NAVAL POSTGRADUATE SCHOOL

Monterey, California

THESIS

RECONFIGURATION IN ROBUST DISTRIBUTED
REAL-TIME SYSTEMS
BASED ON GLOBAL CHECKPOINTS

by

Ronnie Douglas Puett

December 1991

Thesis Advisor: Shirdhar B. Shukla
Co-Advisor : Chyan Yang

Approved for public release; distribution is unlimited

ıclassified
urity Classification of this page

REPORT DOCUMENTATION PAGE

.. Report Security Classification: UNCLASSIFIED

1b. Restrictive Markings

.. Security Classification Authority

3. Distribution Availability of Report: Approved for public release; distribution is unlimited.

›. Declassification/Downgrading Schedule

Performing Organization Report Number(s)

5. Monitoring Organization Report Number(s)

ı. Name of Performing Organization: Naval Postgraduate School

6b. Office Symbol *(if applicable)*: EC

7a. Name of Monitoring Organization: Naval Postgraduate School

. Address *(City, State, and ZIP Code)*: Monterey, CA 93943-5000

7b. Address *(City, State, and ZIP Code)*: Monterey, CA 93943-5000

ı. Name of Funding/Sponsoring Organization

8b. Office Symbol *(if applicable)*

9. Procurement Instrument Identification Number

:. Address *(City, State, and ZIP Code)*

10. Source of Funding Numbers

Program Element Number	Project No.	Task No.	Work Unit Accession No.

. Title *(Include Security Classification)*
:configuration in Robust Distributed Real-Time Systems Based on Global Checkpoints

:. Personal Author(s)
ıett, Ronnie Douglas

ia. Type of Report	13b. Time Covered	14. Date of Report *(Year, Month, Day)*	15. Page Count
Master's Thesis	From ______ To ______.	December 1991	95

 i. Supplementary Notation: The views expressed in this thesis are those of the author and do not reflect the fficial policy or position of the Department of Defense or the United States Government.

'. Cosati Codes

Field	Group	Subgroup

18. Subject Terms *(Continue on reverse if necessary and identify by block number)*
Node Failure/Repair, Transparency, Distributed Real-Time, Migration Checkpointing

›. Abstract *(Continue on reverse if necessary and identify by block number)*

Fast, ultra-reliable, real-time computing is fundamental in today's weapons system. Increased system ıroughput and reliability can be achieved by utilizing distributed systems in which a single application ·ogram executes on multiple processors, connected to a network. The distributed nature of such systems ake it possible to tolerate failures and react to overloads without the application level performance degrading ıacceptably. Fault tolerance in these systems typically involves fault detection and recovery. Repair following ilure involves smooth integration of the repaired processor and subsequent reconfiguration. These actions ust take place transparently, that is without the application program noticing it. Therefore, sufficient formation must be maintained through the use of checkpointing to describe the state of the system at any me and ensure correct operation after failure/repair.

This thesis investigates a possible framework for achieving a fault- tolerant real-time distributed system hich provides transparent function-to-function message passing, status monitoring using periodic health

'. Distribution/Availability of Abstract: ☒ unclassified/unlimited ☐ same as report ☐ DTIC users

21. Abstract Security Classification: UNCLASSIFIED

a. Name of Responsible Individual: Shridhar B. Shukla

22b. Telephone *(Include Area Code)*: (408) 646-2764

22c. Office Symbol: EC/Sh

FORM 1473, 84 MAR — 83 APR edition may be used until exhausted. All other editions are obsolete

security classification of this page
Unclassified

19. ABSTRACT Continued:

messages and maintains a globally consistent system state by carrying out independent checkpointing proc dures. The proposed scheme is simulated using concurrent Ada processing for a four node, twelve functic distributed system.

Approved for public release; distribution is unlimited

Reconfiguration in Robust Distributed Real-Time Systems Based on Global Checkpoints

by

Ronnie D. Puett
Lieutenant, USN
B.S.C.S, University of Mississippi, 1985

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN ELECTRICAL ENGINEERING

from the

NAVAL POSTGRADUATE SCHOOL

December, 1991

ABSTRACT

Fast, ultra-reliable, real-time computing is fundamental in today's weapons system. Increased system throughput and reliability can be achieved by utilizing distributed systems in which a single application program executes on multiple processors, connected to a network. The distributed nature of such systems make it possible to tolerate failures and react to overloads without the application level performance degrading unacceptably. Fault tolerance in these systems typically involves fault detection and recovery. Repair following failure involves smooth integration of the repaired processor and subsequent reconfiguration. These actions must take place transparently, that is without the application program noticing it. Therefore, sufficient information must be maintained through the use of checkpointing to describe the state of the system at any time and ensure correct operation after failure/repair.

This thesis investigates a possible framework for achieving a fault- tolerant real-time distributed system which provides transparent function-to-function message passing, status monitoring using periodic health messages and maintains a globally consistent system state by carrying out independent checkpointing procedures. The proposed scheme is simulated using concurrent Ada processing for a four node, twelve function, distributed system.

TABLE OF CONTENTS

LIST OF FIGURES

I. INTRODUCTION

A. GENERAL

Distributed systems have become increasingly popular in satisfying the requirements for increased computing power and also as a means of achieving fault tolerance in critical real-time systems [Ref. 1]. Distributed systems are often defined to encompass a wide range of loosely coupled computer systems, especially network based systems. In loosely coupled distributed systems, there are no shared resources; therefore, all information exchanged between the relocatable functions must occur via message passing [Ref. 2]. As the processing speed of system nodes and the transmission capacity of message transfer media increase due to technological advances, message transmission time becomes small enough to provide a resource management that makes the distributed nature of the system transparent to the user. This resource management must maintain continuity of processing information for dynamically relocated functions and therefore, requires the system state information to be globally consistent [Ref. 3]. This state consists of the information necessary to describe the characteristics of all system nodes and functions. In order to maintain global consistency, some method of checkpoint and rollback procedures must be utilized. A checkpoint is a saved local state of a node's active functions[Ref. 4]. A set of checkpoints, one per node, is consistent if the saved states form a consistent global state. Rollback is defined as the retransmission of messages from the last checkpoint in order to restart the system after node failure.

Two approaches to node recovery and function reconfiguration are replicated execution and local checkpointing, coupled with rollback, to build a consistent global state. The problems of keeping replicas consistent in the former are formidable [Ref. 5]. Also, the number of node failures which can be tolerated must be known *a priori* in order to determine the requisite number of replications. In the absence of synchronization, functions cannot all recover simultaneously. Recovering functions asynchronously can introduce situations in which a single failure can cause an infinite number of rollbacks, preventing system progress. Local checkpointing may result in a rollback whose completion time can vary considerably; therefore, it is unsuitable to mission critical environments [Ref. 6].

The proposed framework for a distributed system utilizes the replication of code at each node and maintains a global snapshot of the system state. This framework minimizes recovery time, making it unnecessary to use rollback procedures during migration, except in cases of node failure.

B. AIM OF THE STUDY

The objective of this thesis is to implement the framework necessary to provide transparent function-to-function message passing, fault detection and checkpointing in a robust, real-time distributed system. Robustness is the system's ability to withstand failures and utilize reconfiguration to minimize the impact of these failures on overall system performance. Distribution requires the partitioning of an application program into multiple functions, the code for which is resident at every node. However, the responsibility for execution of a particular function is assigned to only one node in this framework. This function assignment may be fixed at initialization or may change as a result of reconfiguration. Communication between these dynamically relocatable functions is via a globally ordered network. This loosely coupled system does not share any resources, as illustrated in Figure 1.1, which is reproduced from another document [Ref. 7].

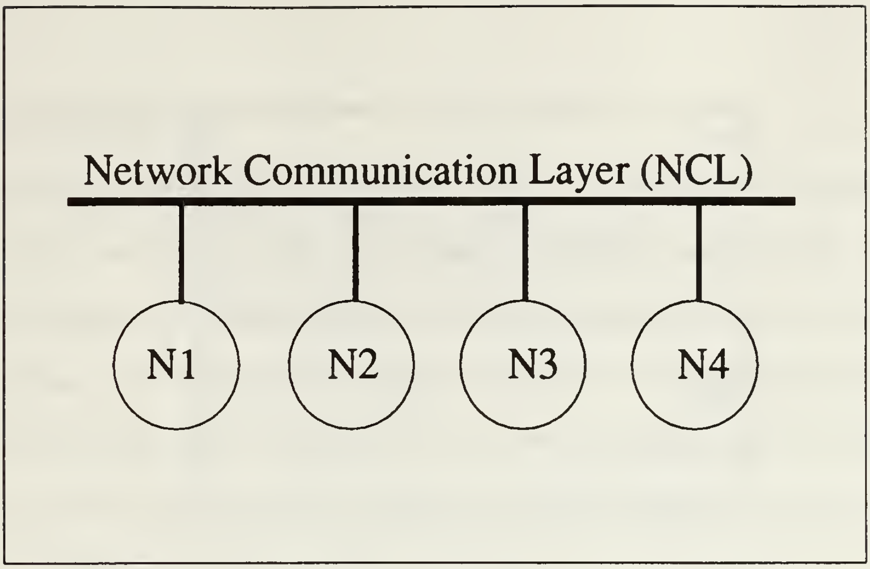

Figure 1.1: A Loosely Coupled Distributed System

The scope of this thesis is to implement the means necessary to provide fault tolerance and maintain the required information to allow a rapid system reconfiguration.

C. METHOD OF APPROACH

This thesis focuses on a single application executing on a distributed system. A layered architecture was chosen to organize the different components in an easy to manage, hierarchical fashion. The layers operate concurrently, yet interface to maintain communication between dynamically relocatable functions. This enables fault tolerance and load balancing efforts to proceed independently without interruption of the actual application processing.

Fault tolerance is accomplished by requiring each node in the system to periodically broadcast its load. Receipt of these status messages does not only indicate that the node is operational, but the load information is also utilized in the reconfiguration algorithms. These algorithms require globally consistent data upon which

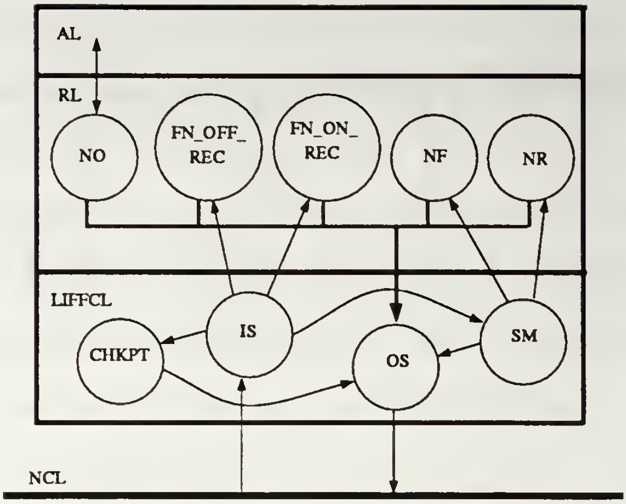

Figure 1.2: Software Layer Configuration at Each Node

to base their decisions. The globally consistent state information is maintained at each node through the use of independent checkpointing procedures. A system node containing four independent software layers and internal communication paths indicated by arcs, is depicted in Figure 1.2, which is reproduced from another document [Ref. 7]. The Network Communication Layer (NCL) must be a globally ordered communications protocol which enables the broadcast of all messages. The Location Invariant Function to Function Communication Layer (LIFFCL) provides each node with the necessary communications interface to the NCL, implements fault tolerance and checkpointing procedures. The LIFFCL is the major emphasis of this thesis and is covered extensively in Chapters III and IV. The Reconfiguration Layer (RL) handles function allocation/reconfiguration and is covered in detail in [Ref. 8]. The Applications Layer (AL) conducts actual application program execution and is responsible for the message queue management of all active functions at a node. Specification of AL functionality is to be covered in future thesis topics.

D. ORGANIZATION

This thesis is organized as follows. Chapter II discusses the issues in a distributed system and the mechanisms necessary to address these issues. Chapter III discusses the means of achieving function to function communications, fault tolerance, and maintaining state information. The detailed action of the tasks within the LIF-FCL of an individual node is illustrated in the state diagrams shown in Chapter IV. An overview of the implementation software and the simulation results are contained in Chapter V. Chapter VI contains the conclusion.

II. ISSUES IN MAINTAINING THE SYSTEM STATE

A. GENERAL

As indicated previously, the *state* of a distributed system entails all the variables necessary to describe any or all of the system components at any point in time. The distributed nature of such a system requires this state information to be current and accessible by all nodes. The integrity of this data must be maintained in order to implement fault tolerant procedures which enable continuity of a function's processing regardless of its location. To prevent the loss of state of the functions running on a node when the node fails, the system state must be periodically updated and distributed to all nodes utilizing checkpointing procedures, as stated in Chapter I. This globally consistent state information is required by reconfiguration algorithms in making relocation decisions. These algorithms are covered in another thesis [Ref. 8]. Issues requiring the use of a system's state information are described in the following sections.

B. ALLOCATION

Allocation is achieved at compile time or during execution. If conducted during execution, it requires knowledge of the current system state information obtained during checkpointing.

C. MAINTAINING STATE OF FUNCTIONS

As stated earlier, reconfiguration efforts require a globally consistent restart point. This restart point is determined by storing a function's unique variables at each node during checkpointing. In order to describe the state of a function, some of the attributes that must be known about a function are the last message received, the last message processed, time remaining till completion, time remaining till deadline, all symbol variables, and general register contents, etc. When a function gets processing time at a node, these statistics are updated and stored for that function. Keeping the state of every function at every node prevents retransmission of messages if the node where the function was active fails or cannot complete the function on time. Another node can activate the function and maintain continuity of processing rather than restarting the function at the last checkpoint. Each node maintains a unique section for the data relevant to its active functions. All nodes share this data by passing other nodes their unique section during checkpoint procedures as described in Chapter I. This allows for ease of transportability of functions and minimizes the communications required for this migration.

D. MAINTAINING STATUS OF NODES

Another factor in reconfiguring a system is the operational status of all nodes. This status is maintained through health monitoring schemes which depend totally on the exchange of status messages. Detection of node failure must result in the migration of the assigned functions to active nodes. Knowledge of each node's status prevents assigning a function to a non-active node.

In conjunction with the status of a node, its current load is also important. Knowledge of every node's loading percentage may prevent a node from becoming overloaded and resulting in functions not being completed on time. If a node is fully loaded, transferring a function to it only overloads the node. This causes a degradation not only to the individual node but the entire system since unnecessary communication is required by the now overloaded node in an effort to migrate a function to reduce loading. By keeping track of a node's status and load, appropriate decisions can be made when reconfiguration is necessary.

E. ROUTING

A function's location must be known at all times if a system is to support function to function communication through the use of *data* messages. Nodes must maintain a queue for each function in order to store all *data* messages destined for a particular function. The active function queues are maintained in the AL, and the non-active function queues are maintained in the LIFFCL. Requiring each node to maintain function queues, minimizes the amount of traffic to be transferred during migration of functions. This prevents rollback during reconfiguration, except in the case of node failure. Checkpointing and fault detection schemes provide the means to update the variables necessary to describe the global state of the system, as indicated above. These variables are maintained in a resource called the node status table (NST), constructed at each node, as shown in Figure 2.1, which is reproduced from another document [Ref. 7]. The NSTs are maintained consistent through the exchange of node *status* messages, as well as *marker* messages during checkpoint. The composition of the NST is detailed in the following section.

F. NODE STATUS TABLE

The NST is comprised of three sections: a section containing status information that is common to all nodes, a section containing all the information unique to the functions that are active on each node, and the node's identity. A given node contains two complete copies of the NST; the duplicate copy being designated node status backup (NSTBAK). Duplication of data guards against loss of information as a result of node failure during checkpointing. The NST contains variables which are used to describe the health of all nodes, the state of all functions, and the events since the

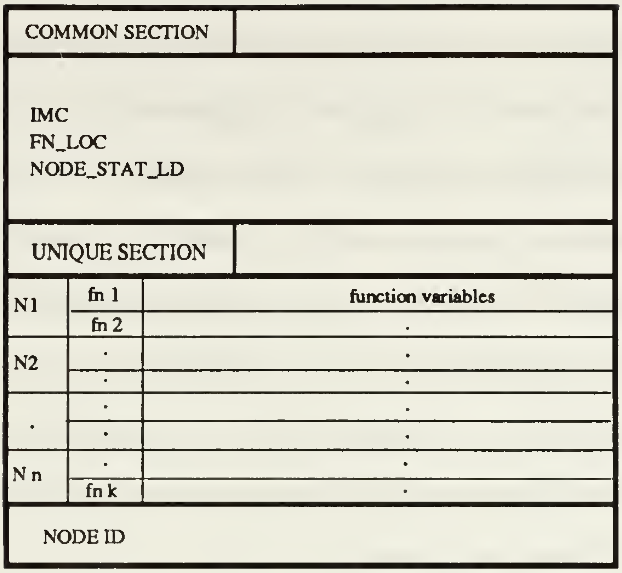

Figure 2.1: Node Status Table

last checkpoint.

1. Common Section

The node status indicates if a node is *up* or *down*. This information is updated through the use of status messages transmitted periodically by each node. If a *periodic status* message is not received from a node within a specified time interval, the node is assumed to have failed and is logged *down*.

2. Unique Section

The unique section contains the current state information for all functions within the system. It consists of a subsection for each system node, with the subsections containing separate records for those functions assigned to the appropriate node. The functions' state information is obtained during checkpointing by each node exchanging the applicable unique subsections of their NST.

Each node records and saves all messages sent between any two checkpoints.

All messages are contained in one of three places at a given node. The active queue in the AL contains messages for all functions assigned to the node and the non-active queue in the LIFFCL contains the messages for all remaining system functions. Also messages not yet transmitted or received by the node are in the `Output Server` or `Input Server` queues respectively. When a function is migrated, the receiving node utilizes the messages from the non-active queue within its LIFFCL to update the active queue for the activated function. Any messages in the output/input queues are not be affected by the migration process. However, if a node fails, its current unique section is not accessible to the new node and any messages in its output/input queues are lost; therefore, a rollback is necessary.

3. Node Identification

NODE_ID is self-explanatory. Several of the algorithms within the LIFFCL and RL use this variable to determine the identity of the node since all nodes are running concurrently. Specifics on the use of NODE_ID can be found in the program located in Appendix A.

4. Local Variables

In addition to the NST, each node maintains local variables used for node recovery, checkpointing, and queue management. These variables are explained in detail in the following sections.

a. Recovery Variables

The recovery variables are utilized by the recovering node to indicate when it is ready to commence normal processing. These variables are utilized to prevent unnecessary communication between the recovering and active nodes as explained below.

Recovery in Progress (RCVRY_IN_PROG) is the variable which indicates that a recovery is taking place. It prevents another *periodic* message from retriggering the recovery process. Retriggering the recovery process could put the nodes in an infinite loop. In this case, recovery of a node can never be completed. Recovery (RCVY) is used to indicate when a node has completed recovery. In order to recover, a node must rebuild its NST. This is accomplished by each of the other nodes sending the common and unique sections of their NST. Each element of RCVY indicates whether the corresponding node has sent its unique and common sections of the NST to the recovering node. Once completion of recovery is detected, the node clears the RCVY array and resets RCVRY_IN_PROG to false. Unique Sent (UNIQ_SENT) is utilized by the active nodes to indicate that a node has responded to a recovery operation by sending its NST sections. Once complete recovery is detected, the nodes reset this variable. UNIQ_SENT prevents additional messages from being generated.

b. Checkpoint Variables

The checkpoint variables are utilized when updating the global state of the system. Checkpoint Taken (CHKPT_TAKEN) is utilized to indicate when a *marker* message has been received from all active nodes. A *marker* message is sent by a node which has conducted a local checkpoint. CHKPT_TAKEN is used by the checkpoint originator to indicate when a checkpoint is complete. Event Count Out (EVNT_CNT_OUT) keeps track of the number of messages that are sent to the network. This is only used to track messages in the output files created by the simulation program.

c. Queue Management

Queue management variables are required to ensure the integrity of all messages at a given node. This is particularly important when dealing with circular queues. Messages can be written over easily if pointers are not maintained properly. For this reason, several variables are maintained for management of the queues. MSG_TO_SEND is used to indicate that there are messages in the queue to send. BLOCK_WRITE is used to prevent overwriting a message in the queue that has not been read. RD_CNT is used as a pointer to the next message to be read. MSG_CNT is used as a pointer to the next available queue slot into which a message can be written.

G. SUMMARY

The status of each node and the current statistics of each function must be maintained in the NST in order to describe the global state of the distributed system. Although maintaining the variables of the NST requires the overhead incurred with checkpointing procedures, the time spent is more than compensated for by quicker fault detection and faster and more efficient reconfiguration algorithms. The checkpointing and fault detection algorithms utilized to maintain the NST are covered in the following chapters.

III. THE LOCATION INVARIANT FUNCTION TO FUNCTION COMMUNICATION LAYER

A. GENERAL

This chapter examines the Location Invariant Function to Function Communication Layer (LIFFCL), its components, and their interface with the other layers of the node. The LIFFCL accomplishes three distinct objectives within the node. The first objective is to provide the node a communication interface with the NCL, in order to support communication between the system functions. Secondly, it performs fault detection by monitoring the health of all system nodes. It also generates *periodic* health (*status*) messages to inform other nodes of its own status. Lastly, the LIFFCL implements checkpoint procedures which are utilized to develop globally consistent system states.

The LIFFCL is comprised of four specific components: `Input Server (IS)`, `Output Server (OS)`, `Status Monitor (SM)`, and `Checkpoint (CP)`. The it provides communication interface with the NCL, via `Output Server` and `Input Server`. `Status Monitor` provides fault detection and `Checkpoint` monitors the occurrence of events at a given node and implements checkpointing. All of the components of this layer shown in Figure 1.2 are covered in detail in the following sections of this chapter. The logical progression of events for a particular task at a given node are illustrated in Chapter IV, utilizing state diagrams.

B. INPUT SERVER

The `Input Server` is responsible for receiving message traffic from the communication layer and redirecting messages to tasks within the node for the required action. It parses the message to determine its type and the destination task to complete the necessary action. It is a process that is activated periodically. It is during this activation time quantum that a node actually receives messages. Therefore, a queue is utilized, in which the NCL places messages. Queue management variables are utilized to indicate overflow and underflow conditions, as well as maintain message ordering within the queue. The `Input Server` consists of two tasks, `Node Initializer` and `Receive Msg`. It is initially given its node identification via a rendezvous call to task `Node Initializer`. Thereafter, `Input Server` is activated periodically by the expiration of a delay statement within the `Receive Msg` task. The duration of this delay is a parameter which can be changed in relation to the periodicity of the NCL delay, in order to analyze the affects on system throughput. The NCL delay determines the rate at which messages are sent to the `Input Server`. The `Input Server` maintains a circular queue which is written into by the NCL. The boolean variable BLOCK_WRITE is set to prevent the NCL from writing over a message that has not yet been read by the `Input Server`. When the NCL has a message to send, if BLOCK_WRITE is false, it places the message into the next available slot of the `Input Server` queue and sets MSG_TO_SEND to true. Upon detecting MSG_TO_SEND, the `Input Server` parses the MSG_KIND field to determine if the message is a *data* or *control* type. *Data* messages is sent to tasks within the AL, or to the function queue manager task of the LIFFCL. Control messages are sent to tasks within the RL or LIFFCL for the appropriate action. If the message is a *data* type and the function designated by the DEST_FUNC field is active on that particular node, the `Input Server` transfers the message to the AL. The AL must update the NST's unique section for the indicated function with the TOT of the last message received for that function and also the last *data* message processed for that function. If the *data* message is for a non-active function, `Input Server` sends it to a non-active function queue array. The details of the AL and the task to manage the non-active function queue are left for another thesis.

If the message is a *control* type, additional parsing of the CNTRL_ACTION field is required. IF the CNTRL_ACTION field is either a *fnon* or a *fnoff*, the `Input Server` transfers the message to RL for further processing. When the CNTRL_ACTION field is a *marker* (MKR) or a *checkpoint complete* (CHKPT) message, `Input Server` transfers the message to `Checkpoint`. If the CNTRL_ACTION field indicates a *status* (STATUS) message the `Input Server` transfers the message to the `Status Monitor`. The appropriate task receives the message by accepting a rendezvous call from `Input Server`. All of the necessary action required of the task is completed prior to the `Input Server` relinquishing processor control. In simulating a failed node, the Input Server only allows *status* messages to be passed to `Status Monitor`. The `Input Server` reads all other messages, but does not call the respective tasks. *Status* messages must be passed to `Status Monitor` since node recovery is triggered by the first *periodic status* message received after a node is restarted as explained later.

C. OUTPUT SERVER

The `Output Server` is responsible for ordering all message traffic generated by tasks within a node and relaying this traffic to the NCL. Ordering of a node's message traffic is accomplished utilizing queue management techniques as described in the previous section. Since all tasks within a node are concurrent processes, messages are placed into the `Output Server` message queue autonomously. For this reason, the queue management variables must be accessible to any task which generates message traffic. Proper maintenance of this queue ensures the chronological ordering of message generating events occurring internally to a node. When a tasks places a message into the `Output Server` queue for transmission, the task sets the boolean variable MSG_TO_SEND to true. Another boolean variable BLOCK_WRITE, is utilized to prevent tasks from overwriting a message in the `Output Server` queue before it can be passed to the NCL. During each periodic activation, if MSG_TO_SEND is true, the next available message in the `Output Server` queue is read from the queue and written into the NCL queue. Prior to placing a message into the NCL queue, `Output Server` appends a logical time stamp on the message for chronological identification purposes. The `Output Server` can only send message traffic if a BLOCK_WRITE condition does not exist within the NCL. The `Output Server` at any given node only relays at most one message during a given activation period. This prevents a given node's `Output Server` from monopolizing the network.

D. STATUS MONITOR

The overall purpose of the `Status Monitor` is to provide fault tolerant facilities for the node, by maintaining the current operational status of all system nodes in its NST. This is accomplished through the three functions that `Status Monitor` performs. The three separate functions are: generate *periodic status* messages indicating the health of the node, monitor and maintain a timer array within the NST to detect failure of other nodes, and processes all *status* messages received by the node. The health of the node is determined by the AL, and is a reflection of the node's ability to complete assigned functions prior to their deadline. A load percentage greater than one indicates an overloaded node. Fault detection is achieved by monitoring the receipt of these *periodic status* message from other system nodes. If a *periodic status* message is not received within a specified interval, node failure is assumed and the appropriate node is reflected as *down* in the NST. *Aperiodic* messages are utilized by the `Status Monitor` only during recovery procedures. `Status Monitor`, accessible from the Input Server, consists of two independent tasks, Status Broadcast (SB) and the Status Received (SR). The Status Broadcast is activated on a periodic basis, utilizing a simple delay statement. The activation of the Status Received is via a rendezvous call from the Input Server upon receipt of a *status* message. The primary means of determining node status, is for each node to periodically broadcast its load percentage to all other nodes. In turn, each node waits for these broadcasts as confirmation that other nodes are in fact operational. The Status Monitor at each node maintains a 1 by N array , each element containing the Time-of-Receipt (TOR) of the last *status* message received from the appropriate node. This value is used in comparisons with the Real-Time-Clock (RTC), to determine if nodes have failed to transmit periodical *status* messages. If a given node's Status Monitor detects the failure of another node, then it logs the failed node as *down* in the NST, and notifys the Node Failure routine.

1. Status Message Receipt

As previously indicated, two types of *status* messages are utilized, *periodic* and *aperiodic*, both of which are *control* type messages with the CONTROL_ACTION field set equal to *status*. All *status* messages received by the Input Server are passed to the Status Monitor for appropriate action.

Periodic messages are used to promulgate the fact that a node is operational, as well as to indicate its current load percentage. These messages are indicated by the presence of a "1" in the DEST_NODE field of the message, with the load percentage contained in the DEST_FUNC field. This loading information is utilized by the RL at each node in determining the receiving node in overload and recovery conditions. Recovery and overload conditions, are covered in another thesis.

The *aperiodic* messages are indicated by the presence of a "2" in the DEST_NODE field of the message. *Aperiodic* messages are transmitted in conjunction with a node recovery only. Upon restart, the recovering node transmits an *aperiodic* message with the load equal to zero, receipt of which causes all active nodes to transmit an *aperiodic* message containing the common and unique sections of their NST.

2. Status Message Broadcast

The `Status Broadcast` periodically generates local *status* broadcast messages, and checks the timeout conditions of other nodes. On each activation, `Status Broadcast` obtains the current value of the RTC and compares that to the TOR of the last *status* message received from the applicable node. If this time differential is greater than a predetermined Timeout interval, the associated node is reflected as *down* in the NST and the `Node Failure` task is called.

E. CHECKPOINTING PROCEDURES

Checkpointing procedures are the cornerstone of a distributed system's framework. As stated earlier, the main purpose of conducting checkpoints is to establish globally consistent points which serve as synchronization points during reconfiguration procedures. A local state of a node is defined by its initial state and the sequence of events that have occurred at that node since the previous checkpoint. An event occurs for each receive occurrence of a message. A checkpoint is merely a snapshot of a local state of a node at any point in time. A set of checkpoints, one for each node in the system, is called a global checkpoint and is consistent if all snapshots form a consistent global state[Ref. 6].

`Checkpoint` contains two independently activated task bodies, `Check Pt` and `Event Cnt`. Task `Check Pt` is activated by a rendezvous call from the `Input Server` upon receipt of a *marker* or *checkpoint complete* message. `Event Cnt`, activated periodically by the use of a delay statement, monitors the number of messages received by a given node and generates a *marker* message after receiving a pre-determined number of messages.

Checkpointing is conducted independently at each node. Checkpointing procedures are initiated by the first node to accumulate the pre-determined number of events. This node broadcasts a *marker* message containing its unique section of the NST. Upon receipt of this *marker* message other nodes conduct checkpoint locally if not already accomplished and update their NST with the unique section contained in the body of the *marker* message. Additionally, when the first *marker* message is received at a given node, the node also transmits a *marker* message containing its own unique section of the NST. Requiring each node in turn to transmit a *marker* message ensures that all nodes have exact replicas of the unique sections of the NST. When the node originating the checkpoint has received a *marker* message from all other active nodes, it transmits a *checkpoint complete* message. The communication protocol, a first-in-first-out network, ensures delivery of the *checkpoint complete* message (CHKPT) to each node occurs after all associated *marker* messages have been received. This ensures complete and identical NSTs at each node. Since there is no global synchronization of checkpointing events, the possibility exists that a node is required to alter its NST between the time of local checkpoint and receipt of *marker* messages from all other nodes. This is accomplished through the use of a temporary copy of a node's unique section, made at checkpoint time. The *marker* messages are retained in the temporary variable until a *checkpoint complete* message is received, at which time the temporary variable is written into the NST and the entire NST is duplicated in the backup copy NSTBAK. This method of retaining a backup copy of the NST, ensures that a globally consistent copy of the previous checkpoint is still available in the event that a node failure occurs during checkpoint procedures.

IV. STATE DIAGRAM REPRESENTATION OF TASKS

A. GENERAL

As previously mentioned, all tasks within the LIFFCL are concurrent processes. `Input Server` and `Output Server` are periodic tasks which are activated through the use of a time delay. A delayed task is suspended by the node's operating system during the period of the delay. Tasks `Status Monitor` and `Checkpoint` are activated by a rendezvous call from the `Input Server` upon receipt of certain message types. This chapter illustrates the logical progression of events occurring within the indicated task as shown in the state diagram. The actual implementation of the user program is covered in the next chapter.

B. INPUT SERVER TASK

`Input Server` periodically checks its queue for a message received. If a message is to be processed, it parses at most two fields to determine the message type as shown in Figure 4.1. Depending on its type, the message is passed to the appropriate layer for further processing in order to complete the necessary action required by the message. If no message is present, `Input Server` releases the processor.

C. OUTPUT SERVER TASK

`Output Server` checks flags set by tasks within the different layers of the node to determine if a message is available for transmission. The `Output Server` accomplishes this by transferring the message from its own queue to the queue of NCL. `Output Server` ensures the NCL queue is not full before writing the message in this queue. A

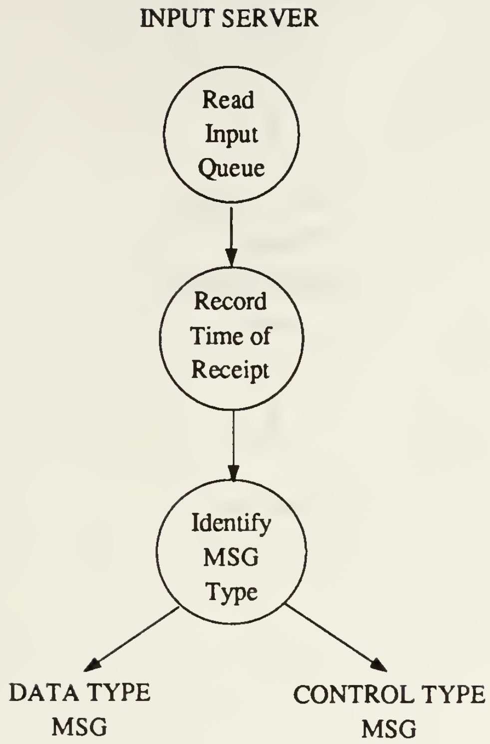

Figure 4.1: Input Server State Diagram

full queue is indicated by the NCL variable BLOCK_WRITE being true. It also time stamps the message to ensure its ordering. These events are illustrated in Figure 4.2.

D. STATUS MONITOR TASK

As indicated previously, `Status Monitor` performs three different functions. Two of these functions, `Status Broadcast` and `Timeout`, generate *periodic status* messages for the node. and monitor the receipt of these messages from other nodes. Additionally, `Status Received` is invoked by the `Input Server` upon receipt of both

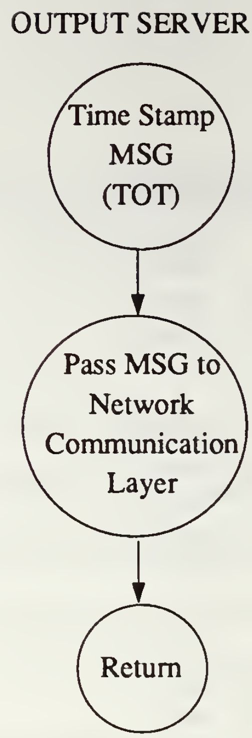

Figure 4.2: Output Server State Diagram

periodic and aperiodic status messages. The three functions and their resulting events are shown in Figures 4.3 and 4.4.

E. CHECKPOINT TASK

`Checkpoint` processes two types of messages pertaining to checkpointing. A *marker* message initiates checkpointing if not already in progress, and a *checkpoint complete* message signifies the successful completion of a checkpoint. Information pertaining to a node's functions is sent in the *marker* message so all nodes can update their NST's. Upon completion of checkpointing, a backup copy of NST is made. This backup copy is utilized during node failure, since the failed node is unable to pass the statistics of its active functions. Two procedures are utilized to process the different message types as shown in Figure 4.5.

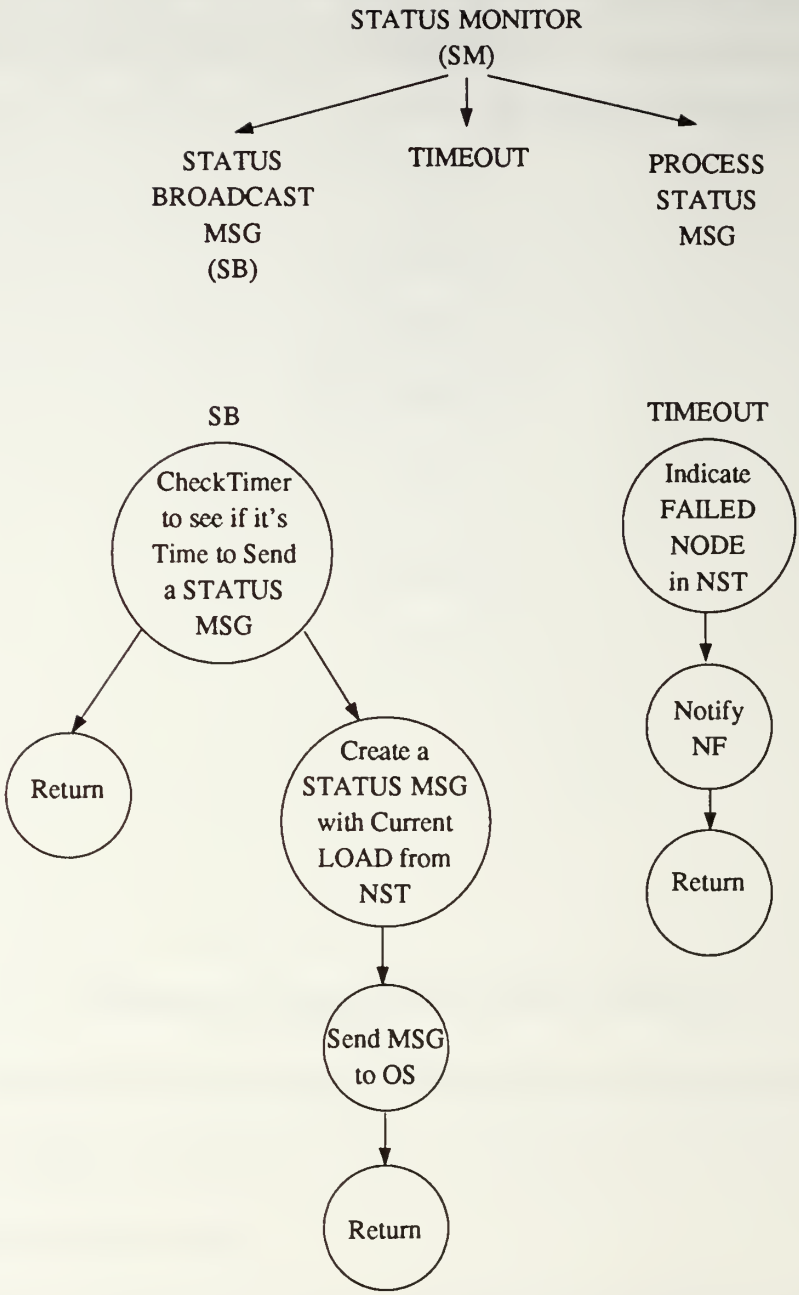

Figure 4.3: Status Monitor Broadcast and Timeout State Diagrams

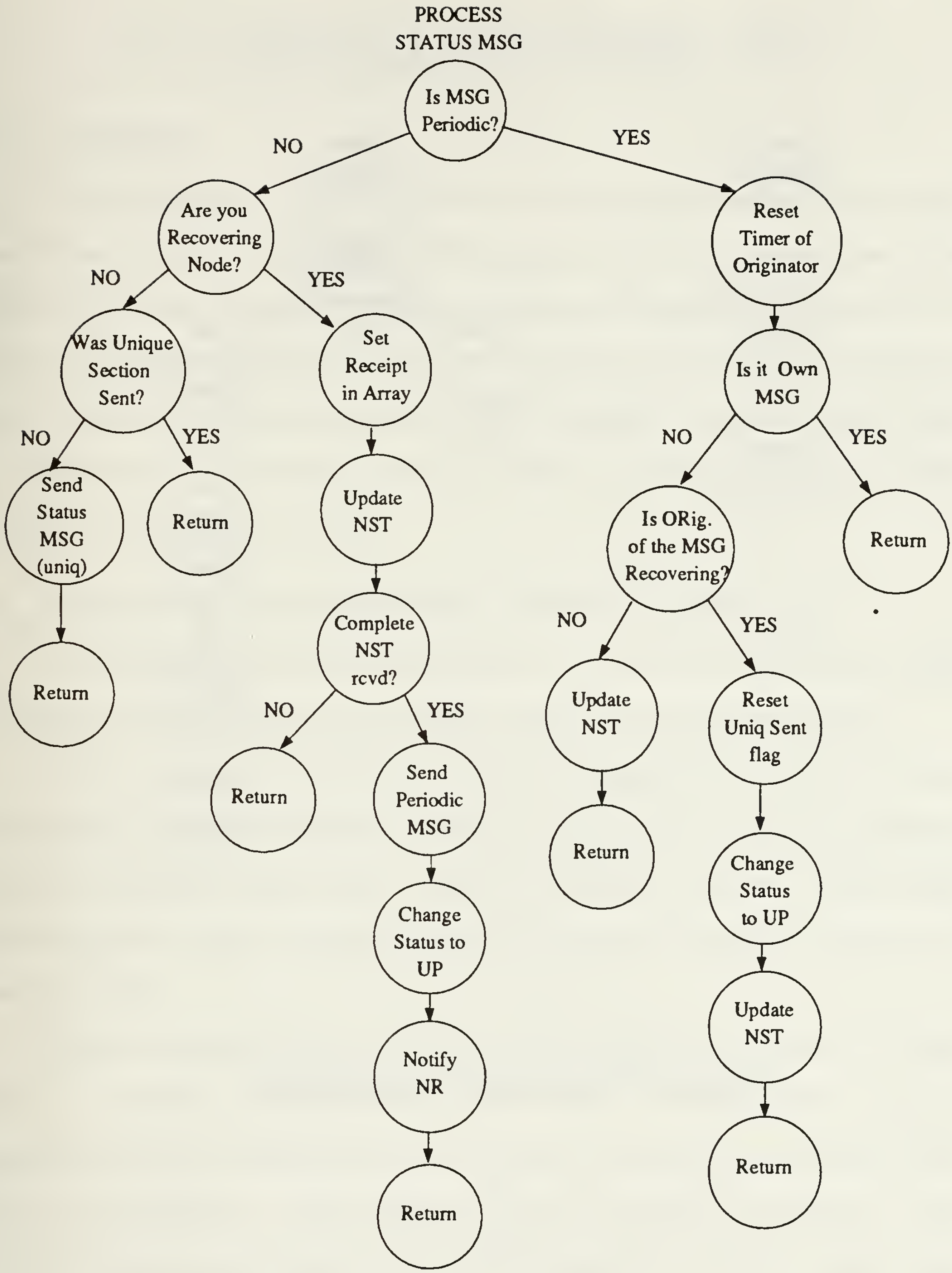

Figure 4.4: Status Monitor Message Received State Diagram

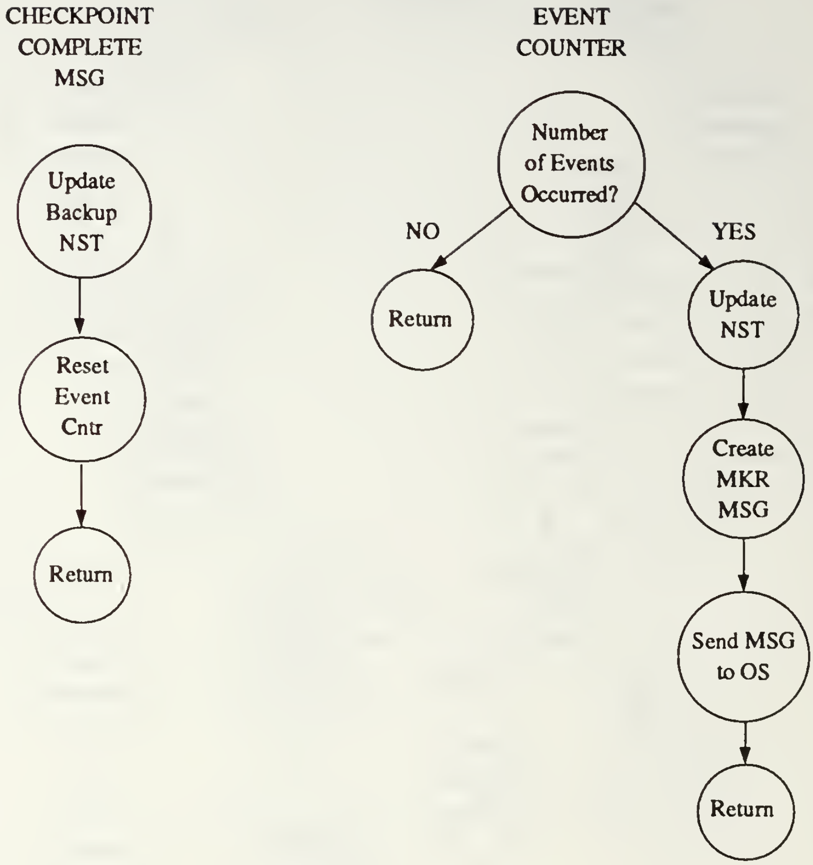

Figure 4.5: Checkpoint State Diagram

V. A SIMULATION USING ADA

A. GENERAL

The simulation of a four node, twelve function, distributed system is implemented as a group of independent Ada packages. Each node is comprised of the `Output` and `Input Servers`, the `Status Monitor`, `Checkpoint`, and the RL. All these components are instantiated for each node and are referred to as the node related components. The system also contains community components which include a globally ordered communication network (NCL), a random event generator (EG), and a front end processor (FEP).

B. SYSTEM-WIDE COMMUNITY COMPONENTS

The community components explained in this section, are the system components not utilized in the actual processing of *data* or *control* type messages.

NCL is used to simulate the transmission of messages from the nodes' `Output Servers` via a broadcast network. The `Input Servers` receive these messages from the NCL utilizing a circular queue. The delay difference between the NCL, `Output Server`, and the `Input Server` determines the number of messages in the queue at any given time.

The random event generator is activated periodically to simulate a real-time event. It simulates node overload and node failure. This simulation verifies the sequence of events occurring within the LIFFCL as a result of node failure/repair and overload conditions. The reconfiguration events normally occurring as a result of this simulation occur primarily in the RL layer and are covered in another thesis [Ref. 8].

C. NODE RELATED COMPONENTS

The node related components are algorithms and tasks utilized for processing the different types of messages received by a node. These components are used to implement each node and are are explained in this section.

The `Input Server` contains two independent task bodies, `Build Node` and `Receive Message`. The `Build Node` task is utilized by the `Front End Processor` only during the initialization of nodes as described previously. The other task receives messages from the NCL via a circular queue. The messages received are parsed to determine the necessary action to be taken. `Input Server` establishes a rendezvous with either the `Checkpoint`, `Status Monitor`, or the RL based on the contents of the MSG_KIND field of a message.

The `Output Server` consists of a single task activated periodically by the expiration of a delay statement. It sends any available messages to NCL during its activation period.

`Checkpoint` handles the process of checkpointing and ensures that a consistent global state is maintained. Any node can originate the checkpoint process by conducting a local checkpoint and sending a *marker* message containing its unique data. The node originating the checkpoint must keep track of *marker* messages received from other nodes and indicate when the checkpoint is complete. Upon receipt of the *marker* messages, all the nodes must store the information passed. This process is continued until a *checkpoint complete* message, sent by the originator is received by all nodes.

As indicated in Chapter III, the `Status Monitor` consists of three independent tasks, `Status Broadcast`, `Timeout`, and `Status Received`. `Status Broadcast` and `Timeout` are activated periodically by the expiration of a delay statement, and `Status Received` establishes a rendezvous with the `Input Server`. `Status Broadcast` is responsible for building and sending the *periodic* message to the `Output Server`. `Timeout` detects the failure of a node to respond with a *periodic* message within a specified time interval. `Status Received` processes both *periodic* and *aperiodic* messages. For *periodic* messages, a node only updates the NST. *Aperiodic* messages signal a node recovery; therefore, a node must respond by sending the unique and common section of its NST.

D. VERIFICATION OF STATE DIAGRAMS

To illustrate the correctness of the state diagrams shown in the previous chapters, timing diagrams are provided. They reflect the sequence of events occurring at a node during simulation following the receipt of messages built and sent by either the Event Generator or the implemented tasks of the LIFFCL.

Maintaining the global state of the system is accomplished by utilizing checkpointing procedures. `Checkpoint` is initiated by the first node to record a predetermined number of events. This node is designated as the checkpoint *originator*. As shown in Figure 5.1, *node 1* originates the checkpoint. The arcs represent the message transmission time between nodes. *Nodes 2, 3 and 4* respond to the *marker* message by conducting a local checkpoint and transmitting a *marker* message. Also it is worth noting that only one node is active at any given time. When *node 1* has received a *marker* message from all nodes, it sends a *checkpoint complete* message signifying a globally consistent checkpoint has been attained. Upon receipt of this *checkpoint complete* message, each node stores the checkpoint data into NSTBAK.

In order for the health of the nodes to be monitored, *periodic* status messages are sent by each node. Each node records the load of the node which sent the *periodic* message. A timer is used to determine if a node responded on time with this message. A diagram listing the periodic events that occur at each node in response to the receipt

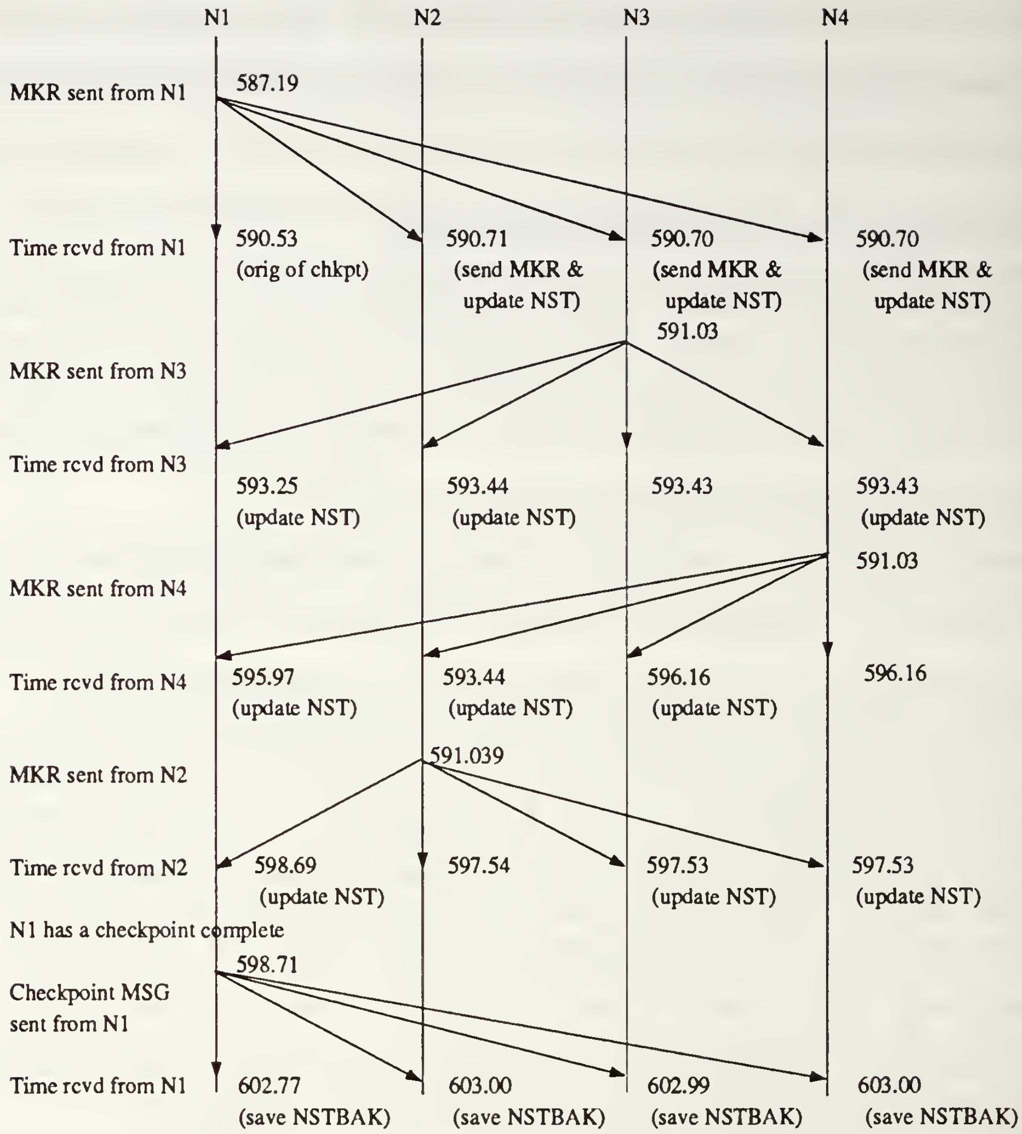

Figure 5.1: Checkpointing Events

of these periodic messages is illustrated in Figure 5.2.

E. SUMMARY

The actual code implemented in this simulation model is contained in Appendix A. The simulation output is contained in Appendix B. Comments have been inserted in the areas where an algorithm or procedure needs to be placed. Areas requiring further development are covered in the next chapter.

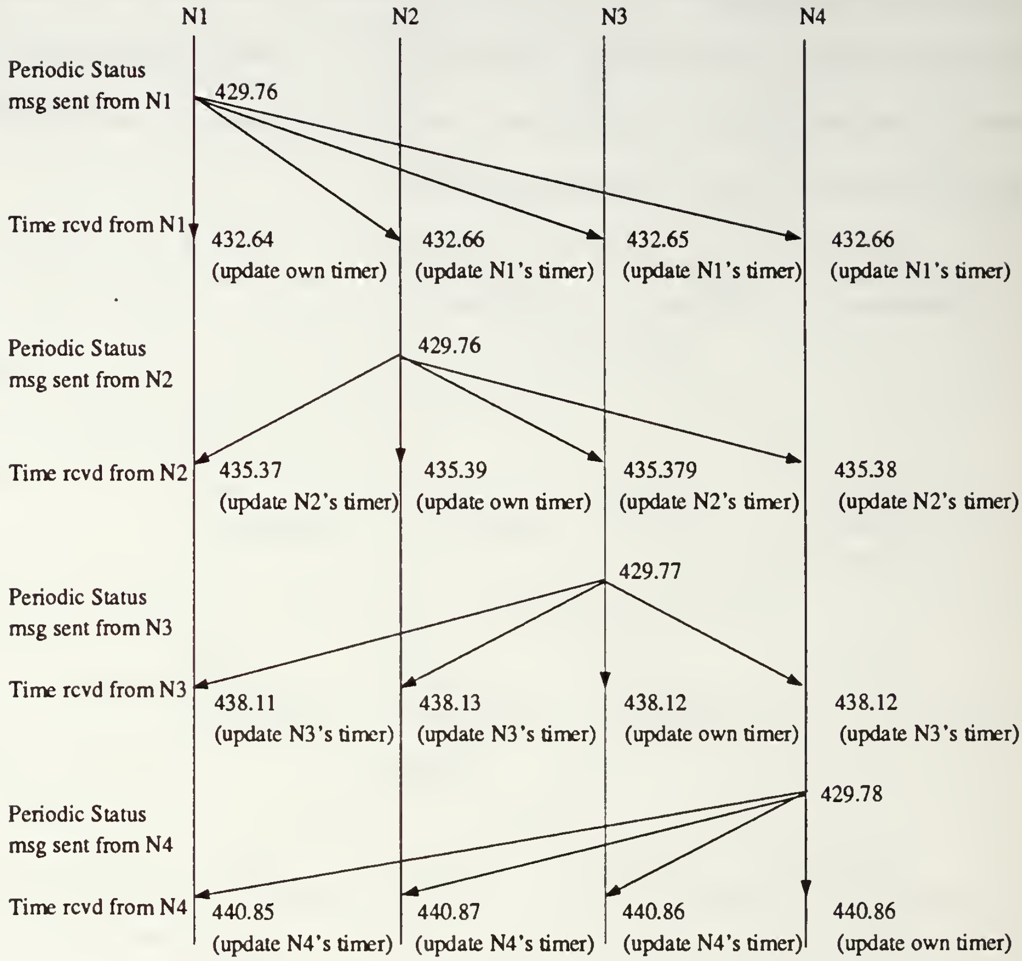

Figure 5.2: Periodic Message Processing

VI. CONCLUSIONS AND FUTURE WORK

A. GENERAL

In this thesis, a scheme for building robust, fault tolerant, distributed systems is presented. The proposed fault detection methodology, combined with the independent checkpointing and recovery techniques, is an effective means of obtaining fault tolerance. The checkpointing procedures enable a globally consistent system state to be stored at every node, allowing for robust reconfiguration efforts as a result of transient failures. Additionally, the duplication of all application code at each node reduces the communications normally associated with rollback/recovery and function migration. Also, requiring nodes to store all data messages received prevents retransmission of requisite message traffic during function migration.

B. CONCLUSION

The fault tolerance implementation described is a simple yet effective means for detecting node failure. However, in some critical real-time systems, the lag time between failure and its detection may need to be reduced. A reduction can be obtained by simply increasing the frequency with which the timeout array contents are examined. The trade-off is a reduction in the time slice that a node can dedicate to application processing.

The proposed asynchronous checkpointing scheme appears to provide better throughput and response time by eliminating the synchronization overhead normally required in creating globally consistent checkpoints. The domino effect, normally associated with asynchronous checkpoint is alleviated by maintaining a backup copy of the previous globally consistent checkpoint data. Should node failure occur during the process of checkpointing, the recovered functions must only rollback to the previous checkpoint.

The availability of large quantities of RAM storage makes the storage of all messages received an alternative. Rollback/recovery time increases dramatically if nodes are required to retransmit all requisite traffic for a recovering node. The linear processing time required for message queue manipulation during checkpointing is negligible compared to the overhead required for retransmission. Furthermore, achievement of a globally consistent state upon recovery requires all messages to be logged at either the transmitting or receiving node. It is believed to be advantageous to maintain the queue as a receive queue.

C. FUTURE WORK

In order to fully realize the capabilities of the proposed scheme, a more intensive analysis on a multi-processor implementation is required. A complete multi-layered system as depicted in Figure 1.2 must be utilized to analyze the periodicity relationship between the NCL, LIFFCL, RL and AL. A multi-processor environment would also yield a more realistic indication of the relationship between the frequency of checkpointing and failure recovery time. To enable truly independent functionality among the software layers of the node, circular queues should be implemented in each task. This prevents the `Input Server` from tying up the processor until a task completes the action required by a message. Also the development of the `Timeout` routine as a separate task would reduce the frequency with which `Status Broadcast` is currently being activated but still maintain a short detection time.

Additionally, queue management for *data* messages must be implemented in order to support the future development of the AL software. The AL software must also provide an interface to the RL and LIFFCL layers.

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
type FUNCTION_REC is                        -- contents of the unique section
   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;
```

```
    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
```

```
          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;
                 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;
    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);
```

```
         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;
```

```
                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;
            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;
              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;
```

```
                 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;
```

```
        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 if msg present
         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;
      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);
```

```
            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;
      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;
         ON := 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 timming */
/* 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     EVNT #   3
                       Reset Timer element of Node #1
39587.19000   Node #1  O_S sending MKR msg.               EVNT #   4
39590.53000   Node #1  C_P rcvd MKR from Node #1          EVNT #   4
                       I originated CHKPT. Not all MKRs yet rcvd.
39593.25000   Node #1  C_P rcvd MKR from Node #3          EVNT #   5
                       I originated CHKPT. Not all MKRs yet rcvd.
39594.87000   Node #1  O_S sending FNOFF to Node #2       EVNT #   5
39595.97000   Node #1  C_P rcvd MKR from Node #4          EVNT #   6
                       I originated CHKPT. Not all MKRs yet rcvd.
39598.69000   Node #1  C_P rcvd MKR from Node #2          EVNT #   6
                       MKRs rcvd from all nodes. Send CHKPT_COMP
39598.71000   Node #1  O_S sending CHKPT msg.             EVNT #   6
39600.05000   Node #1  R_L rcvd FN_OFF from Node #1       EVNT #   5
                       No further action required ATT.
39602.77000   Node #1  C_P rcvd CHKPT from Node #1        EVNT #   6
                       CHKPT orig. Global CHKPT complete store NST
39605.49000   Node #1  R_L rcvd FN_ON from Node #2        EVNT #   7
                       I am the deactivating node and changing NST
                       2  2  3  2  1  2  3  4  1  2  3  4
39610.93000   Node #1  S_M rcvd PERIODIC from Node #3     EVNT #   6
                       Reset Timer element of Node #3
39625.58000   Node #1  O_S sending STATUS msg.            EVNT #   7
39625.89000   Node #1  S_M rcvd PERIODIC from Node #1     EVNT #   7
                       Reset Timer element of Node #1
39628.61000   Node #1  S_M rcvd PERIODIC from Node #4     EVNT #   7
                       Reset Timer element of Node #4
39631.33000   Node #1  S_M rcvd PERIODIC from Node #2     EVNT #   8
                       Reset Timer element of Node #2


39429.76000   Node #2  O_S sending STATUS msg.            EVNT #   1
39432.66000   Node #2  S_M rcvd PERIODIC from Node #1     EVNT #   1
                       Reset Timer element of Node #1
39435.39000   Node #2  S_M rcvd PERIODIC from Node #2     EVNT #   1
                       Reset Timer element of Node #2
39438.13000   Node #2  S_M rcvd PERIODIC from Node #3     EVNT #   1
                       Reset Timer element of Node #3
39440.87000   Node #2  S_M rcvd PERIODIC from Node #4     EVNT #   1
                       Reset Timer element of Node #4
39491.22000   Node #2  O_S sending STATUS msg.            EVNT #   2
39492.57000   Node #2  S_M rcvd PERIODIC from Node #3     EVNT #   2
                       Reset Timer element of Node #3
39495.31000   Node #2  S_M rcvd PERIODIC from Node #4     EVNT #   2
                       Reset Timer element of Node #4
39498.05000   Node #2  S_M rcvd PERIODIC from Node #2     EVNT #   2
                       Reset Timer element of Node #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        EVNT #   5
                       No further action required ATT.
39602.99000   Node #3  C_P rcvd CHKPT from Node #1         EVNT #   6
                       Global CHKPT complete store NST
39605.73000   Node #3  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
39610.22000   Node #3  O_S sending STATUS msg.             EVNT #   6
39611.19000   Node #3  S_M rcvd PERIODIC from Node #3      EVNT #   6
                       Reset Timer element of Node #3
39626.16000   Node #3  S_M rcvd PERIODIC from Node #1      EVNT #   7
                       Reset Timer element of Node #1
39628.89000   Node #3  S_M rcvd PERIODIC from Node #4      EVNT #   7
                       Reset Timer element of Node #4
39631.62000   Node #3  S_M rcvd PERIODIC from Node #2      EVNT #   8
                       Reset Timer element of Node #2


39429.78000   Node #4  O_S sending STATUS msg.             EVNT #   1
39432.66000   Node #4  S_M rcvd PERIODIC from Node #1      EVNT #   1
                       Reset Timer element of Node #1
39435.38000   Node #4  S_M rcvd PERIODIC from Node #2      EVNT #   1
                       Reset Timer element of Node #2
39438.12000   Node #4  S_M rcvd PERIODIC from Node #3      EVNT #   1
                       Reset Timer element of Node #3
39440.86000   Node #4  S_M rcvd PERIODIC from Node #4      EVNT #   1
                       Reset Timer element of Node #4
39491.22000   Node #4  O_S sending STATUS msg.             EVNT #   2
39492.56000   Node #4  S_M rcvd PERIODIC from Node #3      EVNT #   2
                       Reset Timer element of Node #3
39495.30000   Node #4  S_M rcvd PERIODIC from Node #4      EVNT #   2
                       Reset Timer element of Node #4
39498.04000   Node #4  S_M rcvd PERIODIC from Node #2      EVNT #   2
                       Reset Timer element of Node #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        EVNT #   3
                       No further action required ATT.
39528.00000   Node #4  R_L rcvd FN_ON from Node #2         EVNT #   3
                       I am the deactivating node and changing NST
                       1  2  3  2  1  2  3  4  1  2  3  4
39548.80000   Node #4  O_S sending STATUS msg.             EVNT #   4
39551.17000   Node #4  S_M rcvd PERIODIC from Node #3      EVNT #   3
                       Reset Timer element of Node #3
39553.90900   Node #4  S_M rcvd PERIODIC from Node #4      EVNT #   4
                       Reset Timer element of Node #4
39556.63900   Node #4  S_M rcvd PERIODIC from Node #2      EVNT #   4
                       Reset Timer element of Node #2
```

```
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. Kleinrock L., "Distributed Systems," *Communications of the ACM*, Vol 28, No. 11, NOV 1985.

2. Deitel H.M., *Operating Systems*, pp. 500-550, Addison-Wesley Co., 1990.

3. Mullender S. and others, *Distributed Systems*, pp 319-357, Addison-Wesley Co., 1990.

4. Koo R., Toueg S., "Checkpointing and Rollback-Recovery for Distributed Systems," *IEEE Transactions on Software Engineering*, Vol. SE-13, No. 1, JAN 1987.

5. Lala J.H., Harper R.E., Alger L.S., "A Design Approach for Ultrareliable Real-Time Systems," *Computer*, Vol 24, No. 5, MAY 1991.

6. Bhargava B., Lian S., "Independent Checkpointing and Concurrent Rollback for Recovery in Distributed Systems - An Optimistic Approach," *Proc. of 7th Symp. on Reliable Distributed Systems, 1988.*

7. 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.

8. Lehman K., *Function Allocation in a Robust Distributed Real-Time Environment*, Master's Thesis, Naval Postgraduate School, Monterey, California, DEC 1991.

INITIAL DISTRIBUTION LIST

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

Conversion and Repair, USN

Pascagoula, MS 39568–2210

7. Michael W. Masters, Code N35 1

Naval Surface Warfare Center

Dahlgren, VA 22448–5000

Thesis
P947485 Puett
c.1 Reconfiguration in robust distributed real-time systems based on global checkpoints.