

AD-A245 768



2

NAVAL POSTGRADUATE SCHOOL

Monterey, California



DTIC
ELECTE
FEB 12 1992
S D D

THESIS

FUNCTION ALLOCATION IN
A ROBUST DISTRIBUTED
REAL-TIME ENVIRONMENT

by

Karen Kay Lehman

December 1991

Thesis Advisor :
Co-Advisor :

Shridhar B. Shukla
Chyan Yang

Approved for public release; distribution is unlimited

92-03497



92 2 11 101

Unclassified

Security Classification of this page

REPORT DOCUMENTATION PAGE

1a. Report Security Classification UNCLASSIFIED			1b. Restrictive Markings	
2a. Security Classification Authority			3. Distribution Availability of Report Approved for public release; distribution is unlimited.	
2b. Declassification/Downgrading Schedule				
4. Performing Organization Report Number(s)			5. Monitoring Organization Report Number(s)	
6a. Name of Performing Organization Naval Postgraduate School		6b. Office Symbol (if applicable) EC	7a. Name of Monitoring Organization Naval Postgraduate School	
6c. Address (City, State, and ZIP Code) Monterey, CA 93943-5000			7b. Address (City, State, and ZIP Code) Monterey, CA 93943-5000	
8a. Name of Funding/Sponsoring Organization		8b. Office Symbol (if applicable)	9. Procurement Instrument Identification Number	
8c. Address (City, State, and ZIP Code)			10. Source of Funding Numbers	
			Program Element Number	Project No.
			Task No.	Work Unit Accession No.
11. Title (Include Security Classification) Function Allocation in a Robust Distributed Real-Time Environment				
12. Personal Author(s) Lehman, Karen Kay				
13a. Type of Report Master's Thesis		13b. Time Covered From _____ To _____	14. Date of Report (Year, Month, Day) December 1991	
15. Page Count 109				
16. Supplementary Notation The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the United States Government.				
17. Cosati Codes			18. Subject Terms (Continue on reverse if necessary and identify by block number)	
Field	Group	Subgroup	Node Failure/Repair, Transparency, Distributed Real-Time, Migration, Static/Dynamic Allocation,	
19. Abstract (Continue on reverse if necessary and identify by block number) Critical real-time computing systems are characterized by a stringent set of reliability and performance requirements. Distributed systems, often defined to encompass a broad class of loosely coupled computer systems, are an effective means of achieving reliability and increasing system throughput. Among the many desirable characteristics that can be achieved at the application level using such a system are dynamic response to changing processing loads of functions (tasks) and exploitation of inherent parallelism using distribution. In these systems, functions must be assigned and scheduled in an attempt to be completed prior to their deadlines. Initial assignment of functions to processors (nodes) must not preclude their subsequent dynamic reassignment/reconfiguration in response to load changes or failure/repair. These allocation and reconfiguration methodologies are as diverse as their applications. A technique to manage the complexity of building such a system is a layered architecture with reconfiguration accomplished by an individual layer of software.				
20. Distribution/Availability of Abstract ☒ unclassified/unlimited ☐ same as report ☐ DTIC users			21. Abstract Security Classification UNCLASSIFIED	
22a. Name of Responsible Individual Shridhar B. Shukla			22b. Telephone (Include Area Code) (408) 646-2764	22c. Office Symbol EC/Sh

19. ABSTRACT Continued:

This thesis investigates allocation and reconfiguration algorithms. The proposed scheme for initial allocation is based on load balancing utilizing estimated execution times of the functions. The approach with respect to reconfiguration, simulated using concurrent Ada processing for a four node distributed system, is based on globally ordered broadcast communications between functions of the application program.

Approved for public release; distribution is unlimited

Function Allocation in a Robust Distributed Real-Time Environment

by

Karen K. Lehman
Lieutenant, USN
B.S.C.S., Pennsylvania State University, 1983

Submitted in partial fulfillment of the
requirements for the degree of

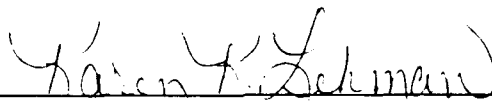
MASTER OF SCIENCE IN ELECTRICAL ENGINEERING

from the

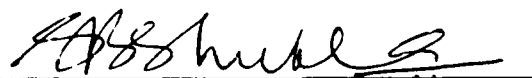
NAVAL POSTGRADUATE SCHOOL

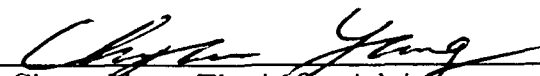
December 1991

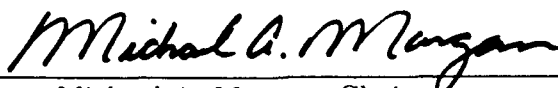
Author:


Karen K. Lehman

Approved by:


Shridhar B. Shukla, Thesis Advisor


Chyan Yang, Thesis Co-Advisor


Michael A. Morgan, Chairman
Department of Electrical and Computer Engineering

Accession For	
NTIS CRA&I	☒
DTIC TAB	☐
Unannounced	☐
Justification	
By	
Distribution/	
Availability Codes	
Dist	Avail and/or Special
A-1	



ABSTRACT

Critical real-time computing systems are characterized by a stringent set of reliability and performance requirements. Distributed systems, often defined to encompass a broad class of loosely coupled computer systems, are an effective means of achieving reliability and increasing system throughput. Among the many desirable characteristics that can be achieved at the application level using such a system are dynamic response to changing processing loads of functions (tasks) and exploitation of inherent parallelism using distribution. In these systems, functions must be assigned and scheduled in an attempt to be completed prior to their deadlines. Initial assignment of functions to processors (nodes) must not preclude their subsequent dynamic reassignment/reconfiguration in response to load changes or failure/repair. These allocation and reconfiguration methodologies are as diverse as their applications. A technique to manage the complexity of building such a system is a layered architecture with reconfiguration accomplished by an individual layer of software.

This thesis investigates allocation and reconfiguration algorithms. The proposed scheme for initial allocation is based on load balancing utilizing estimated execution times of the functions. The approach with respect to reconfiguration, simulated using concurrent Ada processing for a four node distributed system, is based on globally ordered broadcast communications between functions of the application program.

TABLE OF CONTENTS

I.	INTRODUCTION	1
	A. GENERAL	1
	B. AIM OF THE STUDY	1
	C. METHOD OF APPROACH	3
	D. ORGANIZATION	5
II.	ISSUES IN ROBUST DISTRIBUTED SYSTEMS	6
	A. GENERAL	6
	B. ALLOCATION	6
	C. OTHER RECONFIGURATION FACTORS	7
	D. NODE STATUS TABLE	8
	1. Common Section	9
	2. Unique Section	10
	3. Node Identification	11
	4. Other Variables	11
	E. SUMMARY	12
III.	FUNCTION ALLOCATION	13
	A. GENERAL	13
	B. STATIC ALLOCATION	13
	1. Initial Function Allocation Example	15
	C. DYNAMIC ALLOCATION	18
	1. Approaches	18
IV.	RECONFIGURATION FRAMEWORK	22
	A. GENERAL	22

B.	FUNCTION OFF RECEIVED	22
C.	FUNCTION ON RECEIVED	23
D.	NODE OVERLOAD	23
E.	NODE FAILURE	25
F.	NODE RECOVERY	25
V.	RECONFIGURATION ALGORITHMS	28
A.	GENERAL	28
B.	FUNCTION OFF MESSAGE PROCESSING	28
C.	FUNCTION ON MESSAGE PROCESSING	28
D.	NODE OVERLOAD PROCESSING	30
E.	NODE FAILURE PROCESSING	33
F.	NODE RECOVERY PROCESSING	34
VI.	SIMULATION RESULTS AND PROGRAM SPECIFICATIONS	37
A.	GENERAL	37
B.	SUPPORTING TASKS	37
C.	RECONFIGURATION COMPONENTS	38
D.	RECONFIGURATION RESULTS	39
E.	LIMITATIONS AND FUTURE WORK	41
F.	SUMMARY	43
VII.	CONCLUSION	44
A.	GENERAL	44
B.	STATIC ALLOCATION	44
C.	DYNAMIC ALLOCATION	44
D.	PROCESSING OF ALGORITHMS	45
E.	SUMMARY	46
APPENDIX A:	SIMULATION CODE	47

APPENDIX B: SIMULATION OUTPUT	90
REFERENCES	96
INITIAL DISTRIBUTION LIST	97

LIST OF TABLES

3.1	Variables Describing Function Characteristics	16
3.2	Ordered List Of Functions To Be Assigned	17
3.3	Function Assignment To Balance The Load	18
4.1	Example Of Overload On Node 1	24
4.2	Reconfiguration After Node Overload	25
4.3	Node Configuration After Node Failure	26
4.4	Node Configuration After Node Recovery	27

LIST OF FIGURES

1.1	A Loosely Coupled Distributed System	2
1.2	Software Layer Configuration at Each Node	4
2.1	Node Status Table	9
3.1	Function Allocation Algorithm	21
5.1	Processing of a Fnoff at a Node	29
5.2	Processing of Fnon at a Node	31
5.3	Node Overload High-Level Description Algorithm	33
5.4	Node Failure High-Level Description Algorithm	35
5.5	Node Recovery High-Level Description Algorithm	36
6.1	Reconfiguration Events upon Node Overload	40
6.2	Reconfiguration Events upon Node Failure and Recovery	42

I. INTRODUCTION

A. GENERAL

One of the goals of distributed systems is to achieve fast and efficient reconfiguration. Distributed systems are systems in which multiple nodes are working together in the solution of a single problem. The fundamental characteristic of a distributed system is its ability to map individual logical functions of an application program onto many physical nodes. These dynamically relocatable functions are to perform consistently without regard to their physical location. Additionally, the system should be able to withstand a reasonable number of node overloads, failures, and recoveries without severe degradation of the system throughput. Therefore, the reconfiguration algorithms should be fast and efficient to prevent system degradation as much as possible. The resulting dynamic reassignment of functions must attempt to minimize communication as well as maintain a balanced load among the nodes. Minimization of recovery time is also desirable.

The proposed approach to minimize recovery time and system degradation is to replicate code at each node. Replication of code minimizes the overhead required when transferring functions during reconfiguration. It also speeds up the recovery time which, in turn, alleviates system degradation.

B. AIM OF THE STUDY

Design of a framework necessary to support the (re)configuration of a robust, real-time distributed system is the objective of this thesis. An application is partitioned into multiple functions that are distributed among the nodes. Robustness is achieved by duplication of the function code at each node; however, a function is

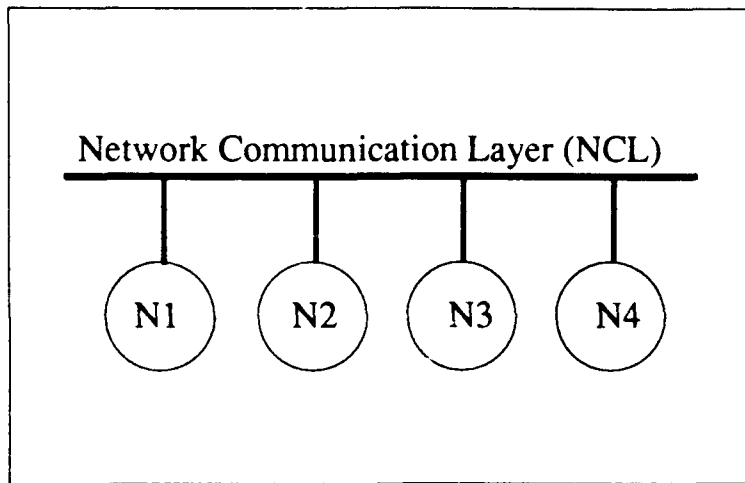


Figure 1.1: A Loosely Coupled Distributed System

active/executing at only one node at any time. The scope of this thesis is to provide a solution to the initial distribution of multiple functions onto the nodes comprising a system as shown in Figure 1.1, which is reproduced from another paper [Ref. 1]. This figure shows that a low-level communication software, called the Network Communication Layer (NCL), connects all the nodes so that a reliable global ordering of messages transmitted and received is seen by all the nodes. Reconfiguration of functions is necessitated in the event of node overload, failure, and recovery by migrating functions.

To achieve both initial allocation and reconfiguration, fast and efficient algorithms are to be provided which attempt to maximize system performance as much as possible. Two factors that are utilized in maximizing the performance are load and intermodular (function) communication (IMC). The load defines how much of a node's processor time is scheduled for function processing. IMC is the function-to-function communication required to facilitate execution of the program logically distributed among several nodes. These two factors are conflicting because an evenly distributed load among the nodes is desired in conjunction with low IMC. However, by evenly distributing the load, IMC tends to increase because two functions with high

mutual IMC may possibly get assigned to two different nodes. The key issue is to balance these factors in the algorithms used to implement allocation and reconfiguration procedures.

C. METHOD OF APPROACH

The emphasis of this thesis is to provide allocation and reconfiguration algorithms for a loosely coupled distributed system. Each node of this system contains three individual layers which maintain their own functionality. The proposed node configuration in terms of the functionality of each layer and their interrelationship is shown in Figure 1.2 which is reproduced from another paper [Ref. 1]. The Application Layer (AL), Location Invariant Function-to-Function Communication Layer (LIFFCL), and the Reconfiguration Layer (RL) operate concurrently and interface appropriately to maintain communication between dynamically relocatable distributed functions. The arcs with only one arrowhead indicate that the particular component has a one-way communication with other components that are pointed to. Likewise, arrowheads at each end of an arc indicates two-way communication. The heavy arrow and lines connecting all components of RL to Output Server (OS) designates that these components of RL must communicate with OS.

The AL manages the processing of the active functions of a node. It must communicate with both the LIFFCL and RL. AL functionality is to be specified by a follow-on thesis. The LIFFCL manages the incoming and outgoing messages of a node. Additionally, checkpointing and the generation of health messages is processed in this layer. The details of the specific components of LIFFCL can be found in another thesis [Ref. 2]. The RL handles the assignment of functions for reconfiguration and is the main emphasis of this thesis. Details of this layer are described in depth later in this thesis.

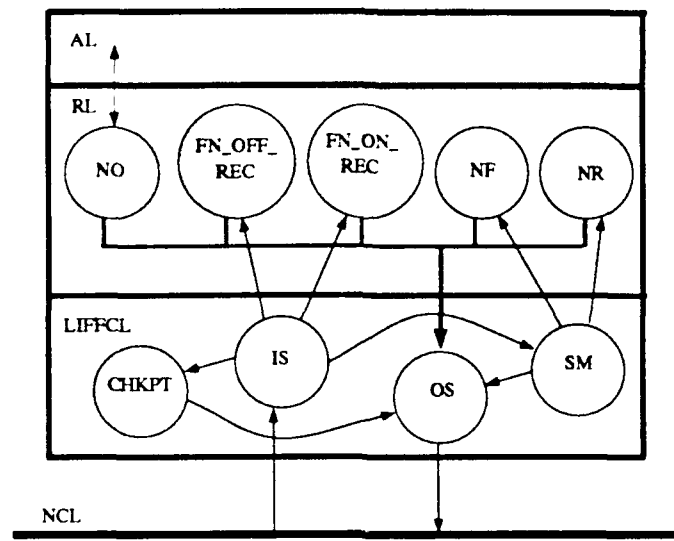


Figure 1.2: Software Layer Configuration at Each Node

Function allocation can be either static, dynamic, or a combination of both. Static allocation tends to reduce/eliminate the run-time execution overhead inherent in dynamic allocation [Ref. 3]. Initial static function allocation assumes complete *a priori* knowledge of all functions and is designed to evenly distribute the load among system nodes [Ref. 4]. Dynamic allocation is more complex than static, but the allocation result tends to be better because it can adapt to actual program execution. In static allocation, the *a priori* attributes of the functions are at best approximations. For dynamic allocation, it has been shown that more efficient algorithms perform almost as well as the more complex algorithms. Therefore, it is more cost effective to use a simple initial function assignment followed by dynamic reconfiguration when necessary. The approach that is taken is to use static allocation only for initial assignment. Dynamic allocation is utilized for reconfiguration due to node overload, node failure, and node recovery. Utilizing both methods enables the factors of load and IMC to be balanced. [Ref. 5]

D. ORGANIZATION

This thesis is organized as follows. Chapter II discusses the issues of a distributed system and the supporting resources necessary to achieve these objectives of transparency with respect to changes in node status and efficient management of the processing resources. Chapter III explains the different function allocation algorithms and describes the algorithms proposed for implementation. The framework necessary to implement the reconfiguration algorithms is explained in Chapter IV. Chapter V contains the detailed description of state diagrams and the high-level description of these reconfiguration algorithms. An overview of the implementation software and the simulation results are contained in Chapter VI. Chapter VII contains the conclusion.

II. ISSUES IN ROBUST DISTRIBUTED SYSTEMS

A. GENERAL

As mentioned in Chapter I, a distributed system consists of multiple functions logically distributed onto many physical nodes. This pattern of distribution defines the system software configuration. Numerous factors influence the development of this configuration, which directly impacts the throughput or performance of the system. Some of the more obvious factors include: function allocation, reconfiguration, maintaining the state and status of all system functions and nodes, as well as the ordering of function events (messages). In critical real-time systems, timely completion of all functions is paramount and requires fast and efficient allocation and reconfiguration algorithms. Allocation and reconfiguration algorithms require a globally consistent description of the system state upon which to base relocation decisions. To properly support the algorithms that determine the instantaneous configuration of the system, all nodes are required to have access to the same system state information. Since no resources are shared between the nodes, each node must retain its own copy of all system state information. What follows is the characterization of how and why this system is configured as it is and what factors play a role in this configuration. The sections which cover the maintenance of the global state of the system are covered in detail in another thesis but are briefly discussed below [Ref. 2].

B. ALLOCATION

Allocation algorithms determine the node where functions are executed. In determining a function's location, consideration must be given to the function-to-

function communication (IMC). An associated cost is the interprocessor communication (IPC) which is a function of IMC. IPC is caused by the processing overhead incurred when software functions resident on different nodes must communicate. Assigning two functions with high mutual IMC to different nodes increases IPC, whereas collocation reduces their IPC to zero. However, two functions with low mutual IMC assigned to different nodes may in fact speed up overall processing time. IMC is an important consideration when determining which function is to be migrated and which node receives it. Functions' attributes must be known at compile-time so that distribution among nodes can be completed effectively. This *a priori* information tells the function's priority, how long it needs for processing (execution time), periodicity, and its deadline time. Normally, if a function cannot be completed by its deadline time, it is transferred to a node that can complete it. Function migration requires current system state information in order to prevent degradation of the system. A function information array is utilized to store these attributes as well as other characteristics based on the processing time of a particular function. [Ref. 6]

C. OTHER RECONFIGURATION FACTORS

Other factors that determine how reconfiguration takes place is the maintenance of function statistics, a node's status and load, and the routing of messages. Maintenance of a function's state allows for ease of transportability and minimizes the communications required for this migration. The node status indicates which nodes are active. The reconfiguration algorithms utilize this to prevent from sending a function to a non-active node. The load of the node is what indicates how much processor time is available. This load designates whether a node is underloaded, fullyloaded, or overloaded. RL also uses this in determining the most appropriate node to receive a function upon migration. Utilizing the load minimizes system degradation by pre-

venting the migration of a function to a fully or overloaded node unless absolutely necessary as in the case of a node failure. Although the RL uses the status and load of the nodes, they are updated and maintained by the LIFFCL. Routing is also maintained by the LIFFCL. It must direct *data* messages either to the AL for processing or to the non-active function queues. These queues, one for each function of the system, are utilized to hold all *data* messages sent to each particular function. The queues help minimize communications when reconfiguration is necessary due to node overload, failure, or recovery. The details of the maintenance of these four factors are described in another thesis [Ref. 2].

For reconfiguration, each node requires access to the information described above which defines the global state of the system at any point in time. For this reason, a resource called the Node Status Table (NST) is constructed at each node to contain this information. The general approach is to keep the NST consistent using the property of globally ordered broadcast messages in the network. Additionally, it is used to ensure transparency with respect to node overload, failure, and recovery.

A detailed description of the NST is listed in the following section, and a diagram showing its contents can be found in Figure 2.1 which is reproduced from another paper [Ref. 1].

D. NODE STATUS TABLE

The NST consists of three sections. The common section contains information that is utilized for reconfiguration and is common to every node. The unique section contains all the information unique to the functions that are active on each node, and the last section contains the node identification. Each node maintains two complete copies of the NST; the duplicate copy is utilized as a backup only. Variables contained in the NST, which are used by the reconfiguration algorithms, describe the health of

COMMON SECTION		
IMC FN_LOC NODE_STAT_LD		
UNIQUE SECTION		
N1	fn 1	function variables
	fn 2	.
N2	.	.
	.	.
.	.	.
	.	.
N n	.	.
	fn k	.
NODE ID		

Figure 2.1: Node Status Table

all nodes, the location of all functions, the last message received and the last message processed for each of the system functions.

1. Common Section

The common section contains *a priori* information on which allocation and reconfiguration algorithms are based in addition to the current assignment of the functions and the loading data of each node.

The IMC matrix contains static entries that are an indication of the amount of function-to-function communications. The diagonal of the matrix contains zeros indicating no self communication. The IMC is used in the overload and failure algorithms to determine which node a function migrates to.

The Node Status and Load Array (NODE_STAT_LD) contains both the status and the load. The status indicates if a node is *up* or *down*. The load is an indicator of how much excess processor time a node has. The reconfiguration algorithms use these variables to determine the most appropriate node(s) to receive function(s).

The Function Location Array (FN_LOC) indicates the node where each function is active. It changes upon node overload, failure, and recovery. This is used heavily in the algorithms for reconfiguration, particularly in determining the total IMC for a given node.

2. Unique Section

The unique section contains the latest operational statistics for each function within the system. The unique section consists of N sections (N equals the total number of nodes in the system) with each containing an array of records, one for each active function of a node. The information contained in each record comprises the characteristics and the current state of a function. The records are updated during checkpointing, recovery, function migration, and upon expiration of a function's processor time slice.

The Function Information Array (FN_INFO) contains the following attributes: priority, periodicity, execution time, and deadline time. These attributes are *a priori* information known at the time of initial function allocation. The priority is only used during initial allocation. The other three attributes are maintained by the AL and used in computing the load of a node. In addition to the *a priori* variables in FN_INFO, AL also maintains four dynamic variables for each function. Two of the variables are time to completion (TTC) and time to deadline (TTD). These change each time a function's processor time slice expires. The two other variables are maintained to keep track of the last message arriving for an active function

(LAST_MSG_REC) and the last message processed (LAST_MSG_PROC) by AL for an active function. LAST_MSG_REC and LAST_MSG_PROC are utilized to modify the queues used for storage of the messages destined for the non-active functions at a particular node. These four variables allow a function's processing to continue at the point where it left off prior to migration. Otherwise, rollback to the last checkpoint is necessary for all function migration. However, if a node fails, it's current unique section is not accessible to the new node; therefore, a rollback is necessary. Rollback and checkpointing are covered in detail in another thesis [Ref. 2].

3. Node Identification

Each node is required to have an identity. It is mainly used to distinguish the node that currently has processor time since all nodes share the same processor for the simulation of this distributed system. The complete listing of the simulation code can be found in Appendix A.

4. Other Variables

In addition to the variables mentioned above, local variables are maintained at each node to conduct recovery, checkpointing, and queue management.

Recovery variables are utilized to determine when a node is back *up*. Checkpointing variables indicate when the NST has been updated in its entirety. Queue management handles the incoming *data* messages and directs them to their proper queues. Since there is limited space in these queues, messages already processed for a given function must be deleted periodically. Management is also required for the transmission of messages to and from the LIFFCL via the NCL. The variables used for each of the three procedures mentioned above and the management of the function queues are specified in another thesis [Ref. 2].

E. SUMMARY

As indicated throughout this chapter, knowledge of the global state of a distributed system is fundamental in the implementation of reconfiguration efforts. Unnecessary communications are avoided during function migration by each node storing all message traffic. This enables the recovery time of function migration due to node overload and node failure to be minimized. Additional overhead is required to maintain the NST; however, the overhead is more than compensated for by the NST enabling the reconfiguration algorithms to be faster and more efficient. Utilizing this NST allows for transparency of reconfiguration to the end user. The NST is what maintains the global state, and without this resource, more complex algorithms with increased overhead are necessary to manage reconfiguration. Additionally, if the information is not current, the reconfiguration decisions cannot maximize the system throughput. The algorithms utilizing the NST data are covered in the following chapters and demonstrate the necessity to keep this information current.

III. FUNCTION ALLOCATION

A. GENERAL

In a loosely coupled distributed processing system, some of the function allocation considerations encompass IMC as well as load balancing. These are conflicting factors because load balancing maximizes throughput by distributing functions among nodes whereas the distribution increases the overhead incurred. Therefore, these factors must be balanced in order to achieve optimal system performance as measured by system throughput [Ref. 6]. Function allocation can be either static or dynamic. Static allocation tends to reduce/eliminate the execution overhead inherent in dynamic allocation [Ref. 3]. Static function allocation is based on estimated execution time and function priority for system initialization and designed to achieve load balancing. Dynamic reconfiguration is more complex than static allocation, but the additional overhead required is justified because the algorithms are based on the most current run-time statistics of each of the nodes. Reconfiguration due to node failure, node overload, or node recovery utilizes dynamic allocation and reflects time constraint considerations as well as IPC/IMC costs. The cost of IPC is measured in terms of the amount of data transferred among functions assigned to different processors [Ref. 7].

B. STATIC ALLOCATION

Static function allocation methods utilize each function's attributes to distribute the functions among the nodes accordingly. One proposal for static allocation is the Heavy Node First (HNF) algorithm. In using the HNF algorithm, the program must be representable by a Directed Acyclic Graph (DAG). A node of a DAG represents an

operation, and the directed edge represents the precedence among the nodes. Each node is associated with a total computation weight. Utilizing a DAG requires that the program behavior be predictable. However, constructs such as loops and conditionals introduce uncertainty in program behavior. This prevents using DAGs to represent a program. HNF attempts to maximize the number of parallel operations and/or minimize the execution delays. This algorithm keeps track of total weight at each node. HNF first assigns the functions in descending order of weight until all functions are assigned. The algorithm continually keeps track of the nodes with the maximum total weight and the minimum weight, as well as the weight of the next function to be assigned and its successor in order to determine which node gets the next function. It was from this HNF algorithm that the Light Node First (LNF) algorithm was developed. [Ref. 3]

LNF encompasses major ideas from HNF, but it utilizes them in a different fashion. Accomplishment of this assignment algorithm requires functions to be available in descending order of estimated execution time (weight). Functions are allocated sequentially to nodes in node number order. This static allocation procedure ensures distribution of functions to achieve a balance of execution time on each node. Since execution time is the "balancing factor", no single node is excessively loaded. Additionally, once all functions are assigned, the priority number of each function is utilized to obtain the chronological order of execution of the functions assigned to a given node. This priority number is determined at the time a program is partitioned into functions. Since a program may have precedences, the function-to-function dependency becomes important. As an example, say function *A* was dependent on a variable generated from function *B*, and both *A* and *B* are active on the same node. Obviously, *B* must get processor time first in order to pass the correct value of the variable to *A*. Therefore, *B* has a higher priority than *A*. It's the logical ordering of

the program and the sequence of events which designates the priority number.

Static allocation methods have inherent shortfalls. In order to implement static methods, all estimated run-time knowledge of the function must be known at compile time [Ref. 8]. The HNF algorithm's major shortfall is the fact that a DAG is required i.e., not all programs are predictable and thus cannot be represented by a DAG. A shortfall of the LNF method is that IPC cost is ignored and can be very high. However, this situation tends to balance itself out after system initialization, because relocation of functions due to node overload and node failure are based on the IPC/IMC costs.

1. Initial Function Allocation Example

The following is a example of the workings of the function allocation. Assumptions are made that the functions to be allocated are available in an array ordered by execution time along with the attributes that are contained in Table 3.1.

The static items are located in the FN.INFO array of each record for a function. The dynamic items are updated by the AL and stored in the same record as the FN.INFO array. The initial function allocation algorithm goes through the list of functions and assign them according to the weights. Allocation only uses the ET and the priority. The remaining variables are used by the AL to determine the load and by the RL when it is necessary for function migration. In addition, a priority number is assigned to each function. This priority number, as explained previously, designates the function order of processing after static allocation only.

The functions are provided in descending order by weight upon initial assignment. For $i=1$ to N , the i th function from the ordered array is assigned to the i th node i.e., the first function in the list is activated at *Node 1*. After each node has received one function, the next N successive functions are assigned to the nodes in reverse order i.e., from *Node N* back down to *Node 1*. This process continues in a snake-like fashion until all functions are assigned. Once all functions are assigned,

TABLE 3.1: VARIABLES DESCRIBING FUNCTION CHARACTERISTICS

<i>Function Attributes</i>	<i>Description</i>
Priority	used for static allocation only to order the function processing
Periodicity	reflects how often a function reoccurs (static)
Execution Time (ET)	indicates the estimated time needed to process a function (static)
Deadline Time (DL)	estimated time a function needs for processing (slightly larger than the ET (static))
Time To Completion (TTC)	initially equal to ET but changes when a function gets processing time (dynamic)
Time To Deadline (TTD)	initially equal to DL but changes when a function gets processing time (dynamic)

each node's functions are ordered in ascending order by priority number. The results of this allocation procedure are equivalent to the Least Node First (LNF) algorithm.

The algorithm is executed at all nodes since at this time there is no other processing going on. No interruptions are necessary at the other nodes because there is no need of sending communications to each node as they can set up their own arrays of the NST at that time. Each time a function is determined to go to a particular node, the nodes can update the $k \times 1$ FN_LOC array in the NST (k equals the total number of functions).

If only one node does the algorithm, it has to send at least k communication messages telling all nodes who gets each function. In order to have flexibility, each node processes the algorithm because the one designated to determine this may be *down*.

An example of the ordering of functions based on their execution time is shown in Table 3.2. The actual assignment of the functions listed in Table 3.2 is depicted in Table 3.3 with the functions in order by priority number. The total load (execution time) designated in Table 3.3 shows that nodes are balanced fairly evenly. The high-level description algorithm that implements this static allocation is shown in Figure 3.1.

TABLE 3.2: ORDERED LIST OF FUNCTIONS TO BE ASSIGNED

<i>Functions</i>	<i>Execution Time</i>	<i>Priority</i>
2	100	6
5	80	3
4	60	2
3	40	1
1	35	4
6	15	5

TABLE 3.3: FUNCTION ASSIGNMENT TO BALANCE THE LOAD

<i>Node No.</i>	<i>Assigned Functions</i>	<i>Total Execution Time (Load)</i>
1	6,2	115
2	5,1	115
3	3,4	100

C. DYNAMIC ALLOCATION

In order to balance the conflicting factors influencing allocation, dynamic allocation is utilized after initial static allocation has occurred. Dynamic reconfiguration due to node overload, node failure, or node recovery reflects time constraint considerations as well as IPC/IMC costs.

1. Approaches

When it is determined that a function cannot be completed by its deadline time (node overload), when a node fails, or when a node recovers, a smooth transition/migration of the functions(s) must take place. This migration is to be transparent to the user. Most of the existing allocation models have considered the minimization of the IPC. The objective of Chu's allocation method is to minimize the maximum processor loading [Ref. 9]. Other attempts have been made to include message transmission delays in the communication costs [Ref. 10]. Some of the more popular methods of reconfiguration are as follows: load sharing (LS), sender initiated (SI), focused addressing, bidding addressing, random scheduling, and flexible which is a combination of both focused and bidding addressing. All these methods, except random scheduling, require each processor to maintain state information from other processors. Therefore, an efficient means of collecting and updating state information, independent of normal IPC, must be developed. LS is composed of the transfer

and location policy which is either initiated by its own processor (source) or by the other processors (servers) [Ref. 11]. When LS is initiated by the source, it is similar to the SI method. Both of these methods, LS and SI, are very similar to focused and/or bidding algorithms.

The dynamic reconfiguration algorithms proposed are a type of focused addressing. The algorithms assign the function(s) to the node(s) capable of completing the function(s) by the deadline time. Additionally, the IMC as well as the node's load status are utilized in determining migration. Duplication of code for all functions at each node and checkpointing both help in minimizing the overhead that is independent of normal IPC/IMC.

Problems associated with the methods mentioned are listed below. For the LS method, further investigation is required because no analytical formula is derived to determine the probability of a function completing by its deadline. In addition, the queue length isn't sufficient in determining each node's load [Ref. 11]. Consideration of cumulative execution times of the functions must take place. Random scheduling requires low overhead in determining the receiving node, but it can easily send the function to an overloaded node. This is due to its randomness and the fact that it doesn't utilize the system state information. Bidding and focused addressing incur more overhead than random scheduling. Focused addressing requires less communication than bidding, but the data utilized is not as current as that used in bidding. The flexible addressing attempts to reap the benefits of both bidding and focused addressing. It has been shown, however, that focused addressing's performance is more stable and consistent regardless of the load. It is for this reason that focused addressing is used in the approach to handle the cases of node overload, failure, and recovery. The problem of data not being current, as mentioned above, can be alleviated in focused addressing through the additional use of *status* messages that are

sent periodically to keep load information current. The RL framework encapsulates the components required to accomplish this reconfiguration and is explained in the following chapter. [Ref. 12]

```

/* Procedure for Static Allocation: assigns all functions */
/* of array fn(f),the array is assumed to be available at */
/* start-up time, in a snake-like fashion based on execution */
/* time and orders them by the priority number */

task body ALLOCATION is
  n : integer := 1;
  f : integer := 1;
  rev : boolean := false;
begin
  GET fn(f) from input array
  while fn(f)  $\neq$  0 loop      -- assign all the functions
    put node n in FN_LOC(fn(f))    -- in NST
    increment n if not rev;
    decrement n if rev;
    if n = 4 then
      rev := true;      -- reverse direction of assignment
    end if;
    if n = 1 then
      rev := false;     -- forward direction of assignment
    end if;
    f = f + 1;
    GET fn(f);
  end loop;
  if active function at node then
    order functions by priority    -- send to A/L in order of
                                   --priority
  end if;
end ALLOCATION;

```

Figure 3.1: Function Allocation Algorithm

IV. RECONFIGURATION FRAMEWORK

A. GENERAL

RL's objective is to accomplish transparent reconfiguration whenever necessary. Utilizing bi-directional interfaces with AL and LIFFCL, RL receives *fnoff* and *fnon* messages from Input Server, notification of node failure from Status Monitor, and notification of node overload from AL. As mentioned previously, the components of the LIFFCL are covered in another thesis i.e., Input Server and Status Monitor [Ref. 2]. Once notified, the RL utilizes information in the NST to make its reconfiguration decisions. The specific components of RL are Function Off Received (FN_OFF_REC), Function On Received (FN_ON_REC), Node Overload (NO), Node Failure (NF), and Node Recovery (NR). Each component is discussed in detail in the following sections.

B. FUNCTION OFF RECEIVED

FN_OFF_REC's primary purpose is to process *fnoff* messages. *Fnoff* messages result when a node detects overload at its site. Upon receipt of a *fnoff* message, RLs at the nodes determine if they are designated to receive the function. This determination is made by checking a destination field of the *fnoff* message. In turn, only the activating node builds and sends a *fnon* message to the LIFFCL. No additional communications are required to retransmit *data* messages for the migrating function since the last checkpoint because each node saves these messages in queues. However, management of these queues is required.

C. FUNCTION ON RECEIVED

FN_ON_REC's primary purpose is to process a *fnon* message. A *fnon* message is generated for two conditions. One is in response to the *fnoff* message described above, and the other condition is when a node recovers from failure. A *fnon* message is broadcasted to indicate the new location of a function. It also serves as a trigger to the deactivating and activating nodes to realign the AL i.e., deleting or adding a function to the active list in AL.

D. NODE OVERLOAD

Node overload results when a node is unable to complete all assigned functions prior to their deadlines. Functions may be migrated to other nodes provided they are underloaded. Node Overload is the algorithm that generates the *fnoff* message. It determines which function is to be migrated and which node receives it. The function to be migrated is the one with the most time left to completion until its deadline. The receiving node is based on IMC and the load status in which load status takes precedence over IMC. The underloaded node with the largest IMC value accepts the migrating function. This decreases the IMC and allow all nodes to complete their functions on time. If equal IMCs exist, the node with the smallest ID activates it. The amount of IMC traffic associated with function migration as a result of node overload, failure, and recovery is reduced due to the robustness of the system i.e., duplication of function code.

An example of node overload is shown in Table 4.1. Referring to this table, *Node 1* realizes it can't meet the deadlines based on the difference between the local clock and the DL and then comparing this difference to the TTC. For example, assume the local clock to be four seconds. Starting with *function 2* of *Node 1*, subtracting the local clock from TTD gives a value of six. Comparing six to TTC shows that

there is just enough time to complete *function 2* by its deadline. Now for *function 6*, the TTD of *function 2* is subtracted from the TTD of *function 6* which gives a value of seven. However, when comparing seven to the TTC value of eight, it shows that *function 6* cannot be completed on time. Therefore, *Node 1* is overloaded and needs to try and and needs to try and migrate *function 6*.

Function 6 is migrated to a node that is underloaded. For this example, assume that *Node 2* is fully loaded and *Node 3* is underloaded. Thus, *function 6* is migrated to *Node 3*. The updated configuration is shown Table 4.2. Upon migration, a new TTD for that function is computed based on the time that the node receives the function, and an additional allowance of 1 "second" for migration time is added in.

Since storage capacity is fairly inexpensive and readily available, it is assumed that sufficient storage exists within a node to store all function-to-function traffic received between checkpoints. This is regardless of whether or not the function is actually resident/active on the node. Another alternate method requires the LIFFCL to retransmit all traffic addressed to the migrating function since the last checkpoint. This, however, increases the overhead for migration drastically.

TABLE 4.1: EXAMPLE OF OVERLOAD ON NODE 1

<i>Node No.</i>	<i>Function</i>	<i>Priority</i>	<i>DL</i>	<i>ET</i>	<i>TTC</i>	<i>TTD</i>	<i>Periodicity</i>
1	2	6	10	7	6	10	15
	6	5	16	9	8	17	0
2	5	3	8	4	2	8	0
	1	4	12	7	4	13	20
3	3	1	8	4	3	9	10
	4	2	25	11	10	28	30

TABLE 4.2: RECONFIGURATION AFTER NODE OVERLOAD

<i>Node No.</i>	<i>Function</i>	<i>Priority</i>	<i>DL</i>	<i>ET</i>	<i>TTC</i>	<i>TTD</i>	<i>Periodicity</i>
1	2	6	10	7	6	10	15
2	5	3	8	4	2	8	0
	1	4	12	7	4	13	20
3	3	1	8	4	3	9	10
	4	2	25	11	10	28	30
	6	5	16	9	8	18	16

E. NODE FAILURE

Node failure results when a node fails to transmit a periodic health message within a prescribed time limit. When a node failure occurs, all the functions executing on it must be recovered at other nodes. Upon detection of a node failure by the Status Monitor, all active nodes simultaneously begin execution of the **Node Failure** algorithm to determine which nodes have to activate the functions currently resident on the failed node. Each node determined to activate the function(s) sends a *fnon* message(s). Although the algorithm results are the same at each node, *fnon* messages are still sent to maintain system continuity.

As an example, consider the functions and assignments as described back in Table 3.3. If *Node 2* is *down*, its functions must be recovered on the other nodes. For this case of reconfiguration, assume that *Node 1* has a higher mutual IMC with *function 1*, and *Node 3* has a higher mutual IMC with *function 5*. Therefore, the reconfiguration is as shown in Table 4.3.

F. NODE RECOVERY

When a node has recovered from failure, it is to be smoothly assimilated into the workload. Upon detection of its own recovery, the node generates a message to

TABLE 4.3: NODE CONFIGURATION AFTER NODE FAILURE

<i>Node</i>	<i>Functions</i>	<i>TTC</i>
1	6,2,1	150
3	3,4,5	180

trigger other nodes to send information which enables it to rebuild the current state of the overall system. In particular, the NST is rebuilt from the information sent by other nodes. The recovered node now possesses the latest system status. When this process is completed, the node sends another message to indicate that it is *up*. From that point, **Node Recovery** determines which function to turn on at its site. Only the recovering node determines which function to activate because its NST contains the most current system status. This is accomplished through the use of a *fnon* message. Only one function is turned on at the node, and it may not necessarily be one that was previously active there. Although it only takes one function, the nodes tend to balance themselves out over time due to node overloads, etc.

As an example, consider the data listed in Table 4.3. *Node 2* is assumed to be recovered and must determine which function it activates. Additionally, *functions 2* and *5* have just completed execution and have the greatest amount of slack-time. Therefore, they are the candidates for migration. Based on NST data, *Node 2* determines that the load of *Node 3* is larger than that of *Node 1*. Thus *Node 2* activates *function 5*. The reconfiguration of this recovery is shown in Table 4.4.

TABLE 4.4: NODE CONFIGURATION AFTER NODE RECOVERY

<i>Node</i>	<i>Functions</i>	<i>TTC</i>
1	6,2,1	150
2	5	80
3	3,4	100

V. RECONFIGURATION ALGORITHMS

A. GENERAL

In order for reconfiguration to occur, bidirectional communication, particularly between LIFFCL and RL, is required. The components of LIFFCL and RL determine what the reconfiguration is and generate the messages necessary to accomplish this. What follows are the algorithms for each component within RL.

B. FUNCTION OFF MESSAGE PROCESSING

When the Input Server receives a *fnoff* message, it sends the message to RL. RL, in turn, calls the procedure `FN_OFF_REC` to process this message. Upon receipt of the *fnoff*, the procedure checks the `DEST_NODE` field. This field designates the node which has to turn the function *on*. Only the activating node takes action by building and sending the *fnon* message. The function to be migrated is listed in the `DEST_FUNC` field. Additionally, the activating node must copy the body of the *fnoff* message to the body of the *fnon* message that it creates. This is necessary because all action taken by the nodes occurs upon receipt of a *fnon* message.

A state diagram of the actions taken in the procedure is shown in Figure 5.1.

C. FUNCTION ON MESSAGE PROCESSING

When the Input Server receives a *fnon* message, it sends the message to RL. RL, in turn, calls the procedure `FN_ON_REC` to process this message. Upon receipt of the *fnon* message, each node must determine if it is the activating or deactivating node. If it is either of these two, RL must notify AL to start or terminate the function listed in `DEST_FUNC`. Additionally, the activating node must modify the unique section of

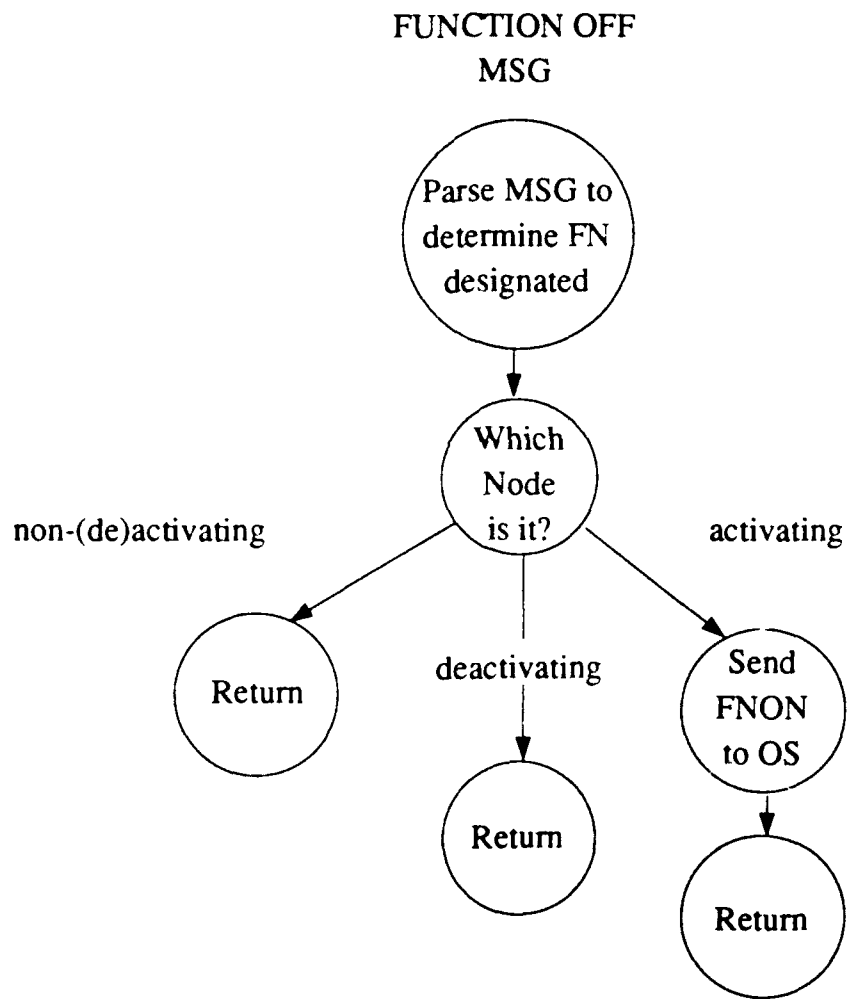


Figure 5.1: Processing of a Fnoff at a Node

NST to reflect the function information sent in the message body. This information is also passed to the AL for further action; it indicates the *data* message with which the AL is to begin processing. The function queue manager (not implemented) modifies the queue for the function based on the LAST_MSG_REC. All nodes update their NST's FN_LOC array to indicate the function's current location.

A state diagram of the actions taken in the procedure is shown in Figure 5.2.

D. NODE OVERLOAD PROCESSING

Node Overload is initiated by notification from AJ. It determines which function is to be migrated from the overloaded node. The criteria used in designating which function is moved is based on the *a priori* information about the function. The possibility exists that more than one function cannot be completed on time. Of these functions, the one with the largest slack-time i.e., largest negative time till completion, is migrated. The node to receive the function is an underloaded node with the largest IMC. If no underloaded node is available for migration, the function remains at its current node. This prevents unnecessary communications and computations of a node because it most likely becomes overloaded if a function migrates there.

Node Overload has three major functions. It creates a TRANS_NODE array which identifies the possible nodes to migrate the function to. It also determines MOVE_FN which is a function causing overload and has the largest slack-time or most negative time remaining. Lastly, it determines which node of TRANS_NODE has the largest IMC with MOVE_FN.

The TRANS_NODE array is built using the NST's NODE_STAT_LD. If the load of a node is less than .75, the node number is stored in the array. A simple loop is used to determine which active function(s) is causing overload. This determination was explained earlier and accompanied with Tables 4.1 and 4.2.

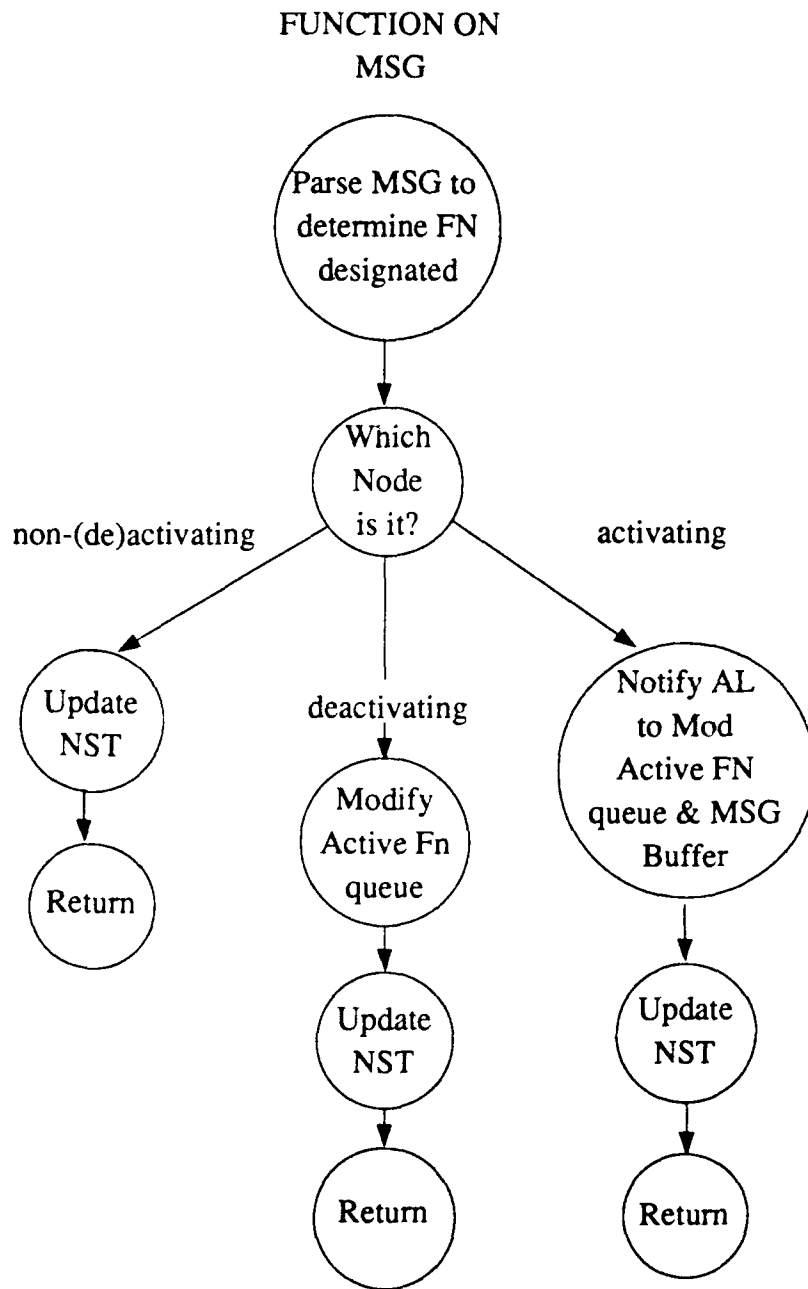


Figure 5.2: Processing of Fnon at a Node

After determination of TRANS_NODE and MOVE_FN, the IMC values are computed and stored in the NODE_IMC array. This is done for each of the nodes listed in TRANS_NODE. The largest value of NODE_IMC designates which node is to be the NEW_NODE. The node number is preserved by TRANS_NODE(*i*) corresponding to the IMC value of NODE_IMC(*i*); thus, NEW_NODE gets the value of TRANS_NODE(*i*). From this point, the *fnoff* message is built and sent to the LIFFCL. The message fields DEST_FUNC and DEST_NODE are assigned the values of MOVE_FN and NEW_NODE respectively. The message body contains the function's current unique section, thus reflecting any processing changes since the last checkpoint. The designated node to receive the migrating function transmits a *fnon* message, indicating its activation of the function. Upon receipt of the *fnon* message, all nodes update the FN_LOC array. Additionally, the receiving node has to update the function's message queue. If the overloaded node does not receive the *fnon* message prior to a timeout, the function must remain on the overloaded node until at least the next checkpoint cycle.

If no nodes are underloaded, the overloaded node must keep the function. Moving a function to a non-underloaded node creates unnecessary communications for turning the function *off* and *on* because the node turning the function *on* runs a very high risk of becoming overloaded. This then creates additional computation of **Node Overload** at the new node. The reason for moving the function with the most slack-time is that the probability of completing on time is greater than if it didn't migrate.

This algorithm can be modified slightly for the AL to use for its computation of the load of the node. This, however, is recommended for future work.

Allowing the overloaded node to singularly determine both the migrating function as well as the receiving node minimizes the interruptions of normal processing

operations. Additionally, requiring the overloaded node to ensure that the migrating function is updated with the required message traffic reduces IMC.

The high-level description algorithm of Node Overload is in Figure 5.3.

```

/* Procedure to determine the function to */
/* migrate and the node to receive it */

procedure Node Overload is
begin
  TRANS_NODE(i) := underloaded nodes;
  MOVE_FN := function with largest slack-time;
  MODE_IMC(i) := total IMC for each node of TRANS_NODE;
  determine which node of TRANS_NODE(i) has largest IMC
    in NODE_IMC(i);
  build FNOFF msg;
  Send FNOFF msg to OS;
end Node Overload;

```

Figure 5.3: Node Overload High-Level Description Algorithm

E. NODE FAILURE PROCESSING

Node Failure is processed at each node upon detection of a failure. Each node determines if it is to receive a function. If so, it generates a *fnon* message. The IMC is all that is utilized in determining if a node turns *on* a function. The load is not considered because the functions active on the failed node must be recovered. Even if a node becomes overloaded, it is necessary since the overall system has been degraded due to failure.

Each node sends *periodic* status messages to indicate that it is *up* and what its current load is. At each node, a **TIMER** array is maintained in the NST for the Status Monitor to check every so often to determine if the nodes have responded with their *periodic* status message. When Status Monitor detects that a *periodic*

message hasn't been received in time, it turns the status of that node to *off* in the `NODE_STATUS_LOAD` array. Status Monitor then passes the `NODE_ID` of the failed node to **Node Failure**. **Node Failure** must first determine which functions were assigned to the failed node. This is accomplished by searching for the `NODE_ID` in the `FN_LOC` array. Once the assigned functions are known, determination must be made as to which node(s) is to receive the migrating functions. Determining the destination node for a migrating function requires the IMC matrix to be searched. The function is assigned to the node with the functions which require the greatest amount of communication with the migrating function. If functions on different nodes have the same IMC value, the node with the smallest ID number receives the migrating function. The larger the value of an element in the IMC matrix, the more function-to-function communication required. Assignment in this fashion minimizes the amount of IMC which is time consuming in this distributed processing design. All active nodes begin execution of the **Node Failure** algorithm. The appropriate nodes build and send *fnon* messages to the LIFFCL. All nodes update their `FN_LOC` array upon receipt of the *fnon* message. The algorithm is only done in a high-level description; therefore, it is assumed that the outcome is the same at each node. Since the algorithm is based on the static IMC, it seems fair to assume the correctness of this statement. The high-level description algorithm for **Node Failure** that handles this process is found in Figure 5.4.

F. NODE RECOVERY PROCESSING

Recovery procedures are initiated by the first *periodic* status message received by the recovering node after a node restart. Upon receipt of a *periodic* message, the recovering node generates an *aperiodic* status message containing a load percentage of zero. Additionally the recovering node also sets a boolean variable `RCVRY_IN_PROGRESS`

```

/* Procedure Node Failure migrates all */
/* functions from the failed node */

procedure Node Failure (Node.Id of failed node) is
begin
  ACTIVE_FN(i) := active functions of failed node;
  for i = 1 to end of ACTIVE_FN loop
    determine node with largest IMC;
    if own node determined then
      build FNON msg;
      Send FNON msg to OS;
    end if;
  end loop;
end Node Failure;

```

Figure 5.4: Node Failure High-Level Description Algorithm

and initializes all elements of its **TIMER** array to the current time. All active nodes, upon receipt of the *aperiodic* message, generate an *aperiodic* message containing the unique and common section of its **NST**.

Each node also sets its flag, **LOC_VAR.UNIQ_SENT**, which prevents the node from repeatedly sending *aperiodic* messages when it receives the *aperiodic* messages sent by other active nodes. Upon receipt of the *aperiodic* message, the Status Monitor of the recovering node stores the body of the messages in its own **NST** and set a flag in its **RCVRY** to indicate receipt of a given node's unique and common sections. After all nodes have responded to the recovering node with an *aperiodic* message, the recovering node generates a *periodic* status message with a load of zero. Receipt of a *periodic* status message with a load percentage of zero indicates to all nodes that the recovering node has, in fact, fully recovered and is ready to commence normal processing. Therefore, all nodes reflect the recovered node as *up* in **NST** and clear their **UNIQ_SENT** flag. The recovered node notifies its **Node Recovery** algorithm for

```

/* Node Recovery takes the function on */
/* the most overloaded node with the */
/* most slack-time */

procedure Node Recovery is
begin
  OL_NODE := node with largest load;
  FN(i) := active functions of OL_NODE;
  for i = 1 to end of FN(i) loop
    MOVE_FN := FN(i) with largest time to deadline;
  end loop;
  build FNON msg;
  Send FNON msg to OS;
end Node Recovery;

```

Figure 5.5: Node Recovery High-Level Description Algorithm

determination of the function to be activated. The high-level description algorithm for **Node Recovery** is shown in Figure 5.5.

VI. SIMULATION RESULTS AND PROGRAM SPECIFICATIONS

A. GENERAL

In order to verify the framework described so far, a system of four nodes and 12 functions has been simulated. Ada is used to implement this system as a group of independent packages. Components of the LIFFCL and the RL make up each node by utilizing the instantiation of these packages. The system also contains a NCL, an Event Generator (EG) package, and the Front End Processor (FEP) procedure. Each of these packages, consist of one or more independent task bodies which contain "accept" statements for establishing rendezvous' between tasks. The periodic tasks are activated initially with a rendezvous call to establish each task's identity. After this initial rendezvous, the periodicity is established by the expiration of a delay statement. Throughout the duration of the delay, a task is suspended by the operating system, releasing the processor for utilization by other tasks.

B. SUPPORTING TASKS

The supporting tasks are those which are not instantiated for each node. Additionally, these tasks do not aid in the processing of the different messages that are broadcast.

NCL is used to simulate a broadcast network. Messages are both transmitted and received by each node's LIFFCL. Details of the sending and receipt of messages via the NCL are covered in depth in another thesis [Ref. 2].

The Event Generator either creates *fnoff* messages or simulates node failure. Additionally, these events occur periodically and are randomly chosen. The two events

created are utilized to verify the correctness of the high-level description algorithms that handle node overload or the recovery procedures that must take place for node failure and recovery. A *fnoff* message is sent to ensure that the node who is to receive a function does, in fact, respond by building and sending a *fnon* message. The *fnoff* message is normally built within the algorithm for **Node Overload**.

Because only one processor is used in simulating the four nodes which run concurrently, one node could not be physically turned *off*. Therefore, in simulating node failure, the Event Generator marks a node as *down*. The node marked *down* will not send its next *periodic* message. The other nodes' LIFFCL detects a failure when this message is not received on time. The details of generation of *periodic* messages and detection of failure is covered in another thesis [Ref. 2]. The FEP is a single procedure executed only once to initialize nodes, open files, and execute a single rendezvous call with each task in order to assign the task's node identity.

C. RECONFIGURATION COMPONENTS

The algorithms and tasks that are used for processing the different types of messages related to reconfiguration are defined as the application components. They are instantiated for each of the nodes. The basic functions of each are covered in the following paragraphs.

RL consists of two procedures, **Fn_Off_Rec** and **Fn_On_Rec**. The Input Server, a component of LIFFCL, rendezvous' with RL when processing is required for *fnoff* and *fnon* messages. For a *fnoff* message, a node checks to see whether or not it must respond with a *fnon* message. If it does, it builds and sends the *fnon* message to the LIFFCL. Unless the node is designated to receive a function, no further action is required. Upon receipt of the *fnon* message, all nodes must update the function location. In addition, the activating and deactivating nodes must make changes to

the active functions in the AL. However, this is not within the scope of the thesis and is recommended for future work.

Although the code for the actual reassignment of functions due to node overload, failure, and recovery is not complete, the events which precede and follow this action are included. As mentioned above, the Event Generator simulates a node overload. The proper action taken by all the nodes, upon receipt of the *fnoff* and *fnon* messages, must occur, and this is tested in the simulation program. Likewise, the necessary action taken upon receipt of *periodic* and *aperiodic* messages is checked for the case of node failure/recovery.

D. RECONFIGURATION RESULTS

To verify the correctness of the state diagrams and high-level description algorithms discussed in the previous chapters, timing diagrams are provided. They reflect the sequence of events that occurred in the output following the receipt of messages that were built and sent by either the Event Generator or the specific tasks for which coding is complete.

For the **Node Overload**, a *fnoff* message is sent out when a node is overloaded. In this case, the Event Generator simulates the overload by sending the *fnoff* message. The message indicates which function is to migrate and which node is to receive it. The node designated to receive it, in turn, responds with a *fnon* message. The diagram indicating the response when receiving both *fnoff* and *fnon* messages is shown in Figure 6.1. This figure shows that *Node 2* is the receiving node, and that it does respond or acknowledge with a *fnon* message. The verbiage beside the arrowheads explains the action needed to be taken at that particular node. The arrows indicate broadcasts, and the numbers beside them indicate simulation time at the time of initiating and completing each broadcast.

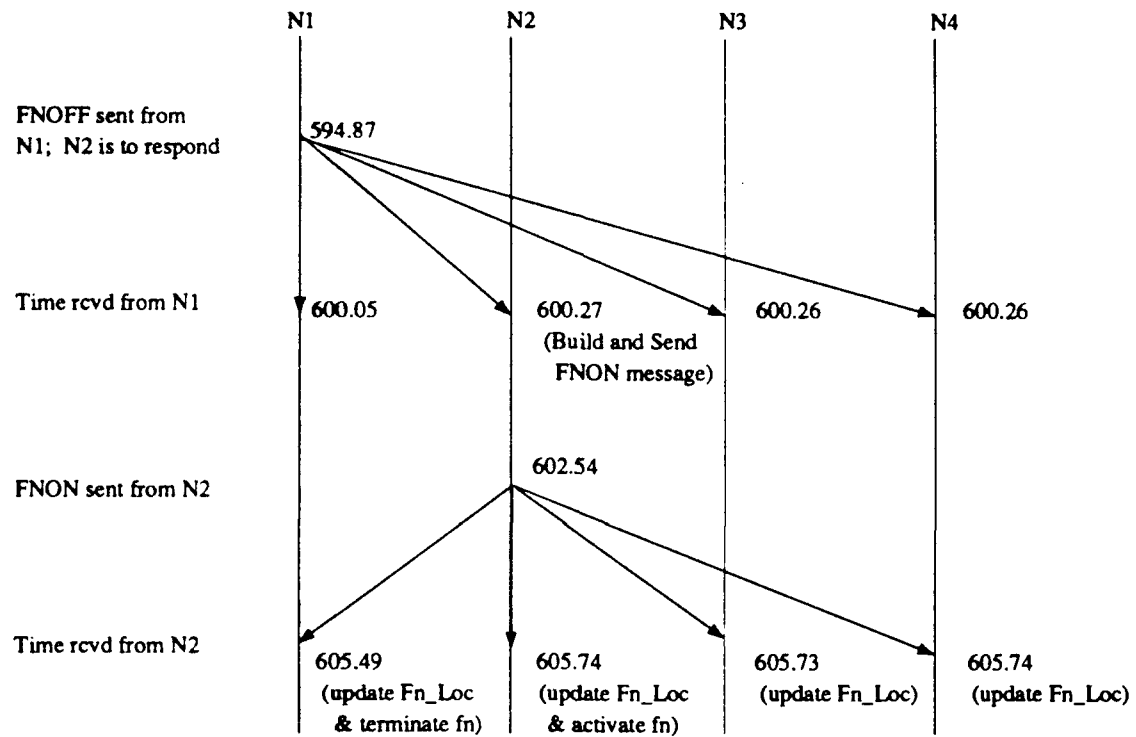


Figure 6.1: Reconfiguration Events upon Node Overload

A failed node is detected when a *periodic* status message is not received at other nodes within a certain time limit. For simulation purposes, the failed node starts a recovery process when it receives the next *periodic* status message. When all active nodes have sent their NST, recovery is complete. The recovered node responds with a *periodic* message containing zero load to let other nodes know it is back up. The **Node Recovery** algorithm is then started at the recovering node to migrate a function. The diagram of these events leading up to the actual assignment of a function is shown in Figure 6.2. This figure shows that *Node 1* failed and then began recovery procedures. Each time *Node 1* received an *aperiodic* status message from the other active nodes, it updated its NST i.e., it is rebuilding the NST. The other nodes update the load of the node which sent the message and the time that the message was sent. After *Node 1* received all the *aperiodic* messages, it built and sent the *periodic* message which indicates to all nodes that *Node 1* is back up.

E. LIMITATIONS AND FUTURE WORK

The current program is not complete at this time. The additional work required to complete the specified operation of the system is described below. The high-level description algorithms need to be converted to actual code and implemented into the existing code. The static allocation algorithm must also initialize the FN_LOC and FN_INFO arrays while the assignment of functions take place. Currently, when the RL establishes a rendezvous with the Input Server, the latter is tied up until the former is done with its processing. A circular queue implemented within RL prevents this from happening. In addition, the RL then needs to be set up with a delay statement in order for it to have a chance to check if messages need processing. The major work that needs attention is the development of AL to conform to the interface requirements specified in the thesis. AL has to determine the load of a

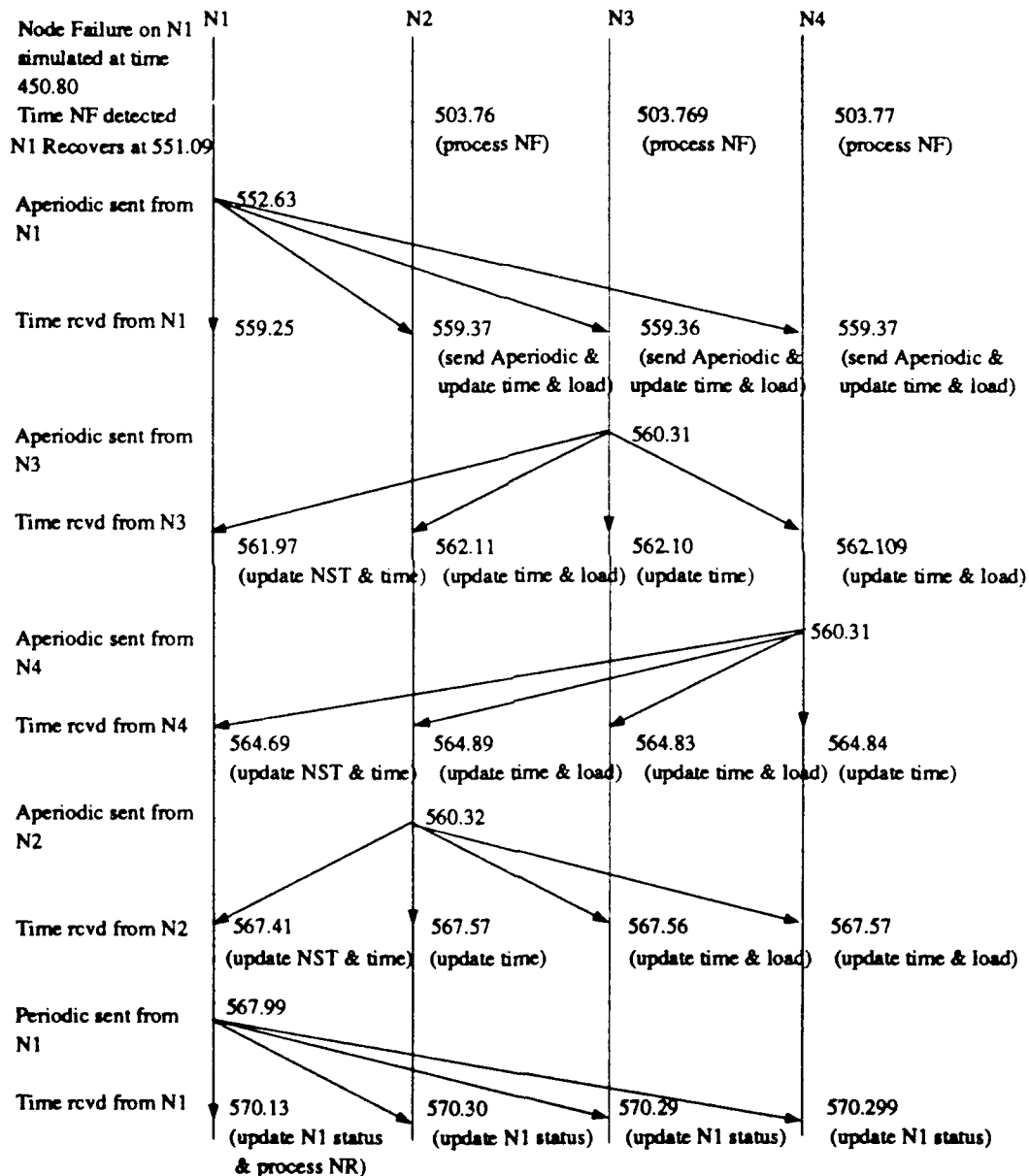


Figure 6.2: Reconfiguration Events upon Node Failure and Recovery

node and notify the **Node Overload** algorithm within RL. The algorithm for **Node Overload** can be modified slightly for AL to utilize when determining the load of its node. Additionally, when AL finishes processing messages for its active functions, it must update the NST's unique section pertaining to the function statistics. AL also needs to be able to modify its active function queues in addition to activating or deactivating the functions. The LIFFCL is required to maintain the queues for the non-active functions, and this needs to be implemented as well.

F. SUMMARY

The actual code implemented in the simulation model and its output are contained in Appendices A and B respectively. The sections which are not completed can be easily implemented into the current code. Comments have been inserted in the areas where a non-implemented algorithm or procedure needs to be placed for the current program to run. The next step, aside from the limitations listed in the previous section, is to get the code running on four separate processors. Additionally, a possible comparison can be done with a system in which all *data* messages need to be retransmitted upon migration versus the method currently used in which *data* messages are stored in queues.

VII. CONCLUSION

A. GENERAL

The main objective in distributed processing is to increase system throughput. Therefore, it is imperative that allocation algorithms are not only efficient and correct, but fast as well. Static allocation is simple and fast, yet the fact that it is based on *a priori* information renders it inflexible [Ref. 12]. Dynamic allocation used in reconfiguration, based on preemptive, priority-based scheduling, compensates for this inflexibility by considering task deadlines and environmental elements.

B. STATIC ALLOCATION

The proposed algorithm, LNF, requires fewer computations than HNF, allowing faster, simpler initial allocation. The complexity of HNF is $O(n \log n)$, n being the number of nodes, whereas LNF is linear [Ref. 6]. Although the initial weights of each node are somewhat equally balanced, LNF also does not guarantee that all nodes successfully process all assigned functions prior to their deadline times. This is partially attributed to the mix of periodic and aperiodic functions and/or the imprecise nature of *a priori* function information which is at best only an estimate of the characteristics [Ref. 5]. The dynamic reconfiguration compensates for this condition and allow for the functions to be migrated.

C. DYNAMIC ALLOCATION

The reconfiguration algorithms utilize a combination of load and IMC, when possible, to maximize system throughput while minimizing the IMC. In the case of node failure, only the IMC is utilized. Both load and IMC could have been used, but

to speed up the recovery of the active functions on the failed node, only the IMC is utilized. The dynamic reconfiguration algorithms are able to use the most current statistics of the global state. Since the information is current, based on the actual processing of each node, the dynamic assignment of functions can be made with more accuracy than static allocation. Additionally, the dynamic allocation allows for reassignment so that system degradation can be minimized. For instance, an overloaded node degrades processing of other nodes, particularly if the other nodes contain active functions which are dependent on functions active on the overloaded node.

D. PROCESSING OF ALGORITHMS

In determining which node(s) actually process the static and dynamic algorithms, a node's status and communication overhead required are considered. For static allocation, all nodes process the algorithm to prevent dependency on one node to do the allocation. It not only decreases communications required, but it allows for any node to be *down* at start-up. If one node did the processing, there are a minimum of k messages required to be transmitted versus no messages required if all nodes process the static allocation algorithm.

For a node overload condition, only the overloaded node processes the **Node Overload** algorithm. This prevents interruption of normal processing occurring at the other nodes. Processing of the algorithm at other nodes could possibly cause overload at these nodes as well.

For the case of node failure, all nodes process the **Node Failure** algorithm. Each node is able to detect the failure and can respond quickly. Also, the only communications required are the *fnon* messages; one for each active function on the failed node. If one node was designated to process the algorithm, the same problem

exists as with static allocation which is having to know which node is *up* to do the processing. Additionally, acknowledgement would be required i.e., a *fnon* message acknowledging that a node will activate the function designated needs to be sent; therefore, at least twice as many messages would be transmitted.

For recovery, only the recovering node processes the **Node Recovery** algorithm because its NST reflects the most current statistics of the system.

E. SUMMARY

Allocation algorithms represent a compromise between conflicting factors, and therefore, have different advantages and disadvantages. In some cases, algorithms that prove to be better than others tend to be very complex and difficult to implement. Additionally, performance of the algorithms can be very dependent upon the system and node configurations. The algorithms described in the thesis minimize the overhead caused by both IMC and the complexity of the algorithm. More importantly, they are designed to work well within the proposed framework of the system.

APPENDIX A: SIMULATION CODE

```
/* This program code is part of a joint project.  Members of */
/* the project team are as follows: S. Shukla, C. Yang, */
/* R. Puett, and K. Lehman */
/* The code is given in its entirety for completeness of */
/* of the topics covered in this thesis */
/* The code is in no particular order except for the first few */
/* sections which are the base for the remaining sections. */
/* Each section has comments preceding it and before each sub- */
/* section or task/procedure within the section to define what */
/* is occurring within that section. */
/* The first section contains the DECLARATIONS which are */
/* used throughout the program.  For each of the remaining */
/* sections, a specification package precedes the package body. */
/* The package PROCESS is the second section because it needs */
/* to be compiled before the packages following it.  It is the */
/* package that contains the algorithms.  The next section is */
/* TRAND.  It is the random number generator and needs to be */
/* compiled prior to compiling COMMNET which follows TRAND. */
/* COMMNET creates the instantiations to form the nodes.  The */
/* ordering of what follows from this point on does not matter. */
/* The remaining sections are listed in the following order: */
/* INS - contains the NODE_INITIALIZER and INPUT_SERVER tasks */
/* OUTS - contains the OUTPUT_SERVER task */
/* CKPT - contains the CHECK_PT and EVENT_CNT tasks */
/* RL - contains the RECONF_LAYER task */
/* SM - contains the STATUS_REC and STATUS_BDCST tasks */
/* FP - contains the EVENT_MAKER i.e., Event Generator */
/* FEP - Front-End Processor which opens output files for each */
/* node and initiates the NST for each node. */
```

```
with text_io; use text_io;
with calendar; use calendar;
package DECLARATIONS is
```

```
F1,F2,F3,F4 : FILE_TYPE;
type MSG_TYPE is (data,control);
type ACTION_TYPE is (MKR,FNON,FNOFF,STATUS,CHKPT);
type IMCM is array(1..12,1..12)of integer; --IPC comms array
type FI is array(1..4)of integer; --function information params.
type FL is array(1..12)of integer; --function location array
type NSL is array(1..2,1..4)of integer;--Node status and load
type RCY is array(1..4)of integer; --array used when recovering
type STAT_TIME is array(1..4)of float; --array used in each node to
type FAIL_FLG is array(1..12)of boolean; --array used in each node to
-- record the times when status
```

```

-- msgs were sent by other nodes
-- contents of the unique section
type FUNCTION_REC is
  record
    TTC          : float;
    TTD          : float;
    FN_INFO      : FI;
    LAST_MSG_PROC : float;
    LAST_MSG_REC  : float;
    REGISTER_VAL  : integer := 0;
    SYMBOL_VAR    : integer := 0;
  end record;
type FUNCTION_STATS is array(1..12) of FUNCTION_REC;
type UNIQUE is array(1..4) of FUNCTION_STATS;
type COMMON is
  record
    NODE_STAT_LD : NSL;          -- node status and load
    FN_LOC       : FL;
    IMC          : IMCM;
  end record;
type BODY_TYPE is
  record
    DATA : string(1..80);
    UNIQ  : FUNCTION_STATS;
    COMM  : COMMON;
  end record;
type MSG_RECORD IS                --msg to be passed on the net
  record
    TOT          : float;        --Time of Transmit of a msg
    TOR          : float;        --Time of Receipt of a msg
    MSG_KIND     : MSG_TYPE;     --type of msg
    DEST_FUNC    : integer := 0; --which fn a msg is sent to
    DEST_NODE    : integer := 0; --node who acts on a msg
    ORIG_FN_NODE : integer := 0; --originator (fn or Node) of msg
    CNTRL_ACTION : ACTION_TYPE
    MSG_BODY     : BODY_TYPE;    --msg that needs to be read
  end record;
Q_SIZE : constant integer := 15; --size of message queues
type QUEUE is array (1..Q_SIZE) of MSG_RECORD;
type MSG_QUEUE is                --queue to hold msgs to send out
  record
    MSG_TO_SEND : boolean := false;--indicates if queue has a msg
    BLOCK_WRITE : boolean := false;--used to block writing to queue
    RD_CNT      : integer := 1;   --the read pointer in queue
    MSG_CNT     : integer := 1;   --the write pointer in queue
    MSG_QUE     : QUEUE;          --holds up to 15 msgs
  end record;
type NODE_STATUS_TABLE is        --defines contents of the NST
  record
    COMMON_SECTION : COMMON;
    UNIQUE_SECTION : UNIQUE;
    NODE_ID        : integer := 0;

```

```

end record;
type VARIABLES is
    --status conditions for a node
    --(local to each node)
    record
        RCVRY_IN_PROG: boolean := false;--indicates node recovery
        RCVRY        : RCY;          --array used in rcvry process
        UNIQ_SENT    : boolean := false;--indicates if a unique section
        -- was sent by a node
        CHKPT_TAKEN  : RCY;          --array used to indicate if a
        -- checkpoint is complete or not
        CHKPT_ORIG   : boolean := false;-- node originating chkpt
        CHKPT_COMPLETE : boolean := false;--a completed checkpoint done
        LOCAL_CHKPT  : boolean := false;--indicates if a node has taken
        -- a checkpoint
        CHKPT_TIMER   : float;
        FIRST_MKR    : boolean := false;--flag to note 1st marker msg to
        -- come across net - indicates a
        -- checkpoint needs to occur
        EVNT_CNT     : integer := 0;  --cnts up to 25 then resets to 1
        --(indicates when a chkpt needs
        -- to be taken)
        EVNT_CNT_OUT : integer := 0;  -- events sent by output server
        ACTIVE_FN_QUE : QUEUE;        -- msgs for assigned functions
        DATA_MSG_QUE : QUEUE;        -- holds msg for all functions
        OUTQ          : MSG_QUEUE;    --queue to hold output msgs
        INQ           : MSG_QUEUE;    --queue to hold input msgs
        TIMER         : STAT_TIME;    --array to hold times of when
        -- status msgs were sent
    end record;
NST,NSTBAK : array(1..4)of NODE_STATUS_TABLE;
LOC_VAR : array(1..4)of VARIABLES;--gives each node a set of Loc Vars
ST      : array(1..4)of NODE_STATUS_TABLE;--temporary copy of NST
NET_BUSY: boolean;          --indicates if network is tied up
NET_Q   : MSG_QUEUE;        --queue to hold msgs for network
FAILED_NODE : FAIL_FLG;    --used to indicated failed node
end DECLARATIONS;

with DECLARATIONS; use DECLARATIONS;
with TEXT_IO; use TEXT_IO;
package PROCESS is

    --this procedure gets and prints the current value of real time
    procedure GET_REAL_TIME(NID: in integer; LT: in out float);

    --this procedure processes a marker msg
    procedure MKR_MSG (M:in out MSG_RECORD;NID:in integer;FLG:in out
        boolean);

    --this procedure processes a function on msg
    procedure FN_ON_MSG (M : in MSG_RECORD; NID : in integer);

```

```
--this procedure processes a function off msg
procedure FN_OFF_MSG(M:in out MSG_RECORD;NID:in integer;MSG_FLAG:
                    in out boolean);
```

```
--this procedure processes a status msg
procedure STAT_MSG (M:in out MSG_RECORD;NID:in integer;FLG:in out
                    boolean);
```

```
--this procedure processes a checkpoint complete msg;
procedure CHK_PT_CMPLT_MSG (M : in MSG_RECORD; NID : in integer);
```

```
end PROCESS;
```

```
with text_io;
package FLOAT_INOUT is new TEXT_IO.FLOAT_IO(FLOAT);
with FLOAT_INOUT; use FLOAT_INOUT;
with text_io; use text_io;
with number_io; use number_io;
with integer_io; use integer_io;
with calendar; use calendar;
with DECLARATIONS; use DECLARATIONS;
```

```
-- The package PROCESS contains all the procedures necessary
-- to process the different types of messages that come into
-- the Input Server. Each procedure is preceeded by a
-- description of its actions.
```

```
package body PROCESS is
```

```
-- Procedure Get Real Time utilizes the system package
-- calendar to access the Real time clock of the system
-- processor. In this case, only the seconds portion of
-- the calendar is utilized.
```

```
procedure GET_REAL_TIME(NID: in integer;LT: in out float) is
```

```
S : DAY_DURATION;
```

```
R : TIME;
```

```
T : float;
```

```
begin
```

```
    R := clock;
```

```
    S := SECONDS(R);
```

```
    T := float(S);
```

```
    LT := T;
```

```
    case NID is
```

```
        when 1 =>
```

```
            PUT(F1,T,6,5,0);
```

```
            SET_COL(F1,15);
```

```
            PUT(F1," Node #1");
```

```
        when 2 =>
```

```

        PUT(F2,T,6,5,0);
        SET_COL(F2,15);
        PUT(F2," Node #2");
    when 3 =>
        PUT(F3,T,6,5,0);
        SET_COL(F3,15);
        PUT(F3," Node #3");
    when 4 =>
        PUT(F4,T,6,5,0);
        SET_COL(F4,15);
        PUT(F4," Node #4");
    when others =>
        NULL;
    end case;
end GET_REAL_TIME;

-- Procedure Function On Message is called from the
-- Reconfiguration task. It processes a FNON message
-- and updates a Node's NST to reflect the indicated
-- function's location.

procedure FN_ON_MSG(M :in MSG_RECORD; NID : in integer) is
    Z,Y,X      : integer;
    GM         : MSG_RECORD;
    PT         : float := 0.0;
    DEACT_NODE : integer;
begin
    GM := M;
    Z := NST(NID).NODE_ID;
    Y := M.DEST_FUNC;
    DEACT_NODE := NST(Z).COMMON_SECTION.FN_LOC(Y);
    NST(Z).COMMON_SECTION.FN_LOC(Y) := M.ORIG_FN_NODE;
    case Z is
        -- write info to specific output file
    when 1 =>
        GET_REAL_TIME(Z,PT);
        SET_COL(F1,25);
        PUT(F1,"R_L rcvd FN_ON from Node #");
        PUT(F1,M.ORIG_FN_NODE,1);
        SET_COL(F1,60);
        PUT(F1,"EVNT #");
        PUT(F1,M.MSG_BODY.UNIQ(1).SYMBOL_VAR,4);
        SET_COL(F1,72);
        if M.ORIG_FN_NODE = Z then -- activating node - turns fn on
            PUT_LINE(F1,"I am the activating node and changing NST.");
        else
            if DEACT_NODE = Z then--deactivating node
                PUT_LINE(F1,"I am the deactivating node and changing NST.");
            else
                PUT_LINE(F1,"Neither act/deact node and changing NST.");
            end if;
        end if;
    end if;
end if;

```

```

SET_COL(F1,72);          -- shows changes in NST from FNON
for R in 1..12 loop
    PUT(F1,NST(Z).COMMON_SECTION.FN_LOC(R),3);
end loop;
NEW_LINE(F1);
when 2 =>
    GET_REAL_TIME(Z,PT);
    SET_COL(F2,25);
    PUT(F2,"R_L rcvd FN_ON from Node #");
    PUT(F2,M.ORIG_FN_NODE,1);
    SET_COL(F2,60);
    PUT(F2,"EVNT #");
    PUT(F2,M.MSG_BODY.UNIQ(1).SYMBOL_VAR,4);
    SET_COL(F2,72);
    if M.ORIG_FN_NODE = Z then --activating node, turns fn on
        PUT_LINE(F2,"I am the activating node and changing NST.");
    else
        if DEACT_NODE = Z then--deactivating node
            PUT_LINE(F2,"I am the deactivating node and changing NST");
        else
            PUT_LINE(F2,"Neither act/deact node and changing NST.");
        end if;
    end if;
    SET_COL(F2,72);      -- shows changes in NST from FNON
    for R in 1..12 loop
        PUT(F2,NST(Z).COMMON_SECTION.FN_LOC(R),3);
    end loop;
    NEW_LINE(F2);
when 3 =>
    GET_REAL_TIME(Z,PT);
    SET_COL(F3,25);
    PUT(F3,"R_L rcvd FN_ON from Node #");
    PUT(F3,M.ORIG_FN_NODE,1);
    SET_COL(F3,60);
    PUT(F3,"EVNT #");
    PUT(F3,M.MSG_BODY.UNIQ(1).SYMBOL_VAR,4);
    SET_COL(F3,72);
    if M.ORIG_FN_NODE = Z then -- activating node - turns fn on
        PUT_LINE(F3,"I am the activating node and changing NST.");
    else
        if DEACT_NODE = Z then--deactivating node
            PUT_LINE(F3,"I am the deactivating node and changing NST");
        else
            PUT_LINE(F3,"Neither act/deact node and changing NST.");
        end if;
    end if;
    SET_COL(F3,72);      -- shows changes in NST from FNON
    for R in 1..12 loop
        PUT(F3,NST(Z).COMMON_SECTION.FN_LOC(R),3);
    end loop;
    NEW_LINE(F3);

```

```

when 4 =>
    GET_REAL_TIME(Z,PT);
    SET_COL(F4,25);
    PUT(F4,"R_L rcvd FN_ON from Node #");
    PUT(F4,M.ORIG_FN_NODE,1);
    SET_COL(F4,60);
    PUT(F4,"EVNT #");
    PUT(F4,M.MSG_BODY.UNIQ(1).SYMBOL_VAR,4);
    SET_COL(F4,72);
    if M.ORIG_FN_NODE = Z then --activating node - turns fn on
        PUT_LINE(F4,"I am the activating node and changing NST.");
    else
        if DEACT_NODE = Z then--deactivating node
            PUT_LINE(F4,"I am the deactivating node and changing NST");
        else
            PUT_LINE(F4,"Neither act/deact node and changing NST.");
        end if;
    end if;
    SET_COL(F4,72);      -- shows changes in NST from FNON
    for R in 1..12 loop
        PUT(F4,NST(Z).COMMON_SECTION.FN_LOC(R),3);
    end loop;
    NEW_LINE(F4);
when others =>
    NULL;
end case;
end FN_ON_MSG;

```

-- Procedure Function Off Message is called by the Reconfiguration task. It processes a FNOFF message and determines if the node is to activate a function. It also generates a FNON message if necessary.

```

procedure FN_OFF_MSG(M:in out MSG_RECORD;NID: in integer;MSG_FLAG:
                    in out boolean) is

```

```

    Z,Y : integer;
    J : MSG_RECORD;
    PT : float := 0.0;
begin
    Z := NST(NID).NODE_ID;
    Y := M.DEST_NODE;
    GET_REAL_TIME(Z,PT);
    case Z is
        when 1 =>
            SET_COL(F1,25);
            PUT(F1,"R_L rcvd FN_OFF from Node #");
            PUT(F1,M.ORIG_FN_NODE,1);
            SET_COL(F1,60);
            PUT(F1,"EVNT #");
            PUT(F1,M.MSG_BODY.UNIQ(1).SYMBOL_VAR,4);
            SET_COL(F1,72);

```

```

    if Z = Y then
        PUT(F1,"FN_ON sent to activate FN #");
        PUT(F1,M.DEST_FUNC,2);NEW_LINE(F1);
    else
        PUT_LINE(F1,"No further action required ATT.");
    end if;
when 2 =>
    SET_COL(F2,25);
    PUT(F2,"R_L rcvd FN_OFF from Node #");
    PUT(F2,M.ORIG_FN_NODE,1);
    SET_COL(F2,60);
    PUT(F2,"EVNT #");
    PUT(F2,M.MSG_BODY.UNIQ(1).SYMBOL_VAR,4);
    SET_COL(F2,72);
    if Z = Y then
        PUT(F2,"FN_ON sent to activate FN #");
        PUT(F2,M.DEST_FUNC,2);NEW_LINE(F2);
    else
        PUT_LINE(F2,"No further action required ATT.");
    end if;
when 3 =>
    SET_COL(F3,25);
    PUT(F3,"R_L rcvd FN_OFF from Node #");
    PUT(F3,M.ORIG_FN_NODE,1);
    SET_COL(F3,60);
    PUT(F3,"EVNT #");
    PUT(F3,M.MSG_BODY.UNIQ(1).SYMBOL_VAR,4);
    SET_COL(F3,72);
    if Z = Y then
        PUT(F3,"FN_ON sent to activate FN #");
        PUT(F3,M.DEST_FUNC,2);NEW_LINE(F3);
    else
        PUT_LINE(F3,"No further action required ATT.");
    end if;
when 4 =>
    SET_COL(F4,25);
    PUT(F4,"R_L rcvd FN_OFF from Node #");
    PUT(F4,M.ORIG_FN_NODE,1);
    SET_COL(F4,60);
    PUT(F4,"EVNT #");
    PUT(F4,M.MSG_BODY.UNIQ(1).SYMBOL_VAR,4);
    SET_COL(F4,72);
    if Z = Y then
        PUT(F4,"FN_ON sent to activate FN #");
        PUT(F4,M.DEST_FUNC,2);NEW_LINE(F4);
    else
        PUT_LINE(F4,"No further action required ATT.");
    end if;
when others =>
    NULL;
end case;

```

```

    if Z = Y then                                -- activating node
        -- create FNON msg to send
        J.MSG_KIND := CONTROL;
        J.DEST_FUNC := M.DEST_FUNC;
        J.ORIG_FN_NODE := Z;
        J.CNTRL_ACTION := FNON;
        -- set flag to indicate msg needs to go to OUTPUT_SERVER
        MSG_FLAG := true;
        M := J;
    end if;
end FN_OFF_MSG;

-- Procedure Status Message processes both periodic and aperiodic
-- status messages. It is called by Status Monitor (SM). The
-- recovery process is handled by this procedure. Recovery is
-- accomplished by rebuilding the NST of the recovering node
-- from the contents of aperiodic messages (i.e. the Unique
-- Section)

procedure STAT_MSG(M : in out MSG_RECORD; NID : in integer; FLG :
                  in out boolean) is
    X,Z,Y : integer;
    GM : MSG_RECORD;
    RCVRY_COMPLETE : boolean := false;
    MY_UNIQ_SENT : boolean := false;
    PT : float := 0.0;
begin --Dest.Node field is used to designate a periodic msg (1)
    -- or an aperiodic msg (2). The Dest.Fn field holds the value
    -- of the load of a node designated by the ORIG_FN_NODE.
    Z := NST(NID).NODE_ID;
    Y := M.DEST_FUNC;
    X := M.ORIG_FN_NODE;
    LOC_VAR(Z).TIMER(X) := M.TOR;    --update periodic time of node
    NST(Z).COMMON_SECTION.NODE_STAT_LD(2,X) := M.DEST_FUNC;
                                         -- node load percentage.
    GET_REAL_TIME(0,PT);
    if LOC_VAR(Z).RCVRY_IN_PROG and
        PT - LOC_VAR(Z).TIMER(Z) > 61.5 then
        LOC_VAR(Z).RCVRY_IN_PROG := false;
        NST(Z).COMMON_SECTION.NODE_STAT_LD(1,Z) := 0;
        NST(Z).COMMON_SECTION.NODE_STAT_LD(2,Z) := 0;
        for J in 1..4 loop                -- clear rcvry array
            LOC_VAR(Z).RCVRY(J) := 0;
        end loop;
        case Z is
            when 1 =>
                GET_REAL_TIME(1,PT);
                SET_COL(F1,72);
                PUT_LINE(F1,"RCVRY attempts unsuccessful. Restart RCVRY");
            when 2 =>
                GET_REAL_TIME(2,PT);

```

```

        SET_COL(F2,72);
        PUT_LINE(F2,"RCVRY attempts unsuccessful. Restart RCVRY");
    when 3 =>
        GET_REAL_TIME(3,PT);
        SET_COL(F3,72);
        PUT_LINE(F3,"RCVRY attempts unsuccessful. Restart RCVRY");
    when 4 =>
        GET_REAL_TIME(4,PT);
        SET_COL(F4,72);
        PUT_LINE(F4,"RCVRY attempts unsuccessful. Restart RCVRY");
    when others =>
        NULL;
    end case;
end if;
if M.DEST_NODE = 1 then
    --periodic msg
    if NST(Z).COMMON_SECTION.NODE_STAT_LD(1,X) = 0 and
        M.DEST_FUNC = 0 then
        LOC_VAR(Z).UNIQ_SENT := false;
        NST(Z).COMMON_SECTION.NODE_STAT_LD(1,X) := 1;
        FAILED_NODE(X) := false;
    end if;
    if not LOC_VAR(Z).RCVRY_IN_PROG and
        NST(Z).COMMON_SECTION.NODE_STAT_LD(1,Z) = 0 then
        PUT_LINE("BUILDING an APERIODIC message.");
        GM.DEST_NODE := 2;    -- build aperiodic status message
        GM.DEST_FUNC := 0;
        GM.ORIG_FN_NODE := Z;
        GM.CNTRL_ACTION := STATUS;
        GM.MSG_KIND := control;
        FLG := true;
        LOC_VAR(Z).RCVRY_IN_PROG := true;
        for I in 1..4 loop -- reset timers of nodes other than the
            if I /= X then -- node whose periodic msg was received
                LOC_VAR(Z).TIMER(I) := PT;
            end if;
        end loop;
    end if;
else
    -- aperiodic msg
    if NST(Z).COMMON_SECTION.NODE_STAT_LD(1,Z) = 0 then
        --recovery node
        LOC_VAR(Z).RCVRY(X) := 1;
        if Z /= X then
            NST(Z).UNIQUE_SECTION(X) := M.MSG_BODY.UNIQ;
            NST(Z).COMMON_SECTION := M.MSG_BODY.COMM;
        end if;
        RCVRY_COMPLETE := true;

        for I in 1..4 loop
            -- check if all nodes sent the
            -- unique sections
            if NST(Z).COMMON_SECTION.NODE_STAT_LD(1,I) = 1 then
                -- active node
            end if;
        end loop;
    end if;
end if;

```

```

        if LOC_VAR(Z).RCVRY(I) = 0 then
            RCVRY_COMPLETE := false;
        end if;
    end if;
end loop;
if RCVRY_COMPLETE then    -- call the node recovery
    -- procedure
    GM.DEST_NODE := 1;    -- build periodic status message
    GM.DEST_FUNC := 0;    -- indicates rcvry complete to
    -- other nodes

    GM.ORIG_FN_NODE := Z;
    GM.CNTRL_ACTION := STATUS;
    GM.MSG_KIND := control;
    FLG := true;
    LOC_VAR(Z).RCVRY_IN_PROG := false;
    for J in 1..4 loop    -- clear rcvry array
        LOC_VAR(Z).RCVRY(J) := 0;
    end loop;
end if;
else    -- not the orig node of APERIODIC
    -- chk if unique section was sent

    if not LOC_VAR(Z).UNIQ_SENT then
        GM.DEST_NODE := 2;    -- build an aperiodic status message
        GM.DEST_FUNC := NST(Z).COMMON_SECTION.NODE_STAT_LD(2,NID);
        GM.ORIG_FN_NODE := Z;
        GM.MSG_BODY.UNIQ := NST(Z).UNIQUE_SECTION(Z);
        GM.MSG_BODY.COMM := NST(Z).COMMON_SECTION;
        GM.CNTRL_ACTION := STATUS;
        GM.MSG_KIND := control;
        FLG := true;
        MY_UNIQ_SENT := true;
        LOC_VAR(Z).UNIQ_SENT := true;
    end if;    -- UNIQ_SENT
end if;
end if;
GET_REAL_TIME(Z,PT);
case Z is
    when 1 =>
        SET_COL(F1,25);
        if M.DEST_NODE = 1 then
            PUT(F1,"S_M rcvd PERIODIC from Node #");
        else
            PUT(F1,"S_M rcvd APERIODIC from Node #");
        end if;
        PUT(F1,M.ORIG_FN_NODE,1);
        SET_COL(F1,60);
        PUT(F1,"EVNT #");
        PUT(F1,M.MSG_BODY.UNIQ(1).SYMBOL_VAR,4);
        SET_COL(F1,72);
        if M.DEST_NODE = 1 then

```

```

        PUT(F1,"Reset Timer element of Node #");
        PUT(F1,M.ORIG_FN_NODE,1);
        NEW_LINE(F1);
    else
        if NST(Z).COMMON_SECTION.NODE_STAT_LD(1,Z) = 0 then
            if RCVRY_COMPLETE then
                PUT_LINE(F1,"Recovery complete,send PERIODIC msg");
            else
                PUT_LINE(F1,"This is the recovering node.");
            end if;
        else
            if LOC_VAR(Z).UNIQ_SENT and MY_UNIQ_SENT then
                PUT_LINE(F1,"Sending APERIODIC with uniq sect.");
            else
                PUT_LINE(F1,"APERIODIC response sent, no action.");
            end if;
        end if;
    end if;
when 2 =>
    SET_COL(F2,25);
    if M.DEST_NODE = 1 then
        PUT(F2,"S_M rcvd PERIODIC from Node #");
    else
        PUT(F2,"S_M rcvd APERIODIC from Node #");
    end if;
    PUT(F2,M.ORIG_FN_NODE,1);
    SET_COL(F2,60);
    PUT(F2,"EVNT #");
    PUT(F2,M.MSG_BODY.UNIQ(1).SYMBOL_VAR,4);
    SET_COL(F2,72);
    if M.DEST_NODE = 1 then
        PUT(F2,"Reset Timer element of Node #");
        PUT(F2,M.ORIG_FN_NODE,1);
        NEW_LINE(F2);
    else
        if NST(Z).COMMON_SECTION.NODE_STAT_LD(1,Z) = 0 then
            if RCVRY_COMPLETE then
                PUT_LINE(F2,"Recovery complete,send PERIODIC msg");
            else
                PUT_LINE(F2,"This is the recovering node.");
            end if;
        else
            if LOC_VAR(Z).UNIQ_SENT and MY_UNIQ_SENT then
                PUT_LINE(F2,"Sending APERIODIC with uniq sect.");
            else
                PUT_LINE(F2,"APERIODIC response sent, no action.");
            end if;
        end if;
    end if;
when 3 =>
    SET_COL(F3,25);

```

```

if M.DEST_NODE = 1 then
    PUT(F3,"S_M rcvd PERIODIC from Node #");
else
    PUT(F3,"S_M rcvd APERIODIC from Node #");
end if;
PUT(F3,M.ORIG_FN_NODE,1);
SET_COL(F3,60);
PUT(F3,"EVNT #");
PUT(F3,M.MSG_BODY.UNIQ(1).SYMBOL_VAR,4);
SET_COL(F3,72);
if M.DEST_NODE = 1 then
    PUT(F3,"Reset Timer element of Node #");
    PUT(F3,M.ORIG_FN_NODE,1);
    NEW_LINE(F3);
else
    if NST(Z).COMMON_SECTION.NODE_STAT_LD(1,Z) = 0 then
        if RCVRY_COMPLETE then
            PUT_LINE(F3,"Recovery complete,send PERIODIC msg");
        else
            PUT_LINE(F3,"This is the recovering node.");
        end if;
    else
        if LOC_VAR(Z).UNIQ_SENT and MY_UNIQ_SENT then
            PUT_LINE(F3,"Sending APERIODIC with uniq sect.");
        else
            PUT_LINE(F3,"APERIODIC response sent, no action.");
        end if;
    end if;
end if;
when 4 =>
    SET_COL(F4,25);
    if M.DEST_NODE = 1 then
        PUT(F4,"S_M rcvd PERIODIC from Node #");
    else
        PUT(F4,"S_M rcvd APERIODIC from Node #");
    end if;
    PUT(F4,M.ORIG_FN_NODE,1);
    SET_COL(F4,60);
    PUT(F4,"EVNT #");
    PUT(F4,M.MSG_BODY.UNIQ(1).SYMBOL_VAR,4);
    SET_COL(F4,72);
    if M.DEST_NODE = 1 then
        PUT(F4,"Reset Timer element of Node #");
        PUT(F4,M.ORIG_FN_NODE,1);
        NEW_LINE(F4);
    else
        if NST(Z).COMMON_SECTION.NODE_STAT_LD(1,Z) = 0 then
            if RCVRY_COMPLETE then
                PUT_LINE(F4,"Recovery complete,send PERIODIC msg");
            else
                PUT_LINE(F4,"This is the recovering node.");
            end if;
        end if;
    end if;
end if;

```

```

        end if;
    else
        if LOC_VAR(Z).UNIQ_SENT and MY_UNIQ_SENT then
            PUT_LINE(F4,"Sending APERIODIC with uniq sect.");
        else
            PUT_LINE(F4,"APERIODIC response sent, no action.");
        end if;
    end if;
end if;
when others =>
    NULL;
end case;
MY_UNIQ_SENT := false;
if FLG then
    M := GM;
end if;
end STAT_MSG;

```

-- Procedure Marker Message processes a MKR message utilized for
 -- the checkpointing process. It is called from the CHECK_PT
 -- task. The node's NST is updated with the contents of the
 -- message body. The procedure also generates a checkpoint
 -- complete message at the node originating checkpoint to
 -- indicate a successful checkpoint.

```

procedure MKR_MSG(M : in out MSG_RECORD; NID : in integer; FLG :
                  in out boolean) is
    X,Z,Y : integer;
    GM : MSG_RECORD;
    PT : float := 0.0;
begin
    Z := NST(NID).NODE_ID;
    Y := M.ORIG_FN_NODE;
    if not LOC_VAR(Z).FIRST_MKR then
        LOC_VAR(Z).FIRST_MKR := true;
        if Y = Z then
            LOC_VAR(Z).CHKPT_ORIG := true;
            LOC_VAR(Z).CHKPT_TAKEN(Z) := 1;
            GET_REAL_TIME(0,PT);
            LOC_VAR(NID).CHKPT_TIMER := PT;
        else
            LOC_VAR(Z).CHKPT_ORIG := false;
        end if;
    end if;
    if Y /= Z then
        -- not originating node of msg
        NST(Z).UNIQUE_SECTION(Y) := M.MSG_BODY.UNIQ;
        if LOC_VAR(Z).CHKPT_ORIG = true then -- check point originator
            LOC_VAR(Z).CHKPT_TAKEN(Y) := 1;
            LOC_VAR(Z).CHKPT_COMPLETE := true;
            for I in 1..4 loop
                if NST(Z).COMMON_SECTION.NODE_STAT_LD(1,I) = 1 then

```

```

-- node active
    if LOC_VAR(Z).CHKPT_TAKEN(I) = 0 then
        LOC_VAR(Z).CHKPT_COMPLETE := false;
    end if;
end if;
end loop;
if LOC_VAR(Z).CHKPT_COMPLETE = true then
    GM.MSG_KIND := CONTROL;
    GM.CNTRL_ACTION := CHKPT;
    GM.ORIG_FN_NODE := Z;
    FLG := true;
end if;
else
    -- not originating node
    if not LOC_VAR(Z).LOCAL_CHKPT then -- didn't send unique sect
        ST(Z) := NST(Z);
        GM.MSG_KIND := CONTROL;
        GM.CNTRL_ACTION := MKR;
        GM.ORIG_FN_NODE := Z;
        GM.MSG_BODY.UNIQ := NST(Z).UNIQUE_SECTION(Z);
        FLG := true;
        LOC_VAR(Z).LOCAL_CHKPT := true; --true if checkpointed
    end if;
end if;
end if;
GET_REAL_TIME(Z,PT);
case Z is
    when 1 =>
        SET_COL(F1,25);
        PUT(F1,"C_P rcvd MKR from Node #");
        PUT(F1,M.ORIG_FN_NODE,1);
        SET_COL(F1,60);
        PUT(F1,"EVNT #");
        PUT(F1,M.MSG_BODY.UNIQ(1).SYMBOL_VAR,4);
        SET_COL(F1,72);
        if LOC_VAR(Z).CHKPT_ORIG then
            if LOC_VAR(Z).CHKPT_COMPLETE then
                PUT_LINE(F1,"MKRs rcvd from all nodes,Send CHKPT_COMP");
            else
                PUT_LINE(F1,"I originated CHKPT. Not all MKRs yet rcvd");
            end if;
        else
            if not LOC_VAR(Z).LOCAL_CHKPT then
                PUT_LINE(F1,"Local CHKPT conducted. Send uniq in MKR.");
            else
                PUT_LINE(F1,"Local CHKPT already conducted. Store UNIQ");
            end if;
        end if;
    when 2 =>
        SET_COL(F2,25);
        PUT(F2,"C_P rcvd MKR from Node #");
        PUT(F2,M.ORIG_FN_NODE,1);

```

```

SET_COL(F2,60);
PUT(F2,"EVNT #");
PUT(F2,M.MSG_BODY.UNIQ(1).SYMBOL_VAR,4);
SET_COL(F2,72);
if LOC_VAR(Z).CHKPT_ORIG then
  if LOC_VAR(Z).CHKPT_COMPLETE then
    PUT_LINE(F2,"MKRs rcvd from all nodes,Send CHKPT_COMP");
  else
    PUT_LINE(F2,"I originated CHKPT. Not all MKRs yet rcvd");
  end if;
else
  if not LOC_VAR(Z).LOCAL_CHKPT then
    PUT_LINE(F2,"Local CHKPT conducted. Send uniq in MKR.");
  else
    PUT_LINE(F2,"Local CHKPT already conducted. Store UNIQ");
  end if;
end if;
when 3 =>
  SET_COL(F3,25);
  PUT(F3,"C_P rcvd MKR from Node #");
  PUT(F3,M.ORIG_FN_NODE,1);
  SET_COL(F3,60);
  PUT(F3,"EVNT #");
  PUT(F3,M.MSG_BODY.UNIQ(1).SYMBOL_VAR,4);
  SET_COL(F3,72);
  if LOC_VAR(Z).CHKPT_ORIG then
    if LOC_VAR(Z).CHKPT_COMPLETE then
      PUT_LINE(F3,"MKRs rcvd from all nodes,Send CHKPT_COMP");
    else
      PUT_LINE(F3,"I originated CHKPT. Not all MKRs yet rcvd");
    end if;
  else
    if not LOC_VAR(Z).LOCAL_CHKPT then
      PUT_LINE(F3,"Local CHKPT conducted. Send uniq in MKR.");
    else
      PUT_LINE(F3,"Local CHKPT already conducted. Store UNIQ");
    end if;
  end if;
when 4 =>
  SET_COL(F4,25);
  PUT(F4,"C_P rcvd MKR from Node #");
  PUT(F4,M.ORIG_FN_NODE,1);
  SET_COL(F4,60);
  PUT(F4,"EVNT #");
  PUT(F4,M.MSG_BODY.UNIQ(1).SYMBOL_VAR,4);
  SET_COL(F4,72);
  if LOC_VAR(Z).CHKPT_ORIG then
    if LOC_VAR(Z).CHKPT_COMPLETE then
      PUT_LINE(F4,"MKRs rcvd from all nodes,Send CHKPT_COMP");
    else
      PUT_LINE(F4,"I originated CHKPT. Not all MKRs yet rcvd");
    end if;
  else
    if not LOC_VAR(Z).LOCAL_CHKPT then
      PUT_LINE(F4,"Local CHKPT conducted. Send uniq in MKR.");
    else
      PUT_LINE(F4,"Local CHKPT already conducted. Store UNIQ");
    end if;
  end if;

```

```

        end if;
    else
        if not LOC_VAR(Z).LOCAL_CHKPT then
            PUT_LINE(F4,"Local CHKPT conducted. Send uniq in MKR.");
        else
            PUT_LINE(F4,"Local CHKPT already conducted. Store UNIQ");
        end if;
    end if;
    when others =>
        NULL;
    end case;
    if FLG then
        M := GM;
    end if;
end MKR_MSG;

-- Procedure Checkpoint Complete Message processes a CHKPT message
-- that was built in the Status Message section. It resets all
-- flags set during the checkpointing process, and it copies
-- checkpoint data into the backup NST (NSTBAK).

procedure CHK_PT_CMPLT_MSG (M : in MSG_RECORD; NID : in integer) is
    Z,Y : integer := M.ORIG_FN_NODE;
    PT : float := 0.0;
begin
    NSTBAK(NID) := ST(NID);
    Z := NST(NID).NODE_ID;
    LOC_VAR(NID).FIRST_MKR := FALSE;
    LOC_VAR(NID).CHKPT_ORIG := FALSE;
    GET_REAL_TIME(Z,PT);
    LOC_VAR(NID).CHKPT_TIMER := PT;
    GET_REAL_TIME(Z,PT);
    case Z is
        when 1 =>
            SET_COL(F1,25);
            PUT(F1,"C_P rcvd CHKPT from Node #");
            PUT(F1,M.ORIG_FN_NODE,1);
            SET_COL(F1,60);
            PUT(F1,"EVNT #");
            PUT(F1,M.MSG_BODY.UNIQ(1).SYMBOL_VAR,4);
            SET_COL(F1,72);
            if Z = Y then
                PUT_LINE(F1,"CHKPT orig. Global CHKPT complete store NST");
            else
                PUT_LINE(F1,"Global CHKPT complete store NST");
            end if;
        when 2 =>
            SET_COL(F2,25);
            PUT(F2,"C_P rcvd CHKPT from Node #");
            PUT(F2,M.ORIG_FN_NODE,1);
            SET_COL(F2,60);
    end case;
end CHK_PT_CMPLT_MSG;

```

```

    PUT(F2,"EVNT #");
    PUT(F2,M.MSG_BODY.UNIQ(1).SYMBOL_VAR,4);
    SET_COL(F2,72);
    if Z = Y then
        PUT_LINE(F2,"CHKPT orig. Global CHKPT complete store NST");
    else
        PUT_LINE(F2,"Global CHKPT complete store NST");
    end if;
when 3 =>
    SET_COL(F3,25);
    PUT(F3,"C_P rcvd CHKPT from Node #");
    PUT(F3,M.ORIG_FN_NODE,1);
    SET_COL(F3,60);
    PUT(F3,"EVNT #");
    PUT(F3,M.MSG_BODY.UNIQ(1).SYMBOL_VAR,4);
    SET_COL(F3,72);
    if Z = Y then
        PUT_LINE(F3,"CHKPT orig. Global CHKPT complete store NST");
    else
        PUT_LINE(F3,"Global CHKPT complete store NST");
    end if;
when 4 =>
    SET_COL(F4,25);
    PUT(F4,"C_P rcvd CHKPT from Node #");
    PUT(F4,M.ORIG_FN_NODE,1);
    SET_COL(F4,60);
    PUT(F4,"EVNT #");
    PUT(F4,M.MSG_BODY.UNIQ(1).SYMBOL_VAR,4);
    SET_COL(F4,72);
    if Z = Y then
        PUT_LINE(F4,"CHKPT orig. Global CHKPT complete store NST");
    else
        PUT_LINE(F4,"Global CHKPT complete store NST");
    end if;
when others =>
    NULL;
end case;
if NST(NID).NODE_ID = Y then  -- CHKPT orig clears MKR array
    for I in 1..4 loop
        LOC_VAR(NID).CHKPT_TAKEN(I) := 0;
    end loop;
end if;
end CHK_PT_CMPLT_MSG;
end PROCESS;

with FLOAT_INOUT; use FLOAT_INOUT;
with MATH; use MATH;
with RANDOM; use RANDOM;
with PROCESS; use PROCESS;
with TEXT_IO, integer_io;

```

```

use TEXT_IO, integer_io;
package TRAND is

-- Procedure Test Random is a random integer generator
-- which normalizes the random variable to the desired
-- range as indicated by the parameter.

procedure TEST_RANDOM (VAR : in out integer);
end TRAND;

package body TRAND is
procedure TEST_RANDOM (VAR : in out integer) is
    X : float;
begin
    delay 2.0;
    X := RANDOM.NEXT_NUMBER;
    if VAR = 4 then
        VAR := integer(X * 4.0);
        while VAR = 0 loop
            -- X4 must be an integer in the
            -- interval 1-4 (# of node)

            delay 1.0;
            X := RANDOM.NEXT_NUMBER;    -- calls the function
            VAR := integer(X * 4.0);
        end loop;
    else
        if VAR = 12 then
            VAR := integer(X * 12.0);
            while VAR = 0 loop
                -- VAR must be an integer in the
                -- interval 1-12 (# of function)

                delay 1.0;
                X := RANDOM.NEXT_NUMBER; -- calls the function
                VAR := integer(X * 12.0);
            end loop;
        else
            -- get a delay parameter
            VAR := integer(-(1.0/0.5) * NAT_LOG(1.0 - X));
            while VAR = 0 loop
                -- the delay must be an integer
                -- greater than 0.

                delay 1.0;
                X := RANDOM.NEXT_NUMBER; -- calls the function
                VAR := integer(X * 4.0);
            end loop;
        end if;
    end if;
end TEST_RANDOM;
end TRAND;

with DECLARATIONS; use DECLARATIONS;
package COMMNET is
task NETWORK is

```

```

    entry SEND_MSG(M : in MSG_RECORD; NID : in integer);
end;
end COMMNET;

```

```

-- The following package statements create instantiations of the
-- indicated package utilized in the formation of a node.

```

```

with OUTS;
package OUTS1 is new OUTS;
with OUTS;
package OUTS2 is new OUTS;
with OUTS;
package OUTS3 is new OUTS;
with OUTS;
package OUTS4 is new OUTS;
with INS;
package INS1 is new INS;
with INS;
package INS2 is new INS;
with INS;
package INS3 is new INS;
with INS;
package INS4 is new INS;
with SM;
package SM1 is new SM;
with SM;
package SM2 is new SM;
with SM;
package SM3 is new SM;
with SM;
package SM4 is new SM;
with CKPT;
package CKPT1 is new CKPT;
with CKPT;
package CKPT2 is new CKPT;
with CKPT;
package CKPT3 is new CKPT;
with CKPT;
package CKPT4 is new CKPT;
with RL;
package RL1 is new RL;
with RL;
package RL2 is new RL;
with RL;
package RL3 is new RL;
with RL;
package RL4 is new RL;
with text_io; use text_io;
with integer_io; use integer_io;
with number_io; use number_io;
with DECLARATIONS; use DECLARATIONS;

```

```

with PROCESS; use PROCESS;
with TRAND; use TRAND;
with INS1; use INS1;
with INS2; use INS2;
with INS3; use INS3;
with INS4; use INS4;

```

```

package body COMMNET is

```

```

-- The NETWORK task manages a circular queue, receiving messages
-- from the Output Server task and relaying them to all the
-- Input Server tasks. It serves as the communication interface
-- between nodes.

```

```

task body NETWORK is

```

```

W,R : integer;
MGEN : MSG_RECORD;
MSG_PRESENT : boolean := false;
DT : DURATION := 2.57;
begin
  loop
    select
      accept SEND_MSG (M: in MSG_RECORD;NID: in integer) do
        NULL;
      end;
    or
      delay DT;
      MSG_PRESENT := false;
      W := NET_Q.MSG_CNT;
      R := NET_Q.RD_CNT;
      if NET_Q.MSG_TO_SEND then
        if R > W then
          MGEN := NET_Q.MSG_QUE(R);
          R := R + 1;
          if R > Q_SIZE then
            if W < 2 then
              NET_Q.MSG_TO_SEND := false;
              NET_Q.BLOCK_WRITE := false;
            end if;
            NET_Q.RD_CNT := 1;
          else
            NET_Q.RD_CNT := R;
          end if;
        else
          if R < W then
            MGEN := NET_Q.MSG_QUE(R);
            R := R + 1;
            if W = R then
              NET_Q.BLOCK_WRITE := false;
              NET_Q.MSG_TO_SEND := false;
            end if;
          end if;
        end if;
      end if;
    end select;
  end loop;
end NETWORK;

```

```

        NET_Q.RD_CNT := R;
    end if;
    end if;
    MSG_PRESENT := true;
end if;
if MSG_PRESENT then
    for Z in 1..4 loop
        W := LOC_VAR(Z).INQ.MSG_CNT;
        R := LOC_VAR(Z).INQ.RD_CNT;
        if not LOC_VAR(Z).INQ.BLOCK_WRITE then
            if W >= R then
                LOC_VAR(Z).INQ.MSG_QUE(W) := MGEN;
                LOC_VAR(Z).INQ.MSG_TO_SEND := true;
                W := W + 1;
                if W > Q_SIZE then
                    if R < 2 then
                        LOC_VAR(Z).INQ.BLOCK_WRITE := true;
                    end if;
                    LOC_VAR(Z).INQ.MSG_CNT := 1;
                else
                    LOC_VAR(Z).INQ.MSG_CNT := W;
                end if;
            else
                if W < R then
                    LOC_VAR(Z).INQ.MSG_QUE(W) := MGEN;
                    LOC_VAR(Z).INQ.MSG_TO_SEND := true;
                    W := W + 1;
                    if W = R then
                        LOC_VAR(Z).INQ.BLOCK_WRITE := true;
                    end if;
                    LOC_VAR(Z).INQ.MSG_CNT := W;
                end if;
            end if;
        end if;
    end loop; -- end for loop
end if;
end select;
end loop;
end NETWORK;
end COMMNET;

```

```

with DECLARATIONS; use DECLARATIONS;
generic
package INS is
task NODE_INITIALIZER is
    entry BUILD_NODE(NID: in integer);
end;
task INPUT_SERVER is
    entry RECEIVE_MSG(M : in MSG_RECORD; NID : in integer);
end;

```

```
end INS;
```

```
with text_io; use text_io;
with integer_io; use integer_io;
with number_io; use number_io;
with PROCESS; use PROCESS;
with DECLARATIONS; use DECLARATIONS;
with COMMNET; use COMMNET;
with TRAND; use TRAND;
with RL1; use RL1;
with RL2; use RL2;
with RL3; use RL3;
with RL4; use RL4;
with SM1; use SM1;
with SM2; use SM2;
with SM3; use SM3;
with SM4; use SM4;
with CKPT1; use CKPT1;
with CKPT2; use CKPT2;
with CKPT3; use CKPT3;
with CKPT4; use CKPT4;
package body INS is
```

```
-- The NODE_INITIALIZER task is utilized to initialize the node's NST,
-- to be utilized in the simulation process.
```

```
task body NODE_INITIALIZER is
```

```
  x,z : integer;
begin
  loop
    select
      accept BUILD_NODE(NID: in integer) do
        x := 1;
        z := NID;
        -- this loop builds the function location array - this
        -- would normally be initialized by the task allocation
        -- which is only done in psuedo code at this time
        for J in 1..12 loop
          NST(z).COMMON_SECTION.FN_LOC(J) := x;
          x := x + 1;
          if x = 5 then
            x := 1;
          end if;
        end loop;
        NST(z).NODE_ID := NID;
        -- this loop initializes all nodes to the "up" status
        -- within each of the NST's
        for J in 1..4 loop
          NST(z).COMMON_SECTION.NODE_STAT_LD(1,J) := 1;
          NST(z).COMMON_SECTION.NODE_STAT_LD(2,J) := J;
```

```

        end loop;
        NSTBAK(z) := NST(z);           -- make backup copy of NST's
    end;
    or
        terminate;
    end select;
end loop;
end;

```

```

-- The INPUT_SERVER task accepts messages from the NETWORK task.
-- It parses the message fields and calls the appropriate task
-- to process the message.

```

```

task body INPUT_SERVER is
    Z,W,R,i : integer;
    MGEN    : MSG_RECORD;
    PT      : float := 0.0;
    MSG_PRESENT : boolean := false;
    DT      : DURATION := 1.35;
begin
    loop
        select
            -- msg being accepted from the network
            accept RECEIVE_MSG (M: in MSG_RECORD;NID: in integer) do
                Z := NST(NID).NODE_ID;
            end;
        or
            delay DT;
            MSG_PRESENT := false;
            W := LOC_VAR(Z).INQ.MSG_CNT;
            R := LOC_VAR(Z).INQ.RD_CNT;
            if LOC_VAR(Z).INQ.MSG_TO_SEND then
                if R > W then
                    MGEN := LOC_VAR(Z).INQ.MSG_QUE(R);
                    R := R + 1;
                    if R > Q_SIZE then
                        if W < 2 then
                            LOC_VAR(Z).INQ.MSG_TO_SEND := false;
                            LOC_VAR(Z).INQ.BLOCK_WRITE := false;
                        end if;
                        LOC_VAR(Z).INQ.RD_CNT := 1;
                    else
                        LOC_VAR(Z).INQ.RD_CNT := R;
                    end if;
                end if;
            else
                if R < W then
                    MGEN := LOC_VAR(Z).INQ.MSG_QUE(R);
                    R := R + 1;
                    if W = R then
                        LOC_VAR(Z).INQ.BLOCK_WRITE := false;
                        LOC_VAR(Z).INQ.MSG_TO_SEND := false;
                    end if;
                end if;
            end if;
        end select;
    end loop;
end;

```

```

        end if;
        LOC_VAR(Z).INQ.RD_CNT := R;
    end if;
    end if;
    MSG_PRESENT := true;
    end if;
if MSG_PRESENT then
    LOC_VAR(Z).EVNT_CNT := LOC_VAR(Z).EVNT_CNT + 1;
    GET_REAL_TIME(0,PT);
    MGEN.TOR := PT;
    case Z is
        -- call specific section of own node
        when 1 =>
            case MGEN.CNTRL_ACTION is
                when MKR ! CHKPT =>
                    if NST(Z).COMMON_SECTION.NODE_STAT_LD(1,1) = 1 then
                        CKPT1.CHECK_PT.MARKER_MSG(MGEN,1);
                    end if;
                when FNON ! FNOFF =>
                    if NST(Z).COMMON_SECTION.NODE_STAT_LD(1,1) = 1 then
                        RL1.RECONF_LAYER.IS_MSG_IN(MGEN,1);
                    end if;
                when STATUS =>
                    SM1.STATUS_REC.STAT_MSG_REC(MGEN,1);
                when others =>
                    NULL;
            end case;
        when 2 =>
            case MGEN.CNTRL_ACTION is
                when MKR ! CHKPT =>
                    if NST(Z).COMMON_SECTION.NODE_STAT_LD(1,2) = 1 then
                        CKPT2.CHECK_PT.MARKER_MSG(MGEN,2);
                    end if;
                when FNON ! FNOFF =>
                    if NST(Z).COMMON_SECTION.NODE_STAT_LD(1,2) = 1 then
                        RL2.RECONF_LAYER.IS_MSG_IN(MGEN,2);
                    end if;
                when STATUS =>
                    SM2.STATUS_REC.STAT_MSG_REC(MGEN,2);
                when others =>
                    NULL;
            end case;
        when 3 =>
            case MGEN.CNTRL_ACTION is
                when MKR ! CHKPT =>
                    if NST(Z).COMMON_SECTION.NODE_STAT_LD(1,3) = 1 then
                        CKPT3.CHECK_PT.MARKER_MSG(MGEN,3);
                    end if;
                when FNON ! FNOFF =>
                    if NST(Z).COMMON_SECTION.NODE_STAT_LD(1,3) = 1 then
                        RL3.RECONF_LAYER.IS_MSG_IN(MGEN,3);
                    end if;
            end case;
    end case;
end if;

```

```

        when STATUS =>
            SM3.STATUS_REC.STAT_MSG_REC(MGEN,3);
        when others =>
            NULL;
    end case;
when 4 =>
    case MGEN.CNTRL_ACTION is
        when MKR ! CHKPT =>
            if NST(Z).COMMON_SECTION.NODE_STAT_LD(1,4) = 1 then
                CKPT4.CHECK_PT.MARKER_MSG(MGEN,4);
            end if;
        when FNON ! FNOFF =>
            if NST(Z).COMMON_SECTION.NODE_STAT_LD(1,4) = 1 then
                RL4.RECONF_LAYER.IS_MSG_IN(MGEN,4);
            end if;
        when STATUS =>
            SM4.STATUS_REC.STAT_MSG_REC(MGEN,4);
        when others =>
            NULL;
    end case;
when others =>
    NULL;
end case;
    end if;
end select;
end loop;
end;
end INS;

```

```

with DECLARATIONS; use DECLARATIONS;
generic
package OUTS is
task OUTPUT_SERVER is
    entry START_OUTPUT(M : in MSG_RECORD; NID : in integer);
end;
end OUTS;

```

```

with text_io; use text_io;
with integer_io; use integer_io;
with number_io; use number_io;
with PROCESS; use PROCESS;
with TRAND; use TRAND;
with DECLARATIONS; use DECLARATIONS;
with COMMNET; use COMMNET;
package body OUTS is

```

```

-- The OUTPUT_SERVER task relays messages from the various tasks
-- within the node, to the communication layer (NETWORK task).
-- The task serializes a node's messages and ensures that the

```

```

-- NETWORK can accept it.

task body OUTPUT_SERVER is
  Z,W,R : integer;
  MGEN : MSG_RECORD;
  PT : float := 0.0;
  MSG_PRESENT : boolean := false;
  DT : DURATION := 3.83;
begin
  loop
    select
      accept START_OUTPUT(M: in MSG_RECORD;NID: in integer) do
        Z := NST(NID).NODE_ID;
      end;
    or
      delay DT;
      MSG_PRESENT := false;
      W := LOC_VAR(Z).OUTQ.MSG_CNT;
      R := LOC_VAR(Z).OUTQ.RD_CNT;
      if LOC_VAR(Z).OUTQ.MSG_TO_SEND then
        if R > W then
          MGEN := LOC_VAR(Z).OUTQ.MSG_QUE(R);
          R := R + 1;
          if R > Q_SIZE then
            if W < 2 then
              LOC_VAR(Z).OUTQ.MSG_TO_SEND := false;
              LOC_VAR(Z).OUTQ.BLOCK_WRITE := false;
            end if;
            LOC_VAR(Z).OUTQ.RD_CNT := 1;
          else
            LOC_VAR(Z).OUTQ.RD_CNT := R;
          end if;
        else
          if R < W then
            MGEN := LOC_VAR(Z).OUTQ.MSG_QUE(R);
            R := R + 1;
            if W = R then
              LOC_VAR(Z).OUTQ.BLOCK_WRITE := false;
              LOC_VAR(Z).OUTQ.MSG_TO_SEND := false;
            end if;
            LOC_VAR(Z).OUTQ.RD_CNT := R;
          end if;
        end if;
        MSG_PRESENT := true;
      end if;
      if MSG_PRESENT then
        GET_REAL_TIME(0,PT);
        MGEN.TOT := PT;
        LOC_VAR(Z).EVNT_CNT_OUT := LOC_VAR(Z).EVNT_CNT_OUT + 1;
        MGEN.MSG_BODY.UNIQ(1).SYMBOL_VAR := LOC_VAR(Z).EVNT_CNT_OUT;
        W := NET_Q.MSG_CNT;
      end if;
    end select;
  end loop;
end OUTPUT_SERVER;

```

```

R := NET_Q.RD_CNT;
if not NET_Q.BLOCK_WRITE then
  if W >= R then
    NET_Q.MSG_QUE(W) := MGEN;
    NET_Q.MSG_TO_SEND := true;
    W := W + 1;
    if W > Q_SIZE then
      if R < 2 then
        NET_Q.BLOCK_WRITE := true;
      end if;
      NET_Q.MSG_CNT := 1;
    else
      NET_Q.MSG_CNT := W;
    end if;
  else
    if W < R then
      NET_Q.MSG_QUE(W) := MGEN;
      NET_Q.MSG_TO_SEND := true;
      W := W + 1;
      if W = R then
        NET_Q.BLOCK_WRITE := true;
      end if;
      NET_Q.MSG_CNT := W;
    end if;
  end if;
end if;
case Z is
  when 1 =>
    GET_REAL_TIME(1,PT);
    SET_COL(F1,25);
    PUT(F1,"O_S sending ");
    case MGEN.CNTRL_ACTION is
      when MKR =>
        PUT(F1,"MKR msg.");
      when FNON =>
        PUT(F1,"FNON msg.");
      when FNOFF =>
        PUT(F1,"FNOFF to Node #");
        PUT(F1,MGEN.DEST_NODE,1);
      when STATUS =>
        PUT(F1,"STATUS msg.");
      when CHKPT =>
        PUT(F1,"CHKPT msg.");
      when others =>
        NULL;
    end case;
    SET_COL(F1,60);
    PUT(F1,"EVNT #");
    PUT(F1,LOC_VAR(Z).EVNT_CNT_OUT,4);
    NEW_LINE(F1);
  when 2 =>

```

```

GET_REAL_TIME(2,PT);
SET_COL(F2,25);
PUT(F2,"O_S sending ");
case MGEN.CNTRL_ACTION is
  when MKR =>
    PUT(F2,"MKR msg.");
  when FNON =>
    PUT(F2,"FNON msg.");
  when FNOFF =>
    PUT(F2,"FNOFF to Node #");
    PUT(F2,MGEN.DEST_NODE,1);
  when STATUS =>
    PUT(F2,"STATUS msg.");
  when CHKPT =>
    PUT(F2,"CHKPT msg.");
  when others =>
    NULL;
end case;
SET_COL(F2,60);
PUT(F2,"EVNT #");
PUT(F2,LOC_VAR(Z).EVNT_CNT_OUT,4);
NEW_LINE(F2);
when 3 =>
  GET_REAL_TIME(3,PT);
  SET_COL(F3,25);
  PUT(F3,"O_S sending ");
  case MGEN.CNTRL_ACTION is
    when MKR =>
      PUT(F3,"MKR msg.");
    when FNON =>
      PUT(F3,"FNON msg.");
    when FNOFF =>
      PUT(F3,"FNOFF to Node #");
      PUT(F3,MGEN.DEST_NODE,1);
    when STATUS =>
      PUT(F3,"STATUS msg.");
    when CHKPT =>
      PUT(F3,"CHKPT msg.");
    when others =>
      NULL;
  end case;
  SET_COL(F3,60);
  PUT(F3,"EVNT #");
  PUT(F3,LOC_VAR(Z).EVNT_CNT_OUT,4);
  NEW_LINE(F3);
when 4 =>
  GET_REAL_TIME(4,PT);
  SET_COL(F4,25);
  PUT(F4,"O_S sending ");
  case MGEN.CNTRL_ACTION is
    when MKR =>

```

```

        PUT(F4,"MKR msg.");
    when FNON =>
        PUT(F4,"FNON msg.");
    when FNOFF =>
        PUT(F4,"FNOFF to Node #");
        PUT(F4,MGEN.DEST_NODE,1);
    when STATUS =>
        PUT(F4,"STATUS msg.");
    when CHKPT =>
        PUT(F4,"CHKPT msg.");
    when others =>
        NULL;
    end case;
    SET_COL(F4,60);
    PUT(F4,"EVNT #");
    PUT(F4,LOC_VAR(Z).EVNT_CNT_OUT,4);
    NEW_LINE(F4);
    when others =>
        NULL;
    end case;
end if;
end select;
end loop;
end;
end OUTS;

with DECLARATIONS; use DECLARATIONS;
generic
package CKPT is
task CHECK_PT is
    entry MARKER_MSG(M : in MSG_RECORD; NID : in integer);
    entry CHKPT_COMP(M : in MSG_RECORD; NID : in integer);
end;
task EVENT_CNT is
    entry EVNT_CNT_FULL(NID : in integer);
end;
end CKPT;

with text_io; use text_io;
with integer_io; use integer_io;
with number_io; use number_io;
with PROCESS; use PROCESS;
with DECLARATIONS; use DECLARATIONS;
with COMMNET; use COMMNET;
package body CKPT is

-- The CHECK_PT task is called by the INPUT_SERVER when a
-- marker (MKR) or checkpoint complete (CHKPT) message is
-- received. This task calls MKR_MSG or CHK_PT_CMPLT_MSG

```

-- respectfully, for further processing of the messages.

```
task body CHECK_PT is
  MGEN : MSG_RECORD;
  FLG  : boolean;
  Z,W,R : integer;
begin
  loop
    select
      accept MARKER_MSG (M: in MSG_RECORD;NID: in integer) do
        Z := NST(NID).NODE_ID;
        MGEN := M;
        FLG := FALSE;
        case M.CNTRL_ACTION is
          when MKR =>
            PROCESS.MKR_MSG(MGEN, Z, FLG);
            if FLG then
              W := LOC_VAR(Z).OUTQ.MSG_CNT;
              R := LOC_VAR(Z).OUTQ.RD_CNT;
              if not LOC_VAR(Z).OUTQ.BLOCK_WRITE then
                if W >= R then
                  LOC_VAR(Z).OUTQ.MSG_QUE(W) := MGEN;
                  LOC_VAR(Z).OUTQ.MSG_TO_SEND := true;

                  W := W + 1;
                  if W > Q_SIZE then
                    if R < 2 then
                      LOC_VAR(Z).OUTQ.BLOCK_WRITE := true;
                    end if;
                    LOC_VAR(Z).OUTQ.MSG_CNT := 1;
                  else
                    LOC_VAR(Z).OUTQ.MSG_CNT := W;
                  end if;
                else
                  if W < R then
                    LOC_VAR(Z).OUTQ.MSG_QUE(W) := MGEN;
                    LOC_VAR(Z).OUTQ.MSG_TO_SEND := true;
                    W := W + 1;
                    if W = R then
                      LOC_VAR(Z).OUTQ.BLOCK_WRITE := true;
                    end if;
                    LOC_VAR(Z).OUTQ.MSG_CNT := W;
                  end if;
                end if;
              end if;
            end if;
          when CHKPT =>
            Z := NST(NID).NODE_ID;
            PROCESS.CHK_PT_CMPLT_MSG(M,Z);
          when others =>
            null;
        end case;
      end select;
    end loop;
  end task;
```

```

        end case;
    end;
    or
        terminate;
    end select;
end loop;
end;

```

```

-- The EVENT_CNT task monitors the events at a node and originates
-- the checkpoint process once a predetermined number of events has
-- occurred.

```

```

task body EVENT_CNT is
    MGEN : MSG_RECORD;
    FLG  : boolean;
    Z,W,R : integer;
    CNT   : integer := 10;
    PT    : float := 0.0;
begin
    loop
        select
            accept EVNT_CNT_FULL(NID : in integer) do
                Z := NST(NID).NODE_ID; -- initialize for simulation
                CNT := CNT * NID;
            end;
        or
            delay 33.7;
            GET_REAL_TIME(0,PT);
            if LOC_VAR(Z).CHKPT_ORIG and
                PT-LOC_VAR(Z).CHKPT_TIMER > 68.1 then
                LOC_VAR(Z).LOCAL_CHKPT := false;
                LOC_VAR(Z).FIRST_MKR := FALSE;
                LOC_VAR(Z).CHKPT_ORIG := FALSE;
                LOC_VAR(Z).CHKPT_TIMER := PT;
                for I in 1..4 loop
                    LOC_VAR(Z).CHKPT_TAKEN(I) := 0;
                end loop;
                case Z is
                    when 1 =>
                        GET_REAL_TIME(1,PT);
                        SET_COL(F1,72);
                        PUT_LINE(F1,"CHKPT unsuccessful. Restarting CHKPT");
                    when 2 =>
                        GET_REAL_TIME(2,PT);
                        SET_COL(F2,72);
                        PUT_LINE(F2,"CHKPT unsuccessful. Restarting CHKPT");
                    when 3 =>
                        GET_REAL_TIME(3,PT);
                        SET_COL(F3,72);
                        PUT_LINE(F3,"CHKPT unsuccessful. Restarting CHKPT");
                    when 4 =>

```

```

        GET_REAL_TIME(4,PT);
        SET_COL(F4,72);
        PUT_LINE(F4,"CHKPT unsuccessful. Restarting CHKPT");
        when others =>
            NULL;
        end case;
    end if;
    if LOC_VAR(Z).EVNT_CNT > CNT and
       not LOC_VAR(Z).LOCAL_CHKPT then
        ST(Z) := NST(Z);
        MGEN.ORIG_FN_NODE := Z;
        MGEN.MSG_KIND := control;
        MGEN.CNTRL_ACTION := MKR;
        LOC_VAR(Z).EVNT_CNT := 0;
        MGEN.MSG_BODY.UNIQ := NST(Z).UNIQUE_SECTION(Z);
        LOC_VAR(Z).LOCAL_CHKPT := true;
        LOC_VAR(Z).CHKPT_TIMER := PT;
        W := LOC_VAR(Z).OUTQ.MSG_CNT;
        R := LOC_VAR(Z).OUTQ.RD_CNT;
        if not LOC_VAR(Z).OUTQ.BLOCK_WRITE then
            if W >= R then
                LOC_VAR(Z).OUTQ.MSG_QUE(W) := MGEN;
                LOC_VAR(Z).OUTQ.MSG_TO_SEND := true;
                W := W + 1;
                if W > Q_SIZE then
                    if R < 2 then
                        LOC_VAR(Z).OUTQ.BLOCK_WRITE := true;
                    end if;
                    LOC_VAR(Z).OUTQ.MSG_CNT := 1;
                else
                    LOC_VAR(Z).OUTQ.MSG_CNT := W;
                end if;
            else
                if W < R then
                    LOC_VAR(Z).OUTQ.MSG_QUE(W) := MGEN;
                    LOC_VAR(Z).OUTQ.MSG_TO_SEND := true;
                    W := W + 1;
                    if W = R then
                        LOC_VAR(Z).OUTQ.BLOCK_WRITE := true;
                    end if;
                    LOC_VAR(Z).OUTQ.MSG_CNT := W;
                end if;
            end if;
        end if;
    end if;
end select;
end loop;
end;
end CKPT;

```

```

with DECLARATIONS; use DECLARATIONS;
generic
package RL is
task RECONF_LAYER is
    entry IS_MSG_IN(M : in MSG_RECORD; NID : in integer);
end;
end RL;

with text_io; use text_io;
with integer_io; use integer_io;
with number_io; use number_io;
with PROCESS; use PROCESS;
with DECLARATIONS; use DECLARATIONS;
with COMMNET; use COMMNET;
package body RL is

-- The RECONF_LAYER task is called by the INPUT_SERVER task
-- to process both FNON and FNOFF messages.
-- It calls procedures FN_ON_REC nad FN_OFF_REC to process
-- these types of messages.

task body RECONF_LAYER is
    -- specific calls may need to pass a msg back out
    -- if so, set the -- msg flag

    MSG_FLAG : boolean := FALSE;
    MGEN      : MSG_RECORD;
    Z,C,W,R : integer;
begin
    loop
        select
            -- input server call R_L with a msg to send
            accept IS_MSG_IN (M: in MSG_RECORD; NID : in integer) do
                Z := NST(NID).NODE_ID;
                MGEN := M;
            -- the R_L determines whether a fn needs to be started or terminated
            -- in the active fn queue - it will notify the application layer to
            -- take the required action
            case M.CNTRL_ACTION is
                when FNON =>
                    PROCESS.FN_ON_MSG(M, NID);
                when FNOFF =>
                    PROCESS.FN_OFF_MSG(MGEN, Z, MSG_FLAG);
                    if MSG_FLAG then
                        -- msg needs to go to O_S but
                        -- will add msg to out queue
                        -- to get processed by O_S
                        W := LOC_VAR(Z).OUTQ.MSG_CNT;
                        R := LOC_VAR(Z).OUTQ.RD_CNT;
                        if not LOC_VAR(Z).OUTQ.BLOCK_WRITE then
                            if W >= R then

```

```

        LOC_VAR(Z).OUTQ.MSG_QUE(W) := MGEN;
        LOC_VAR(Z).OUTQ.MSG_TO_SEND := true;
        W := W + 1;
        if W > Q_SIZE then
            if R < 2 then
                LOC_VAR(Z).OUTQ.BLOCK_WRITE := true;
            end if;
            LOC_VAR(Z).OUTQ.MSG_CNT := 1;
        else
            LOC_VAR(Z).OUTQ.MSG_CNT := W;
        end if;
    else
        if W < R then
            LOC_VAR(Z).OUTQ.MSG_QUE(W) := MGEN;
            LOC_VAR(Z).OUTQ.MSG_TO_SEND := true;
            W := W + 1;
            if W = R then
                LOC_VAR(Z).OUTQ.BLOCK_WRITE := true;
            end if;
            LOC_VAR(Z).OUTQ.MSG_CNT := W;
        end if;
    end if;
    end if;
    MSG_FLAG := FALSE;
end if;
when others =>
    NULL;
end case;
end;
or
    terminate;
end select;
end loop;
end;
end RL;

```

```

with DECLARATIONS; use DECLARATIONS;
generic
package SM is
task STATUS_REC is
    entry STAT_MSG_REC(M : in MSG_RECORD; NID : in integer);
end;
task STATUS_BDCST is
    entry STAT_BDCST_CHK(NID : in integer);
end;
end SM;

```

```

with FLOAT_INOUT; use FLOAT_INOUT;
with text_io; use text_io;

```

```

with integer_io; use integer_io;
with number_io; use number_io;
with PROCESS; use PROCESS;
with DECLARATIONS; use DECLARATIONS;
with COMMNET; use COMMNET;
package body SM is

-- The STATUS_BDCST task generates periodic status messages
-- for the node. Also incorporated in this task is the
-- Timeout routine , which implements node failure detection.

task body STATUS_BDCST is
    MGEN : MSG_RECORD;
    FLG  : boolean;
    SB   : boolean := false;
    Z,C,W,R : integer;
    PT   : float := 0.0;
begin
    loop
        select
            accept STAT_BDCST_CHK(NID: in integer) do
                Z := NST(NID).NODE_ID;
            end;
        or
            delay 15.0;
            GET_REAL_TIME(0,PT);
            for I in 1..4 loop
                if NST(Z).COMMON_SECTION.NODE_STAT_LD(1,I) = 1 and
                    PT - LOC_VAR(Z).TIMER(I) > 65.0 then
                    NST(Z).COMMON_SECTION.NODE_STAT_LD(1,I) := 0;
                    case Z is
                        when 1 =>
                            GET_REAL_TIME(1,PT);
                            SET_COL(F1,25);
                            PUT(F1,"S_M detects FAILURE on  Node #");
                            PUT(F1,I,1);
                            SET_COL(F1,72);
                            PUT_LINE(F1,"Notify NF task.");
                        when 2 =>
                            GET_REAL_TIME(2,PT);
                            SET_COL(F2,25);
                            PUT(F2,"S_M detects FAILURE on  Node #");
                            PUT(F2,I,1);
                            SET_COL(F2,72);
                            PUT_LINE(F2,"Notify NF task.");
                        when 3 =>
                            GET_REAL_TIME(3,PT);
                            SET_COL(F3,25);
                            PUT(F3,"S_M detects FAILURE on  Node #");
                            PUT(F3,I,1);
                            SET_COL(F3,72);
                    end case;
                end if;
            end loop;
        end select;
    end loop;
end STATUS_BDCST;

```

```

        PUT_LINE(F3,"Notify NF task.");
    when 4 =>
        GET_REAL_TIME(4,PT);
        SET_COL(F4,25);
        PUT(F4,"S_M detects FAILURE on  Node #");
        PUT(F4,I,1);
        SET_COL(F4,72);
        PUT_LINE(F4,"Notify NF task.");
    when others =>
        NULL;
    end case;
end if;
end loop;
if NST(Z).COMMON_SECTION.NODE_STAT_LD(1,Z) = 1
    and not FAILED_NODE(Z) then
    if PT - LOC_VAR(Z).TIMER(Z) > 44.0 then
        MGEN.DEST_NODE := 1;
        MGEN.DEST_FUNC := Z;
        MGEN.CNTRL_ACTION := STATUS;
        MGEN.ORIG_FN_NODE := Z;
        MGEN.MSG_KIND := control;
        W := LOC_VAR(Z).OUTQ.MSG_CNT;
        R := LOC_VAR(Z).OUTQ.RD_CNT;
        if not LOC_VAR(Z).OUTQ.BLOCK_WRITE then
            if W >= R then
                LOC_VAR(Z).OUTQ.MSG_QUE(W) := MGEN;
                LOC_VAR(Z).OUTQ.MSG_TO_SEND := true;
                W := W + 1;
                if W > Q_SIZE then
                    if R < 2 then
                        LOC_VAR(Z).OUTQ.BLOCK_WRITE := true;
                    end if;
                    LOC_VAR(Z).OUTQ.MSG_CNT := 1;
                else
                    LOC_VAR(Z).OUTQ.MSG_CNT := W;
                end if;
            else
                if W < R then
                    LOC_VAR(Z).OUTQ.MSG_QUE(W) := MGEN;
                    LOC_VAR(Z).OUTQ.MSG_TO_SEND := true;
                    W := W + 1;
                    if W = R then
                        LOC_VAR(Z).OUTQ.BLOCK_WRITE := true;
                    end if;
                    LOC_VAR(Z).OUTQ.MSG_CNT := W;
                end if;
            end if;
        end if;
    end if;
end if;
end select;

```

```

    end loop;
end;

```

```

-- The STATUS_REC task is called by the INPUT_SERVER when a
-- status message is received. In turn this task calls the
-- STATUS_MSG procedure for further processing.

```

```

task body STATUS_REC is
    MGEN : MSG_RECORD;
    FLG  : boolean;
    SB   : boolean := false;
    Z,C,W,R : integer;
    PT   : float := 0.0;
begin
    loop
        select
            accept STAT_MSG_REC (M:in MSG_RECORD;NID: in integer) do
                Z := NST(NID).NODE_ID;
                MGEN := M;
                FLG := FALSE;
                LOC_VAR(Z).TIMER(MGEN.ORIG_FN_NODE) := M.TOT;
                PROCESS.STAT_MSG(MGEN, Z, FLG);
                if FLG then
                    W := LOC_VAR(Z).OUTQ.MSG_CNT;
                    R := LOC_VAR(Z).OUTQ.RD_CNT;
                    if not LOC_VAR(Z).OUTQ.BLOCK_WRITE then
                        if W >= R then
                            LOC_VAR(Z).OUTQ.MSG_QUE(W) := MGEN;
                            LOC_VAR(Z).OUTQ.MSG_TO_SEND := true;
                            W := W + 1;
                            if W > Q_SIZE then
                                if R < 2 then
                                    LOC_VAR(Z).OUTQ.BLOCK_WRITE := true;
                                end if;
                                LOC_VAR(Z).OUTQ.MSG_CNT := 1;
                            else
                                LOC_VAR(Z).OUTQ.MSG_CNT := W;
                            end if;
                        else
                            if W < R then
                                LOC_VAR(Z).OUTQ.MSG_QUE(W) := MGEN;
                                LOC_VAR(Z).OUTQ.MSG_TO_SEND := true;
                                W := W + 1;
                                if W = R then
                                    LOC_VAR(Z).OUTQ.BLOCK_WRITE := true;
                                end if;
                                LOC_VAR(Z).OUTQ.MSG_CNT := W;
                            end if;
                        end if;
                    end if;
                end if;
            end if;
        end select;
    end loop;
end;

```

```

        end if;
    end;
    or
        terminate;
    end select;
end loop;
end;
end SM;

```

```

with DECLARATIONS; use DECLARATIONS;
package FP is
task EVENT_MAKER is
    entry NEW_EVENT(NID: in integer);
end;
end FP;

```

```

with FLOAT_INOUT; use FLOAT_INOUT;
with text_io; use text_io;
with integer_io; use integer_io;
with number_io; use number_io;
with TRAND; use TRAND;
with calendar; use calendar;
with DECLARATIONS; use DECLARATIONS;
with PROCESS; use PROCESS;
package body FP is

```

```

-- The EVENT_MAKER task is utilized to simulate an actual
--distributed processing system.

```

```

task body EVENT_MAKER is
MGEN,outmsg : MSG_RECORD;
x,Z,W,R : integer;
N : integer := 0;
EN,ON,DN : integer;
MSG_BUF_EMPTY : boolean := false;
MSG_PRESENT : boolean := false;
PT : float := 0.0;
ST : DURATION := 63.15;
begin    -- begin Front_End Processor
    loop
        select
            accept NEW_EVENT(NID: in integer) do
                Z := NID;
            end;
        or
            delay ST;
            N := N + 1;
            MSG_PRESENT := false;
            EN := 12;
            TRAND.TEST_RANDOM(EN);

```

```

EN := EN mod 2;
case EN is
  when 1 =>
    MSG_PRESENT := true;
    outmsg.CNTRL_ACTION := FNOFF;
    ON := 4;
    TRAND.TEST_RANDOM(ON);--get an active random orig node

    WHILE NST(Z).COMMON_SECTION.NODE_STAT_LD(1,ON) = 0 loop
      delay 2.0;
      CN := 4;
      TRAND.TEST_RANDOM(ON);
    end loop; -- end while loop
    outmsg.ORIG_FN_NODE := ON;
    DN := 4;
    TRAND.TEST_RANDOM(DN);--get an active random dest
      --node that is not = to the orig node
    WHILE NST(Z).COMMON_SECTION.NODE_STAT_LD(1,DN) = 0
      or DN = ON loop
      delay 2.0;
      DN := 4;
      TRAND.TEST_RANDOM(DN);
    end loop; -- end while loop
    outmsg.DEST_NODE := DN;
    x := 1;      -- get an active fn from orig. node
    while NST(Z).COMMON_SECTION.FN_LOC(x) /= ON
      and x < 13 loop
      x := x + 1;
    end loop;
    if x < 13 then
      outmsg.DEST_FUNC := x;
    else
      MSG_PRESENT := false;
    end if;
    outmsg.MSG_BODY.UNIQ(1).REGISTER_VAL := DN;
    outmsg.MSG_KIND := CONTROL;
  when 0 =>
    ON := 4;
    TRAND.TEST_RANDOM(ON);
    WHILE NST(Z).COMMON_SECTION.NODE_STAT_LD(1,ON)=0 loop
      ON := 4;
      TRAND.TEST_RANDOM(ON);
    end loop; -- end while loop
    if not FAILED_NODE(ON) then
      FAILED_NODE(ON) := true;
    end if;
    case ON is
      when 1 =>
        GET_REAL_TIME(1,PT);
        SET_COL(F1,25);
        PUT_LINE(F1,"FP generating Node FAILURE");

```

```

        when 2 =>
            GET_REAL_TIME(2,PT);
            SET_COL(F2,25);
            PUT_LINE(F2,"FP generating Node FAILURE");
        when 3 =>
            GET_REAL_TIME(3,PT);
            SET_COL(F3,25);
            PUT_LINE(F3,"FP generating Node FAILURE");
        when 4 =>
            GET_REAL_TIME(4,PT);
            SET_COL(F4,25);
            PUT_LINE(F4,"FP generating Node FAILURE");
        when others =>
            NULL;
    end case;
    MSG_PRESENT := false;
    when others =>
        null;
    end case;
    if MSG_PRESENT then
        MGEN := outmsg;
        Z := MGEN.ORIG_FN_NODE;
        W := LOC_VAR(Z).OUTQ.MSG_CNT;
        R := LOC_VAR(Z).OUTQ.RD_CNT;
        if not LOC_VAR(Z).OUTQ.BLOCK_WRITE then
            LOC_VAR(Z).OUTQ.MSG_QUE(W) := MGEN;
            LOC_VAR(Z).OUTQ.MSG_TO_SEND := true;
            W := W + 1;
            if W > Q_SIZE then
                LOC_VAR(Z).OUTQ.MSG_CNT := 1;
            end if;
            if W = R then
                LOC_VAR(Z).OUTQ.BLOCK_WRITE := true;
            else
                LOC_VAR(Z).OUTQ.MSG_CNT := W;
            end if;
        end if;
    end if;
end select;
end loop;
end;
end FP;

```

```

with text_io; use text_io;
with integer_io; use integer_io;
with number_io; use number_io;
with FLOAT_INOUT; use FLOAT_INOUT;
with calendar; use calendar;
with DECLARATIONS; use DECLARATIONS;
with PROCESS; use PROCESS;

```

```

with COMMNET; use COMMNET;
with FP; use FP;
with OUTS1; use OUTS1;
with OUTS2; use OUTS3;
with OUTS3; use OUTS3;
with OUTS4; use OUTS4;
with INS1; use INS1;
with INS2; use INS2;
with INS3; use INS3;
with INS4; use INS4;
with SM1; use SM1;
with SM2; use SM2;
with SM3; use SM3;
with SM4; use SM4;
with RL1; use RL1;
with RL2; use RL2;
with RL3; use RL3;
with RL4; use RL4;
with CKPT1; use CKPT1;
with CKPT2; use CKPT2;
with CKPT3; use CKPT3;
with CKPT4; use CKPT4;

```

```

-- The procedure FEP is utilized to open individual
-- output files for each node. It also initiates each node's
-- NST for simulation purposes and assigns each task its
-- node identification number.

```

```

procedure FEP is
  MGEN,outmsg : MSG_RECORD;
  Z,W,R : integer;
  PT : float := 0.0;
begin
  -- begin Front_End Processor
  OPEN(F1,MODE=>OUT_FILE,NAME=>"NOUT1");
  OPEN(F2,MODE=>OUT_FILE,NAME=>"NOUT2");
  OPEN(F3,MODE=>OUT_FILE,NAME=>"NOUT3");
  OPEN(F4,MODE=>OUT_FILE,NAME=>"NOUT4");
  INS1.NODE_INITIALIZER.BUILD_NODE(1);
  INS2.NODE_INITIALIZER.BUILD_NODE(2);
  INS3.NODE_INITIALIZER.BUILD_NODE(3);
  INS4.NODE_INITIALIZER.BUILD_NODE(4);
  GET_REAL_TIME(0,PT);
  for L in 1..4 loop
    for N in 1..4 loop --initialize periodic time array
      --of each node
      LOC_VAR(L).TIMER(N) := PT + float(N * 0.1);
    end loop;
    case L is
      -- give identity to tasks within packages
      when 1 =>
        SM1.STATUS_BDCST.STAT_BDCST_CHK(1);
        CKPT1.EVENT_CNT.EVNT_CNT_FULL(1);

```

```

        INS1.INPUT_SERVER.RECEIVE_MSG(outmsg,1);
        OUTS1.OUTPUT_SERVER.START_OUTPUT(outmsg,1);
    when 2 =>
        SM2.STATUS_BDCST.STAT_BDCST_CHK(2);
        CKPT2.EVENT_CNT.EVNT_CNT_FULL(2);
        INS2.INPUT_SERVER.RECEIVE_MSG(outmsg,2);
        OUTS2.OUTPUT_SERVER.START_OUTPUT(outmsg,2);
    when 3 =>
        SM3.STATUS_BDCST.STAT_BDCST_CHK(3);
        CKPT3.EVENT_CNT.EVNT_CNT_FULL(3);
        INS3.INPUT_SERVER.RECEIVE_MSG(outmsg,3);
        OUTS3.OUTPUT_SERVER.START_OUTPUT(outmsg,3);
    when 4 =>
        SM4.STATUS_BDCST.STAT_BDCST_CHK(4);
        CKPT4.EVENT_CNT.EVNT_CNT_FULL(4);
        INS4.INPUT_SERVER.RECEIVE_MSG(outmsg,4);
        OUTS4.OUTPUT_SERVER.START_OUTPUT(outmsg,4);
    when others =>
        NULL;
    end case;
end loop;
FP.EVENT_MAKER.NEW_EVENT(1);
end FEP;

```

APPENDIX B: SIMULATION OUTPUT

```

/* The output is given in its entirety. The specific events */
/* pertaining to this thesis have been provided in timing */
/* diagrams listed in previous chapters */
/* The first column indicates the time of occurrence. Column two */
/* specifies which node is active, and column three indicates what */
/* event is taking place. Column four designates the event number */
/* of the node which sent the message. The node which sent the */
/* message is listed in the previous column. The last column, */
/* which appears on a new line, explains what action is done at */
/* the active node (column two). */

```

39429.76000	Node #1	O_S sending STATUS msg.	EVNT #	1
39432.64000	Node #1	S_M rcvd PERIODIC from Node #1	EVNT #	1
		Reset Timer element of Node #1		
39435.37000	Node #1	S_M rcvd PERIODIC from Node #2	EVNT #	1
		Reset Timer element of Node #2		
39438.11000	Node #1	S_M rcvd PERIODIC from Node #3	EVNT #	1
		Reset Timer element of Node #3		
39440.85000	Node #1	S_M rcvd PERIODIC from Node #4	EVNT #	1
		Reset Timer element of Node #4		
39450.88000	Node #1	FP generating Node FAILURE		
39492.55000	Node #1	S_M rcvd PERIODIC from Node #3	EVNT #	2
		Reset Timer element of Node #3		
39495.29000	Node #1	S_M rcvd PERIODIC from Node #4	EVNT #	2
		Reset Timer element of Node #4		
39498.03000	Node #1	S_M rcvd PERIODIC from Node #2	EVNT #	2
		Reset Timer element of Node #2		
39503.76000	Node #1	S_M detects FAILURE on Node #1		
		Notify NF task.		
39551.09000	Node #1	S_M rcvd PERIODIC from Node #3	EVNT #	3
		Reset Timer element of Node #3		
39552.63000	Node #1	O_S sending STATUS msg.	EVNT #	2
39553.81000	Node #1	S_M rcvd PERIODIC from Node #4	EVNT #	4
		Reset Timer element of Node #4		
39556.53000	Node #1	S_M rcvd PERIODIC from Node #2	EVNT #	4
		Reset Timer element of Node #2		
39559.25000	Node #1	S_M rcvd APERIODIC from Node #1	EVNT #	2
		This is the recovering node.		
39561.97000	Node #1	S_M rcvd APERIODIC from Node #3	EVNT #	4
		This is the recovering node.		
39564.69000	Node #1	S_M rcvd APERIODIC from Node #4	EVNT #	5
		This is the recovering node.		
39567.41000	Node #1	S_M rcvd APERIODIC from Node #2	EVNT #	5
		Recovery complete - send PERIODIC msg.		
39567.99000	Node #1	O_S sending STATUS msg.	EVNT #	3

39570.13000	Node #1	S_M rcvd PERIODIC from Node #1 Reset Timer element of Node #1	EVNT # 3
39587.19000	Node #1	O_S sending MKR msg.	EVNT # 4
39590.53000	Node #1	C_P rcvd MKR from Node #1 I originated CHKPT. Not all MKRs yet rcvd.	EVNT # 4
39593.25000	Node #1	C_P rcvd MKR from Node #3 I originated CHKPT. Not all MKRs yet rcvd.	EVNT # 5
39594.87000	Node #1	O_S sending FNOFF to Node #2	EVNT # 5
39595.97000	Node #1	C_P rcvd MKR from Node #4 I originated CHKPT. Not all MKRs yet rcvd.	EVNT # 6
39598.69000	Node #1	C_P rcvd MKR from Node #2 MKRs rcvd from all nodes. Send CHKPT_COMP	EVNT # 6
39598.71000	Node #1	O_S sending CHKPT msg.	EVNT # 6
39600.05000	Node #1	R_L rcvd FN_OFF from Node #1 No further action required ATT.	EVNT # 5
39602.77000	Node #1	C_P rcvd CHKPT from Node #1 CHKPT orig. Global CHKPT complete store NST	EVNT # 6
39605.49000	Node #1	R_L rcvd FN_ON from Node #2 I am the deactivating node and changing NST 2 2 3 2 1 2 3 4 1 2 3 4	EVNT # 7
39610.93000	Node #1	S_M rcvd PERIODIC from Node #3 Reset Timer element of Node #3	EVNT # 6
39625.58000	Node #1	O_S sending STATUS msg.	EVNT # 7
39625.89000	Node #1	S_M rcvd PERIODIC from Node #1 Reset Timer element of Node #1	EVNT # 7
39628.61000	Node #1	S_M rcvd PERIODIC from Node #4 Reset Timer element of Node #4	EVNT # 7
39631.33000	Node #1	S_M rcvd PERIODIC from Node #2 Reset Timer element of Node #2	EVNT # 8
39429.76000	Node #2	O_S sending STATUS msg.	EVNT # 1
39432.66000	Node #2	S_M rcvd PERIODIC from Node #1 Reset Timer element of Node #1	EVNT # 1
39435.39000	Node #2	S_M rcvd PERIODIC from Node #2 Reset Timer element of Node #2	EVNT # 1
39438.13000	Node #2	S_M rcvd PERIODIC from Node #3 Reset Timer element of Node #3	EVNT # 1
39440.87000	Node #2	S_M rcvd PERIODIC from Node #4 Reset Timer element of Node #4	EVNT # 1
39491.22000	Node #2	O_S sending STATUS msg.	EVNT # 2
39492.57000	Node #2	S_M rcvd PERIODIC from Node #3 Reset Timer element of Node #3	EVNT # 2
39495.31000	Node #2	S_M rcvd PERIODIC from Node #4 Reset Timer element of Node #4	EVNT # 2
39498.05000	Node #2	S_M rcvd PERIODIC from Node #2 Reset Timer element of Node #2	EVNT # 2
39503.76000	Node #2	S_M detects FAILURE on Node #1 Notify NF task.	
39523.90000	Node #2	R_L rcvd FN_OFF from Node #4	EVNT # 3

		FN_ON sent to activate FN # 4		
39525.78000	Node #2	O_S sending FNON msg.	EVNT #	3
39528.00000	Node #2	R_L rcvd FN_ON from Node #2	EVNT #	3
		I am the activating node and changing NST.		
		1 2 3 2 1 2 3 4 1 2 3 4		
39548.80900	Node #2	O_S sending STATUS msg.	EVNT #	4
39551.17900	Node #2	S_M rcvd PERIODIC from Node #3	EVNT #	3
		Reset Timer element of Node #3		
39553.91000	Node #2	S_M rcvd PERIODIC from Node #4	EVNT #	4
		Reset Timer element of Node #4		
39556.64000	Node #2	S_M rcvd PERIODIC from Node #2	EVNT #	4
		Reset Timer element of Node #2		
39559.37000	Node #2	S_M rcvd APERIODIC from Node #1	EVNT #	2
		Sending APERIODIC with NST unique sections.		
39560.32000	Node #2	O_S sending STATUS msg.	EVNT #	5
39562.11000	Node #2	S_M rcvd APERIODIC from Node #3	EVNT #	4
		APERIODIC response already sent, no action.		
39564.84000	Node #2	S_M rcvd APERIODIC from Node #4	EVNT #	5
		APERIODIC response already sent, no action.		
39567.57000	Node #2	S_M rcvd APERIODIC from Node #2	EVNT #	5
		APERIODIC response already sent, no action.		
39570.30000	Node #2	S_M rcvd PERIODIC from Node #1	EVNT #	3
		Reset Timer element of Node #1		
39590.71000	Node #2	C_P rcvd MKR from Node #1	EVNT #	4
		Local CHKPT already conducted. Store UNIQ.		
39591.04000	Node #2	O_S sending MKR msg.	EVNT #	6
39593.44000	Node #2	C_P rcvd MKR from Node #3	EVNT #	5
		Local CHKPT already conducted. Store UNIQ.		
39596.17000	Node #2	C_P rcvd MKR from Node #4	EVNT #	6
		Local CHKPT already conducted. Store UNIQ.		
39597.54000	Node #2	C_P rcvd MKR from Node #2	EVNT #	6
		Local CHKPT already conducted. Store UNIQ.		
39600.27000	Node #2	R_L rcvd FN_OFF from Node #1	EVNT #	5
		FN_ON sent to activate FN # 1		
39602.54000	Node #2	O_S sending FNON msg.	EVNT #	7
39603.00000	Node #2	C_P rcvd CHKPT from Node #1	EVNT #	6
		Global CHKPT complete store NST		
39605.74000	Node #2	R_L rcvd FN_ON from Node #2	EVNT #	7
		I am the activating node and changing NST.		
		2 2 3 2 1 2 3 4 1 2 3 4		
39611.20000	Node #2	S_M rcvd PERIODIC from Node #3	EVNT #	6
		Reset Timer element of Node #3		
39625.59000	Node #2	O_S sending STATUS msg.	EVNT #	8
39626.17000	Node #2	S_M rcvd PERIODIC from Node #1	EVNT #	7
		Reset Timer element of Node #1		
39628.90000	Node #2	S_M rcvd PERIODIC from Node #4	EVNT #	7
		Reset Timer element of Node #4		
39631.63000	Node #2	S_M rcvd PERIODIC from Node #2	EVNT #	8
		Reset Timer element of Node #2		

39429.77000	Node #3	O_S sending STATUS msg.	EVNT #	1
39432.65000	Node #3	S_M rcvd PERIODIC from Node #1	EVNT #	1
		Reset Timer element of Node #1		
39435.37900	Node #3	S_M rcvd PERIODIC from Node #2	EVNT #	1
		Reset Timer element of Node #2		
39438.12000	Node #3	S_M rcvd PERIODIC from Node #3	EVNT #	1
		Reset Timer element of Node #3		
39440.86000	Node #3	S_M rcvd PERIODIC from Node #4	EVNT #	1
		Reset Timer element of Node #4		
39491.19000	Node #3	O_S sending STATUS msg.	EVNT #	2
39492.56000	Node #3	S_M rcvd PERIODIC from Node #3	EVNT #	2
		Reset Timer element of Node #3		
39495.30000	Node #3	S_M rcvd PERIODIC from Node #4	EVNT #	2
		Reset Timer element of Node #4		
39498.04000	Node #3	S_M rcvd PERIODIC from Node #2	EVNT #	2
		Reset Timer element of Node #2		
39503.76900	Node #3	S_M detects FAILURE on Node #1		
		Notify NF task.		
39523.89000	Node #3	R_L rcvd FN_OFF from Node #4	EVNT #	3
		No further action required ATT.		
39527.99000	Node #3	R_L rcvd FN_ON from Node #2	EVNT #	3
		Neither act/deact node and changing NST.		
		1 2 3 2 1 2 3 4 1 2 3 4		
39548.80000	Node #3	O_S sending STATUS msg.	EVNT #	3
39551.16000	Node #3	S_M rcvd PERIODIC from Node #3	EVNT #	3
		Reset Timer element of Node #3		
39553.90000	Node #3	S_M rcvd PERIODIC from Node #4	EVNT #	4
		Reset Timer element of Node #4		
39556.63000	Node #3	S_M rcvd PERIODIC from Node #2	EVNT #	4
		Reset Timer element of Node #2		
39559.36000	Node #3	S_M rcvd APERIODIC from Node #1	EVNT #	2
		Sending APERIODIC with NST unique sections.		
39560.31000	Node #3	O_S sending STATUS msg.	EVNT #	4
39562.10000	Node #3	S_M rcvd APERIODIC from Node #3	EVNT #	4
		APERIODIC response already sent, no action.		
39564.83000	Node #3	S_M rcvd APERIODIC from Node #4	EVNT #	5
		APERIODIC response already sent, no action.		
39567.56000	Node #3	S_M rcvd APERIODIC from Node #2	EVNT #	5
		APERIODIC response already sent, no action.		
39570.29000	Node #3	S_M rcvd PERIODIC from Node #1	EVNT #	3
		Reset Timer element of Node #1		
39590.70000	Node #3	C_P rcvd MKR from Node #1	EVNT #	4
		Local CHKPT already conducted. Store UNIQ.		
39591.03000	Node #3	O_S sending MKR msg.	EVNT #	5
39593.43000	Node #3	C_P rcvd MKR from Node #3	EVNT #	5
		Local CHKPT already conducted. Store UNIQ.		
39596.16000	Node #3	C_P rcvd MKR from Node #4	EVNT #	6
		Local CHKPT already conducted. Store UNIQ.		
39597.53000	Node #3	C_P rcvd MKR from Node #2	EVNT #	6
		Local CHKPT already conducted. Store UNIQ.		

39600.26000	Node #3	R_L rcvd FN_OFF from Node #1 No further action required ATT.	EVNT #	5
39602.99000	Node #3	C_P rcvd CHKPT from Node #1 Global CHKPT complete store NST	EVNT #	6
39605.73000	Node #3	R_L rcvd FN_ON from Node #2 Neither act/deact node and changing NST. 2 2 3 2 1 2 3 4 1 2 3 4	EVNT #	7
39610.22000	Node #3	O_S sending STATUS msg.	EVNT #	6
39611.19000	Node #3	S_M rcvd PERIODIC from Node #3 Reset Timer element of Node #3	EVNT #	6
39626.16000	Node #3	S_M rcvd PERIODIC from Node #1 Reset Timer element of Node #1	EVNT #	7
39628.89000	Node #3	S_M rcvd PERIODIC from Node #4 Reset Timer element of Node #4	EVNT #	7
39631.62000	Node #3	S_M rcvd PERIODIC from Node #2 Reset Timer element of Node #2	EVNT #	8
39429.78000	Node #4	O_S sending STATUS msg.	EVNT #	1
39432.66000	Node #4	S_M rcvd PERIODIC from Node #1 Reset Timer element of Node #1	EVNT #	1
39435.38000	Node #4	S_M rcvd PERIODIC from Node #2 Reset Timer element of Node #2	EVNT #	1
39438.12000	Node #4	S_M rcvd PERIODIC from Node #3 Reset Timer element of Node #3	EVNT #	1
39440.86000	Node #4	S_M rcvd PERIODIC from Node #4 Reset Timer element of Node #4	EVNT #	1
39491.22000	Node #4	O_S sending STATUS msg.	EVNT #	2
39492.56000	Node #4	S_M rcvd PERIODIC from Node #3 Reset Timer element of Node #3	EVNT #	2
39495.30000	Node #4	S_M rcvd PERIODIC from Node #4 Reset Timer element of Node #4	EVNT #	2
39498.04000	Node #4	S_M rcvd PERIODIC from Node #2 Reset Timer element of Node #2	EVNT #	2
39503.77000	Node #4	S_M detects FAILURE on Node #1 Notify NF task.		
39521.94000	Node #4	O_S sending FNOFF to Node #2	EVNT #	3
39523.90000	Node #4	R_L rcvd FN_OFF from Node #4 No further action required ATT.	EVNT #	3
39528.00000	Node #4	R_L rcvd FN_ON from Node #2 I am the deactivating node and changing NST 1 2 3 2 1 2 3 4 1 2 3 4	EVNT #	3
39548.80000	Node #4	O_S sending STATUS msg.	EVNT #	4
39551.17000	Node #4	S_M rcvd PERIODIC from Node #3 Reset Timer element of Node #3	EVNT #	3
39553.90900	Node #4	S_M rcvd PERIODIC from Node #4 Reset Timer element of Node #4	EVNT #	4
39556.63900	Node #4	S_M rcvd PERIODIC from Node #2 Reset Timer element of Node #2	EVNT #	4

39559.37000	Node #4	S_M rcvd APERIODIC from Node #1	EVNT #	2
		Sending APERIODIC with NST unique sections.		
39560.31000	Node #4	O_S sending STATUS msg.	EVNT #	5
39562.10900	Node #4	S_M rcvd APERIODIC from Node #3	EVNT #	4
		APERIODIC response already sent, no action.		
39564.84000	Node #4	S_M rcvd APERIODIC from Node #4	EVNT #	5
		APERIODIC response already sent, no action.		
39567.57000	Node #4	S_M rcvd APERIODIC from Node #2	EVNT #	5
		APERIODIC response already sent, no action.		
39570.29900	Node #4	S_M rcvd PERIODIC from Node #1	EVNT #	3
		Reset Timer element of Node #1		
39590.70000	Node #4	C_P rcvd MKR from Node #1	EVNT #	4
		Local CHKPT already conducted. Store UNIQ.		
39591.03000	Node #4	O_S sending MKR msg.	EVNT #	6
39593.43000	Node #4	C_P rcvd MKR from Node #3	EVNT #	5
		Local CHKPT already conducted. Store UNIQ.		
39596.16000	Node #4	C_P rcvd MKR from Node #4	EVNT #	6
		Local CHKPT already conducted. Store UNIQ.		
39597.53000	Node #4	C_P rcvd MKR from Node #2	EVNT #	6
		Local CHKPT already conducted. Store UNIQ.		
39600.26000	Node #4	R_L rcvd FN_OFF from Node #1	EVNT #	5
		No further action required ATT.		
39602.99900	Node #4	C_P rcvd CHKPT from Node #1	EVNT #	6
		Global CHKPT complete store NST		
39605.74000	Node #4	R_L rcvd FN_ON from Node #2	EVNT #	7
		Neither act/deact node and changing NST.		
		2 2 3 2 1 2 3 4 1 2 3 4		
39611.19900	Node #4	S_M rcvd PERIODIC from Node #3	EVNT #	6
		Reset Timer element of Node #3		
39625.58000	Node #4	O_S sending STATUS msg.	EVNT #	7
39626.17000	Node #4	S_M rcvd PERIODIC from Node #1	EVNT #	7
		Reset Timer element of Node #1		
39628.90000	Node #4	S_M rcvd PERIODIC from Node #4	EVNT #	7
		Reset Timer element of Node #4		
39631.62900	Node #4	S_M rcvd PERIODIC from Node #2	EVNT #	8
		Reset Timer element of Node #2		

REFERENCES

1. Shukla S., Yang C., Puett R., Lehman K., Masters M., "A Framework for Node Failure/Repair Transparency in Distributed Real-time Systems," paper submitted to the Fault Tolerant Computing International Symposium, Boston, MA, 1992.
2. Puett, R., *Reconfiguration in Robust Distributed Real-Time Systems Based on Global Checkpoints*, Master's Thesis, Naval Postgraduate School, Monterey, California, DEC 1991.
3. Shirazi B., Wang M. Pathak G., "Analysis and Evaluation of Heuristic Methods for Static Task Scheduling," *Journal of Parallel and Distributed Computing* 10, pp 222-232, 1990.
4. Deitel H.M., *Operating Systems*, pp. 500-550, Addison-Wesley Co., 1990.
5. Lo V.M., "Heuristic Algorithms for Task Assignment in Distributed Systems," *IEEE Transaction on Computers*, Vol 37, No. 11, pp 1384-1397, NOV 1988.
6. Chu W.W., Holloway L.J., Lan M-T, Efe K., "Task Allocation in Distributed Data Processing," *Computer*, Vol 13, pp 57-69, NOV 1980.
7. Houstis C.E., "Module Allocation of Real-Time Applications to Distributed Systems," *IEEE Transactions on Software Engineering*, Vol 16, No. 7, pp 699-708, JUL 1990.
8. Ma P-Y.R., Lee E.Y.S., Tsuchiya M., "A Task Allocation Model for Distributed Computing Systems," *IEEE*, Vol C-31, pp 41-47, No. 1, JAN 1982.
9. Chu W.W., Lan M.T., "Task Allocation and Precedence Relations for Distributed Real-Time Systems," *IEEE Transactions on Computers*, Vol 36, No. 6, pp 57-69, JUN 1987.
10. Williams E.A., "Assigning Processes to Processors in Distributed Systems," *Proceedings IEEE Conference on Parallel Processing*, pp 404-406, 1983.
11. Shin K.G., Chang Y.C., "Load Sharing in Distributed Real-Time Systems with State-Change Broadcasts," *IEEE Transactions on Computers*, Vol 38, No. 8, pp 1124-1143, AUG 1989.
12. Ramamritham K., Stankovic J.A., Zhao W., "Distributed Scheduling of Tasks with Deadlines and Resource Requirements," *IEEE Transactions on Computers*, Vol 38, No. 8, AUG 1989.

INITIAL DISTRIBUTION LIST

- | | | |
|----|---|---|
| 1. | Defense Technical Information Center | 2 |
| | Cameron Station | |
| | Alexandria, VA 22304-6145 | |
| 2. | Library, Code 52 | 2 |
| | Naval Postgraduate School | |
| | Monterey, CA 93943-5000 | |
| 3. | Chairman, Code EC | 1 |
| | Department of Electrical and Computer Engineering | |
| | Naval Postgraduate School | |
| | Monterey, CA 93943-5000 | |
| 4. | Professor Shridhar B. Shukla, Code EC/Sh | 1 |
| | Department of Electrical and Computer Engineering | |
| | Naval Postgraduate School | |
| | Monterey, CA 93943-5000 | |
| 5. | Professor Chyan Yang, Code EC/Ya | 1 |
| | Department of Electrical and Computer Engineering | |
| | Naval Postgraduate School | |
| | Monterey, CA 93943-5000 | |
| 6. | Commanding Officer | 1 |
| | Supervisor of Shipbuilding | |

Conversion and Repair, USN

Pascagoula, MS 39568-2210

7. Michael W. Masters, Code N35

1

Naval Surface Warfare Center

Dahlgren, VA 22448-5000