of queues, etc. Assume also that a processor failure is exponentially distributed with a failure rate λ. Let the number of processors in a node be N_n and f denote the number of faulty processors at a given time t. Depending on the number of faults, the distributed voting algorithm uses different procedures, as follows:

Case 1: Number of faults $f \le \lceil N_n/2 \rceil$. In this case, a majority vote is attainable and the results obtained by the faulty processors can be masked out concurrently by the coordinator processor without any extra delay.

Case 2: Number of faults $\lceil N_n/2 \rceil \le f \le N_n - 1$. In this case, the majority of processors are faulty. However, there is at least one fault-free processor that can be identified by the diagnostic routines. This processor ensures reliable execution of the tasks assigned to its node, but with a time penalty that results from invoking the local diagnostic procedures.

Case 3: Number of faults $f = N_n$. In this case, all processors of a node are faulty; consequently, the node is in a failed state. The cluster coordinator invokes the distributed node recovery procedure to start a higher level recovery procedure, as previously described.

Node reliability can thus be defined as the probability of the node's being in either case 1 or case 2, that is, $f \le N_n - 1$. The expression for node reliability is obtained by computing the probability that at least one processor is operating successfully and is given by

$$R_n(N_n) = C \times R_{\text{bus}} \sum_{f=0}^{N_n-1} \binom{N_n}{f} r^{N_n-f}(1-r)^f \quad (1)$$

where C denotes the coverage of the recovery procedure, R_{bus} denotes the reliability of the system bus, and

$$\binom{N_n}{f}$$

is the binomial factor and is given by

$$\frac{N_n!}{(N_n-f)!f!}$$

In the above expression, the term r^{N_n-f} denotes the probability of having N_n-f fault-free processors, while $(1-r)^f$ denotes the probability of having f faulty processors. The

$$\binom{N_n}{f}$$

denotes the number of combinations in which there are f faulty processors chosen from N_n in a node. If the coverage factor is equal to one, the node can be viewed as a parallel redundant system with a redundancy level of N_n. Node reliability can be evaluated as $(1 - (1-r)^{N_n})$.

Node reliability can be expressed with respect to time, if we assume that the processors fail according to an exponential distribution function with a failure rate λ. Consequently, node reliability at a given time t is given by

$$R_n(N_n) = C \times R_{\text{bus}} \sum_{f=0}^{N_n-1} \binom{N_n}{f} (\exp^{-\lambda t})^{N_n-f}(1-\exp^{-\lambda t})^f \quad (2)$$

Coverage C is an important parameter, and system reliability is extremely sensitive to its value. The coverage factor reflects the system's ability to recover automatically from a fault once it occurs during normal operation. It depends on the techniques used to detect, mask, locate, and repair faults, and to reconfigure and recover from the effects of a failure. The methods used to predict coverage are therefore based on assumptions about the expected behavior of faults and how they are handled once they occur. Dugan and Trivedi[5] presented several methods for predicting the coverage factor for different types of error behavior assumptions. In FTODS, a distributed voting algorithm is used to detect faulty processors and to mask their errors dynamically. Therefore, no recovery is needed as long as a majority vote can be obtained (case 1). Also, this algorithm uses a redundant system bus for comparing the results obtained by the processors. Since there is no single point of failure in the FTODS architecture and in the fault-tolerant algorithms, the coverage C is expected to be high; in this analysis it is assumed to be 1.

A node's mean time to failure can also be evaluated from the above expression (Equation 2) by integrating the node reliability expression:

$$\text{MTTF} = \int_{t=0}^{t=\infty} R_n$$

To measure the reliability improvement as a result of introducing redundancy, we define a measure called the reliability improvement factor (RIF). This measure describes the relative increase in reliability for using N_n redundant processors to the maximal possible increase in reliability. Let's assume that $R_n(1)$ denotes the simplex reliability of a node. The maximal increase in reliability is obtained when $R_n(1)$ is increased to 1. The RIF for a given redundancy level (N_n) is computed as

$$\text{RIF} = \frac{R_n(N_n) - R_n(1)}{1 - R_n(1)}$$

Figure 9 shows the RIF obtained for three different levels of redundancy (3, 4, and 5). In this analysis, the reliability of a simplex bus is assumed to be a constant and equal to 0.95, because we are interested in studying the effect of redundancy level on node reliability. It is clear that more than 95 percent of the possible reliability improvement can be achieved when four processors are used. However, a triple modular redundancy configuration (level 3) could

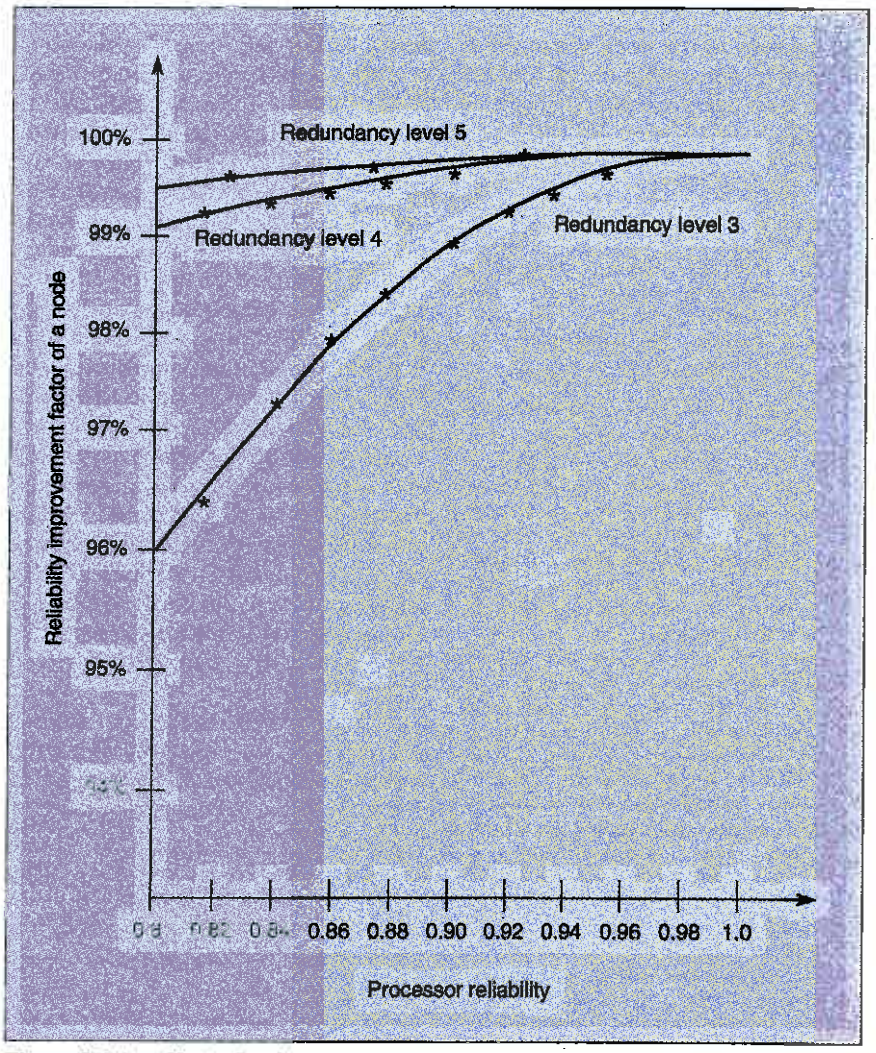

Figure 9. The effect of redundancy level on node reliability.

be sufficient for situations in which the processor's initial reliability is high. The same analysis can be used to measure the improvement in the MTTF when different redundancy levels are used.

Reliability analysis of an atomic transaction. Let T denote a transaction with a collection of n actions, that is, $T = a_1, a_2, \ldots, a_n$, where a_i represents an action to run on a node.

The set of nodes that run the actions of a given transaction (T) and the set of links connecting them form a tree referred to in CCR protocol as an atomic action tree. An AAT is assumed operational when all its components (nodes and links) are operational.

Figure 10 shows a transaction consisting of three actions a_1, a_2, and a_3 in which each action can run redundantly on two nodes of a cluster. In this example, action a_1 can run on node x_1 or x_2, a_2 can run on node x_3 or x_4, and a_3 can run on x_5 or x_6. Because of this redundancy, eight possible trees can be used to run this transaction:

$$\mathrm{AAT}_1 = x_{b_1}x_{b_2}x_{b_3}x_{b_4}x_{g_1}x_{g_2}x_{g_3}x_1x_3x_5$$
$$\mathrm{AAT}_2 = x_{b_1}x_{b_2}x_{b_3}x_{b_4}x_{g_1}x_{g_2}x_{g_3}x_1x_3x_6$$
$$\mathrm{AAT}_3 = x_{b_1}x_{b_2}x_{b_3}x_{b_4}x_{g_1}x_{g_2}x_{g_3}x_1x_4x_5$$
$$\mathrm{AAT}_4 = x_{b_1}x_{b_2}x_{b_3}x_{b_4}x_{g_1}x_{g_2}x_{g_3}x_1x_4x_6$$
$$\mathrm{AAT}_5 = x_{b_1}x_{b_2}x_{b_3}x_{b_4}x_{g_1}x_{g_2}x_{g_3}x_2x_3x_5$$
$$\mathrm{AAT}_6 = x_{b_1}x_{b_2}x_{b_3}x_{b_4}x_{g_1}x_{g_2}x_{g_3}x_2x_3x_6$$
$$\mathrm{AAT}_7 = x_{b_1}x_{b_2}x_{b_3}x_{b_4}x_{g_1}x_{g_2}x_{g_3}x_2x_4x_5$$
$$\mathrm{AAT}_8 = x_{b_1}x_{b_2}x_{b_3}x_{b_4}x_{g_1}x_{g_2}x_{g_3}x_2x_4x_6$$

Transaction reliability (TR) can be defined as the conditional probability that at least one of these trees is operational. The literature is rich with algorithms to evaluate this probability, and if we apply the Syrel algorithm,[6] TR can be given as

$$\mathrm{TR} = r_{b_1}r_{b_2}r_{b_3}r_{b_4}r_{g_1}r_{g_2}r_{g_3}[r_1r_3r_5 + r_1r_3r_6q_5 + r_1r_4r_5q_3 + r_1r_4r_6q_3q_5 + r_2r_3r_5q_1 + r_2r_3r_6q_1q_5 + r_2r_4r_5q_1q_3 + r_2r_4r_6q_1q_3q_5]$$

where q_i denotes the unreliability of node i and is equal to $(1 - r_i)$.

A transaction's reliability can be increased by introducing redundancy so that its actions can be executed on several processors. Redundant execution of actions can be performed on processors located at remote nodes, all at one node, or a combination of these two cases. For the network shown in Figure 10, the transaction reliability is analyzed for the following three cases:

Case 1: Execution of redundant actions at remote nodes. In this case, each node has only one processor and the actions are executed on remote nodes. Concurrency control and redundancy management are complicated because of the remote distribution of the redundant computations.

Case 2: Execution of redundant actions at local nodes. In this case, each node has four redundant processors that can concurrently execute an action of T. Since all of the redundant computations run on the processors of the same node, concurrency control and redundancy management are simplified significantly.

Case 3: Execution of redundant actions on remote redundant nodes. This is a combination of the first two cases.

Figure 11 shows the transaction reliability for these three cases. Note that the transaction reliability for the second case is better than that of case 1. However, case 2 has twice as many redundant processors as case 1. Furthermore, there is no significant improvement in reliability for case 3 over case 2 in spite of the fact that case 3 uses twice the redundancy of case 2. Moreover, the algorithms needed to achieve concurrency control and redundancy management in case 3 are more complicated than those of case 2 because the redundant actions run on both local and remote nodes.

From this analysis, we can conclude that replicating the computations locally represents a cost-effective solution that maximizes reliability and also sim-

plifies the algorithms required to achieve recovery and consistency control. In FTODS, the computing modules are designed to provide architectural support to run transactions in a configuration similar to that described in case 2.

The computing modules of the proposed FTODS support an efficient implementation of fault-tolerant algorithms. The use of static redundancy within each node guarantees fault tolerance and reliable execution of critical tasks. Furthermore, the use of local diagnostic routines to identify faulty components reduces the complexity of recovery algorithms significantly, along with traffic on the communications network, since these functions are executed using the processors available at a node. In transaction-processing-based distributed systems, permanent memory is required for achieving atomic transactions. In FTODS, the permanent memory is designed as a two-level hierarchy with semiconductor technology used in the first level and magnetic technology in the second. Providing semiconductor permanent memory improves performance significantly because transactions can be committed much faster than by accessing magnetic disks. ■

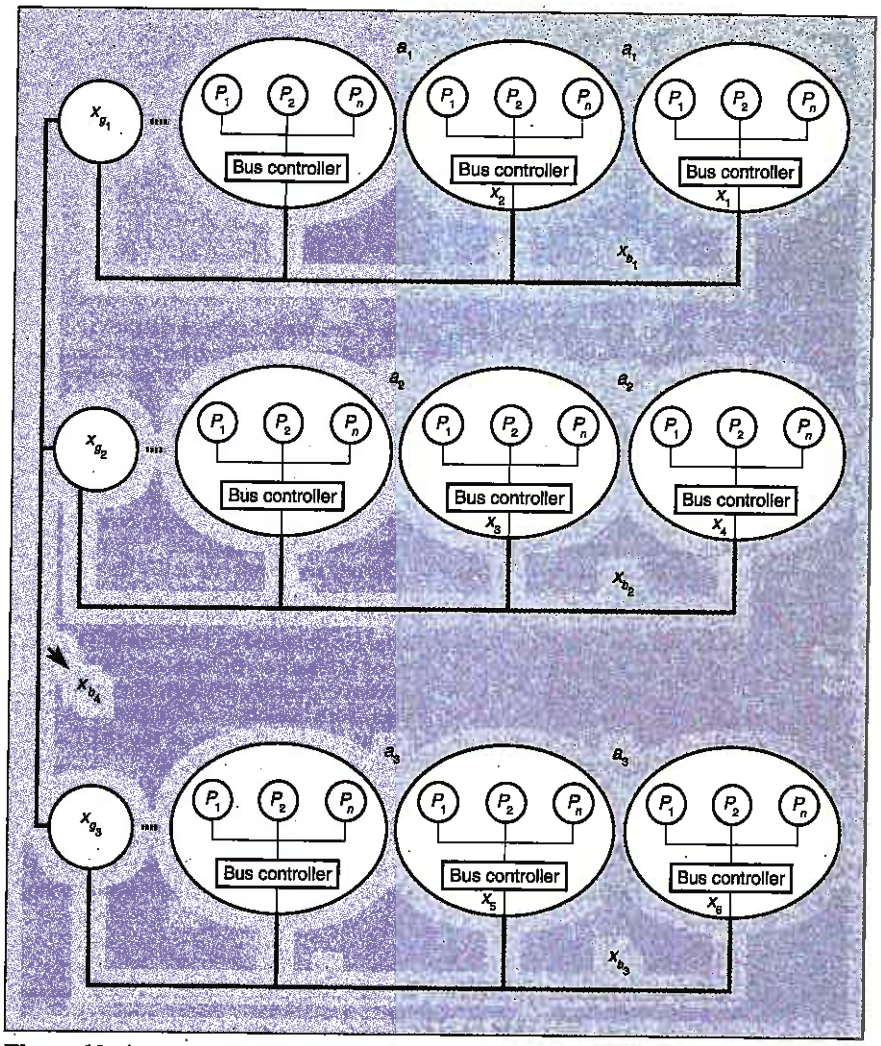

Figure 10. An example of a transaction execution.

References

1. J.A. Stankovic, "A Perspective on Distributed Computer Systems," *IEEE Trans. Computers*, Vol. C-33, No. 12, Dec. 1984, pp. 1,102-1,115.
2. V.P. Nelson, "Fault-Tolerant Computing: Fundamental Concepts," *Computer*, Vol. 23, No. 7, July 1990, pp. 19-25.
3. *Distributed Systems*, S. Mullender, ed., Addison-Wesley, Reading, Mass., 1989.
4. A.S. Tanenbaum, *Computer Networks*, Prentice Hall, Englewood Cliffs, N.J., 1988.
5. J.B. Dugan and K.S. Trivedi, "Coverage Modeling of Fault-Tolerant Systems," *IEEE Trans. Computers*, Vol. C-38, No. 6, June 1989, pp. 775-787.
6. S. Hariri and C.S. Raghavendra, "SYREL: A Symbolic Reliability Algorithm Based on Path and Cutset Methods," *IEEE Trans. Computers*, Vol. C-36, No. 10, Oct. 1987, pp. 1,224-1,232.

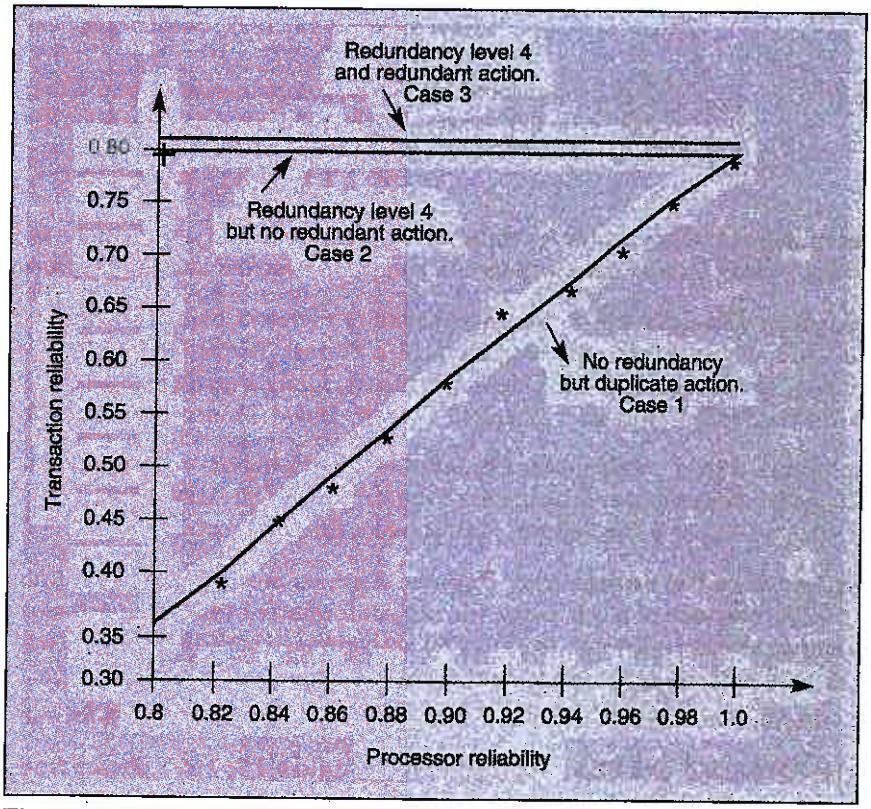

Figure 11. The effect of redundancy level on transaction reliability.

Salim Hariri is an assistant professor in the Electrical and Computer Engineering Department at Syracuse University, Syracuse, New York, and has worked and consulted at AT&T Bell Labs. His research focuses on computer architecture, distributed systems, fault-tolerant computing, and reliability and performance analysis of parallel and distributed systems. He is the program chair for the First International Symposium on High-Performance Distributed Computing (HPDC 1), scheduled for September 9-10.

Hariri received a BSEE, with distinction, from Damascus University, Damascus, Syria, in 1977; an MSc from Ohio State University, Columbus, Ohio, in 1982; and a PhD in computer engineering from the University of Southern California. He is a member of the IEEE, the IEEE Computer Society, and the ACM.

Alok Choudhary has been on the faculty of the Department of Electrical and Computer Engineering at Syracuse University since 1989. His research interests include parallel computer architectures, software development environments for parallel computers, and computer vision. He was a guest editor for the February 1992 issue of *Computer* on parallel processing for computer vision and image understanding.

Choudhary received his BE in electrical and electronics engineering from the Birla Institute of Technology and Science, Pilani, India. He obtained an MS from the University of Massachusetts, Amherst, and a PhD from the University of Illinois, Urbana-Champaign, both in electrical and computer engineering. He is a member of the IEEE Computer Society and the ACM.

Behcet Sarikaya is on the faculty of Bilkent University, Ankara, Turkey, and was previously with the Department of Computer Science and Operations Research at the University of Montreal. His research interests include all aspects of conformance testing and high-speed networks. He has authored numerous published papers on communication protocols and served on the program committees of three protocol conferences.

Sarikaya received a BSEE, with honors, and an MSc in computer science from the Middle East Technical University, Ankara, Turkey, in 1973 and 1976, respectively, and a PhD in computer science from McGill University, Montreal, in 1984. He is a senior member of the IEEE, and a member of the IEEE Computer Society and the ACM.

Hariri and Choudhary can be contacted at Syracuse University, Department of Electrical and Computer Engineering, Syracuse, NY 13244-4100; e-mail hariri@cat.syr.edu or choudhar@cat.syr.edu. Sarikaya's address is Bilkent University, Department of Computer Engineering and Information Sciences, Bilkent, Ankara 06533, Turkey; e-mail sarikaya%trbilun.bitnet@cunyvm.cuny.edu.