Chapter 19

THE RAPIDS SIMULATOR: A TESTBED FOR EVALUATING SCHEDULING, ALLOCATION, AND FAULT-RECOVERY ALGORITHMS IN DISTRIBUTED REAL-TIME SYSTEMS

M. Allalouf, J. Chang, G. Durairaj, J. Haines, V. R. Lakamraju, K. Toutireddy, O.S. Unsal, K. Yu, I. Koren and C. M. Krishna
Electrical and Computer Engineering Department,
University of Massachusetts,
Amherst, MA

Abstract Embedded parallel and distributed computing systems for real-time applications are becoming commonplace. Many such real-time applications are life-critical and require extensive fault-tolerance capabilities in order to ensure very high reliability. At the same time, cost, power, weight, and volume constraints require that any introduced redundancy must be efficiently used. Thus, failure-recovery strategies must be implemented to allow the system to most efficiently manage its resources in the presence of one or more failures, while attempting to continue the execution of the current tasks so as to not miss any deadlines. We have developed such a resource-management algorithm, which selects the optimal failure recovery procedure to be employed when a processor failure is detected. Further, in order to test this and other possible recovery algorithms, we have developed the RAPIDS simulator testbed. This paper describes the testbed, and shows how it provides a framework in which to simulate distributed real-time systems. In addition we describe how RAPIDS allows the user to validate fault recovery algorithms.

Keywords: Real-time systems, time-critical recovery, event-driven simulation, distributed systems, PVM

1. INTRODUCTION

One of the side-effects of the reduction in microprocessor cost has been the proliferation of computers in embedded applications, ranging all the way from washing machines to aerospace applications. Many of these applications are life-critical, and must offer very high reliability. The traditional approach

to ensuring adequate fault-tolerance has been to throw massive redundancy at the problem, in the hope that with a sufficiently good failure detection and reconfiguration algorithm in place, the overall reliability will be sufficiently high.

The key drawback of massive redundancy is that it is not practical in many applications. Many applications cannot afford it. This may be as a result of cost considerations (e.g., in automobiles, where even slight additions to the cost can have a major impact on marketability), or from other constraints such as power, weight, heat dissipation, and volume (e.g., in long-term space applications).

One cannot do away with redundancy completely: by definition, fault-tolerance is only possible when there is reserve capacity to be used upon failure. To be able to deploy the appropriate level of redundancy and no more (or less), two things are necessary. The first is an efficient fault-detection and reconfiguration mechanism. The second is a good resource-management algorithm which can aid in making sure that whatever redundancy is available is put to best use. Fault detection and switchover have been the staple of fault- tolerant computing for many years. The problem of efficiently managing resources, however, has received less attention [1] [9] [16].

We have developed such a resource management algorithm which selects the optimal failure recovery procedure to be employed when a processor failure is detected. We term the algorithm RAMP (Reduced State Space Markov Decision Process). It has been developed using a Markov reward model [8] and is described in greater detail in Section 4. In order to fully validate our algorithms and assess the reliability of embedded systems employing these algorithms, a simulation testbed was constructed.

In this paper, we describe the RAPIDS (Recovery Policies for real-time Distributed Systems) simulation testbed. The testbed can be used to demonstrate how system reliability can be enhanced as a result of using a particular resource management algorithm, such as the one we have developed. It can also be used to evaluate and compare the performance of systems with various task allocation schemes, task scheduling algorithms and system architecture features like the number of processors and the type of communication network. The simulator operates by taking a system specification, a task set to be "run", and allows faults to be injected to see how the system is able to recover.

The paper is organized as follows. In Section 2, we describe the basic simulator structure. Section 3 contains a discussion of how faults are injected, detected, and recovered from. Section 4 provides an overview of the RAMP algorithm. Section 5 consists of a discussion of the simulator's Graphical User Interface (GUI). The paper concludes with a brief discussion.

2. SIMULATOR STRUCTURE

The simulator is event-driven and has a layered structure as shown in Figure 19.1. It runs on one or more workstations using the PVM (Parallel Virtual Machine) software [2]. Our simulator exploits several advantages of PVM: a unified framework within which the parallelism in RAPIDS is achievable, a relatively simple and easy to program message passing interface for parallel process synchronization, and the ability to automatically spawn (and destroy) processes as required.

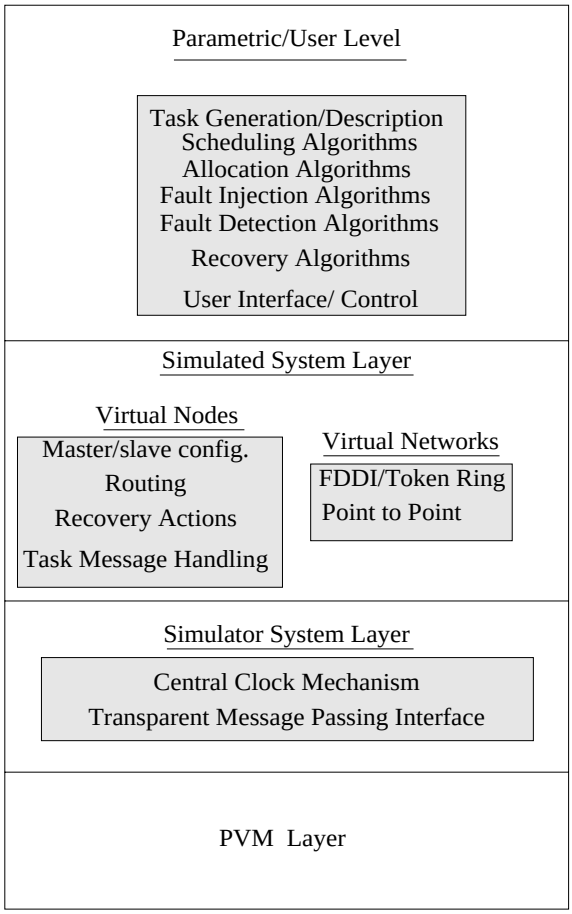

Figure 19.1 The four layers of the RAPIDS simulator.

The complete simulated system contains the following components:

- Each of the *virtual nodes* represents a processor in a distributed environment.

- A *virtual network* forms the interconnection between the virtual nodes
- A *clock* mechanism to synchronize these elements.
- A *task set* for the system to "run".

The first three components (nodes, network, and clock) are each simulated using a separate PVM process. The most important element in each of these components is an event queue that is the basis for the event-driven simulation. The task set is contained in various data structures. A more detailed description of how each component works and is implemented in RAPIDS follows in this section. Fault injection and handling of faults is covered in Section 3. The Graphical User Interface (GUI), through which the user is able to specify various system configuration parameters, is described in Section 5.

2.1 VIRTUAL NODES

Each virtual node represents a computing element on which the tasks run, as though running on a real CPU. It exchanges messages with other nodes via the virtual network. The nodes run in a master-slave configuration. All specified system tasks arrive at the master node and are allocated to the slave nodes according to the allocation algorithm. The slave nodes then schedule and execute tasks assigned to them by the master. The master is also responsible for detecting faults and overseeing the entire system recovery process.

Each virtual node maintains a list of tasks that are assigned to it, a queue of running tasks, and an event queue. The event queue is a list of events that must be performed in the near future. Possible events include the start or completion of a task, sending or receiving a message, checkpointing, or a task deadline. The event queue is processed as follows:

```
event_processing_algorithm {
 while(not end_of_mission ) {
   Send next event time to global clock.
   Wait for that time to be broadcast by the clock.
   Execute all events corresponding to that time.
   Update event queue (adding or removing events as appropriate).
 }
}
```

Apart from the next event time, the node also sends the number of PVM messages that it has sent and received during the last iteration. This information is used by the clock to ensure causality constraints are met. See Section 2.4 for more information.

The duties of each node can be classified into two areas: scheduling and running tasks, and handling communication to and from those tasks.

Scheduling and running tasks. At the start of a simulation run, the master node is initialized and sent the user-specified task set. The master, then *allocates* the tasks to the slave nodes. Currently, one of two algorithms may be used to allocate tasks:

- In the round-robin algorithm, tasks are allocated to the nodes in a round-robin fashion with no regard to balancing the processing load on each node.

- In the utilization-based algorithm, a task is allocated to the node which has the least utilization.

The user specifies which algorithm is to be used.

The node, upon receiving a task, performs the following series of actions. It generates an instance of the task and inserts two events into the event queue: a deadline of the current task iteration, and an one corresponding to when the next instance of the task must be generated. Then the task is *scheduled*. Two scheduling algorithms are currently implemented in the simulator: the Rate Monotonic (RM) algorithm [3] [5] and the Earliest Deadline First (EDF) algorithm [3] [7]. The user chooses one of them for each node in the simulated system. As with allocation algorithms, other scheduling algorithms can easily be added to the simulator.

As is discussed in Section 2.3, a task can be either a real task or a synthetic task. Execution, from the virtual nodes' perspective, is very similar for each type. The task in service is executed by placing events related to various phases of execution into the event queue, such as: message send events, message receive events, an end-of-task event, as well as the deadline event. When the end-of-task event occurs, the virtual node removes that instance of the task from the queue of running tasks and removes that task instance's deadline event from the event queue. The node then asks the scheduler for the next task to be run, and "executes" it. If the deadline event is encountered then it means that the node has not finished execution of the task instance and thus that it has missed its deadline.

Inter-task communication. In order for one task instance to send a message to another task instance, running on a different virtual node the following series of actions occur.

1. On the sending node, an event is generated that marks when the virtual node passes the message off to the virtual network, for example, at time t. The PVM delay that is involved in the transfer of a message from the node PVM process to the network PVM process is masked by not allowing the virtual clock to advance during the flight of the message.

2. The virtual network then calculates how long it will take to deliver the message to the recipient node, d, and schedules an event at $t + d$.

3. At $t + d$, the message is delivered to the recipient virtual node, and placed in a mailbox.

In order to receive a message, the recipient node does the following:

1. The recipient task alerts the virtual node that it needs, or expects, to receive a message at a particular time, τ.

2. When time τ is reached, the node checks the mailbox to see if the message has arrived:

   - If so, the task continues execution.
   - If not, then the task is considered to be blocked, waiting for the message, and is swapped out of service. When the message does finally arrive, the task is swapped back in, and allowed to preempt any lower priority task that may have been executing in the meantime.

2.2 VIRTUAL NETWORK

The virtual network simulates the entire interconnection network of the distributed system. Nodes can be connected via broadcast or point-to-point networks. A variety of protocols such as FDDI, IEEE 802.5 Token Ring [10], or IEEE1394 (Firewire) [11] are supported by RAPIDS. The virtual network is implemented as a single PVM processes. This one process maintains the state of the entire network as a series of message queues, one for each virtual node, and in addition maintains its own event queue.

In general, for the broadcast topologies, a token (or similar access control mechanism depending on the protocol) is introduced into the network and shared among the nodes according to the specified protocol. However the "token" movement is only actually simulated when a node has a packet to send. When the network is empty, the token passing is stopped, to improve the efficiency of the simulation, as otherwise the circulating token would create a large number of events to be simulated, slowing the simulation greatly. When a node has a packet to be sent, the virtual network calculates the current position of the token and introduces it in the proper node.

The first step in message delivery is the transfer of messages from the virtual nodes to the network process. Upon receipt of a message, the network calculates the transmission delay of that message. Using this delay, it schedules an event corresponding to message delivery, and delivers the message at the appropriate time.

The total transmission delay, d, (i.e. the time interval between the packet arrival at the network and its delivery to the destination) experienced by a packet is given by $d = W + S + T$ where:

- W is the time spent in the queue before starting transmission.
- S is the service time. This is the time taken to transmit all the bits in the packet (proportional to the size of the packet). It is dependent on the bandwidth of the channel.
- T is the propagation delay. This is the time taken for a single bit to travel to the destination node. It is a characteristic of the channel used and is roughly proportional to the wire length between the source and the destination.

A faithful simulation of the interconnect network is crucial in obtaining a realistic cost of message passing, task migration etc. It affects the estimation of overheads involved in doing the recovery and reconfiguration actions and consequently, the performance of the overall system.

2.3 TASKS

These represent the virtual jobs or processes that are to be executed by the system. Tasks are scheduled and run in the environment specified by the RAPIDS user. All tasks are characterized by some general parameters:

- Period is the interval between task releases.
- Deadline is the time, relative to the release time, by which the task must have completed its execution.
- Execution time is the anticipated time each iteration takes to complete.
- Release time is relative to the start of the period and specifies when the task is to begin execution.
- Redundancy specifies the number of nodes on which the task should be replicated for fault-tolerance.
- Priority specifies the priority of this task relative to others in the system.
- Checkpoint size specifies how large the task is memory-wise, and affects how long it will take to move the checkpoint to a good node in case of a fault.

Apart from these characteristics, tasks may be either real or synthetic.

Synthetic Tasks. These tasks are not real software tasks but are represented solely by a set of attributes that simulate a tasks runtime [13]. As above, they are: the period, deadline, phase, redundancy, and priority. In addition, messages may be specified to be sent (or received) from one task to another. Messages are defined by their size, and the time at which they are to be sent and received relative to the start of execution.

Real Tasks. Running real tasks can provide "real" implications of deadline misses. This was the motivation behind the inclusion of real tasks thus making RAPIDS both event-driven as well as execution-driven. In order to obtain the most consistent results the real tasks should be run on a homogenous system. As an example, the RTHT Benchmark was implemented in conjunction with RAPIDS. The RTHT Benchmark is a target-tracking application, developed at Honeywell [17] [18]. We have integrated it with RAPIDS such that it runs on the physical processor(s) hosting the RAPIDS simulator, but is controlled by the simulated system. Timing values between various events in task execution are obtained by observing the timing as the "real" application executes on the CPU. Thus the original application computation is actually performed. This has been done by modifying the RTHT to run in time-slices that are scheduled by the simulated virtual nodes. If or when the RTHT task (in the virtual node) is preempted by a higher priority task the real processing is halted until the task is put back in service.

2.4 GLOBAL EVENT PROCESSING AND THE CENTRAL CLOCK

In any simulation, time is an important concept. In a parallel simulation, such as RAPIDS, it is vital to make sure that as much parallelism as possible is achieved, while making sure that correctness is maintained.

Virtual nodes and the simulated network have their own local version of *virtual time* and run independently of each other except when there is a need for interaction with other objects (PVM process). Interaction between the virtual objects is defined by PVM messages. As PVM provides no guarantees as to exactly how fast or slow a particular message might be delivered, we must take additional steps to ensure that messages obey causality constraints of the application. If an event A must happen only after another event B, it is imperative to obey this causality constraint even if the events are executed in separate processes [4]. However, as RAPIDS is made of distinct processes, with potentially unrelated ideas of virtual time (for parallelization), it might be possible for these causality rules to be violated. Thus synchronization has to be introduced into the parallel, distributed simulation systems to ensure correct-

ness. We present below the central clock mechanism that was chosen to solve this synchronization issue in the simulator.

RAPIDS is basically an event-driven simulator. Time does not advance continuously, but in discrete quanta, according to the expected next event. In each step of synchronization the next earliest event among all PVM processes is determined and the virtual time is set to the value of this event. The advantage of the event-driven approach is that it is economical in the amount of the time taken for the completion of the simulation. One of the difficulties in such simulation is maintaining same notion of global time across the system. RAPIDS uses a variation of the Breathing Time Buckets technique [12].

Operation. The central clock unit is represented by a PVM process. It handles a list of all the objects (nodes and network) and the next event time (NET) for each virtual object. Initially the central clock holds a value of zero, but during initialization phase each objects informs the clock about their next event times. The central clock then finds the minimum of these values, sets this value to be the next global time and broadcasts it to the objects. The central clock uses the following algorithm for updating virtual time:

```
clock_updation_algorithm {
 while(not end_of_mission) {
   Receive NET values from virtual nodes and the network;
   Update the list of NETs;
   Find the minimum of the NETs;
   If (calculated minimum > latest global virtual time) {
     If (there are any PVM messages "in-flight")
       Wait until they are received;
     Broadcast the minimum as the current global virtual time;
   }
 }
}
```

Node-Clock Interaction. Each node orders its events according to their time – early events before later ones. When the node receives a global time update that corresponds to its NET, it fetches this event and executes the operation that it is supposed to do at that time. Then it updates the central clock with the time of its next event and the number of PVM messages it has sent or received during this iteration. The clock finds out whether there are any PVM messages "in flight" by equating the number of messages sent to the number of messages that have been received at each node. Since the central clock waits for this update, it will not update the global time until the last PVM message has been received. This assures us that the global time is always consistent.

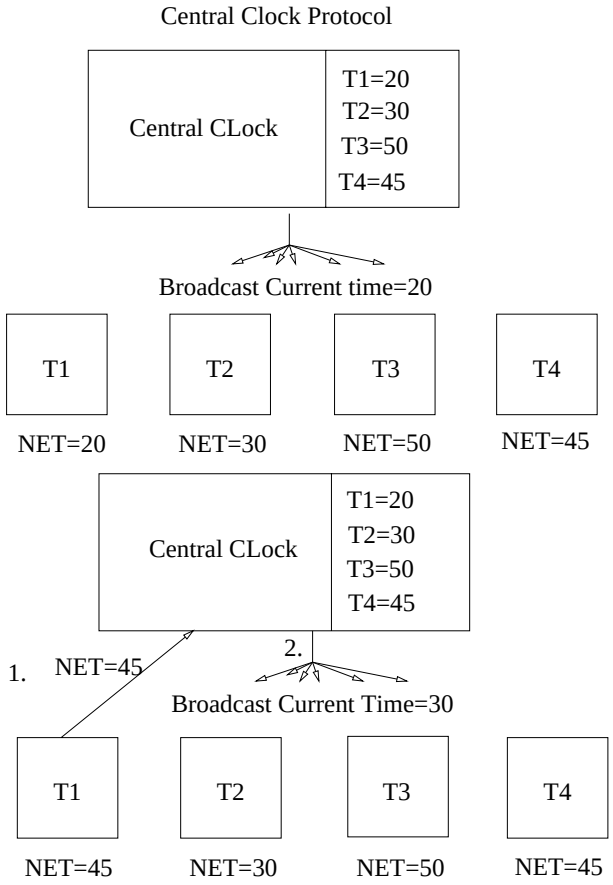

Figure 19.2 Illustration of the clock protocol.

Message Passing. The virtual network must also interact with the central clock in order to ensure correct message passing. The network cannot prepare the order of events in advance since messages are not predictable. To solve this problem the central clock is informed about any message passing that occurs through the network. The clock is not allowed to tick when messages sent out by the nodes are not yet received at the network, or vice-versa. An example of global time broadcast can be seen in Figure 19.2. The example includes the following steps:

1. The central clock initially has next event times of the four objects involved in the simulation as 20, 30, 50 and 45, respectively. It calculates a minima of these times and broadcasts that value as Global Current Time. Thus, 20 is broadcast as the current time.

2. When this value is received, all events with timestamps equal to this value are processed at each of the objects. The object which had this value as NET then updates the central clock with its new value of NET. Any message passing involved is also reported along with this value.

3. When the central clock receives this message, it updates the NET of the object from which the update message is received and then waits for any PVM messages that may still be "in transit". The clock then computes a new global virtual time and broadcasts it.

3. SIMULATION OF FAULTS

One of the major goals of the RAPIDS simulator is to observe the behavior of the simulated system in the presence of faults. Thus we have provided mechanisms by which the user may inject faults into the simulated system in order to see how well it recovers.

3.1 FAULT INJECTION

The Fault Injection module is responsible for injecting faults into various components of the system. This is basically accomplished by telling the particular component that it has received a fault. The component then simulates the faulty state, forcing the system to react.

Currently only the nodes are susceptible to failure but failure can be extended to cover communication links as well. Faults can be specified in two ways: first by specifying the Poisson rates for transient and permanent faults for each node, and second by specifying a table of one time faults for each node.

3.2 FAULT DETECTION AND RECOVERY

In order to experiment with fault tolerance in the system the user should make use of some form of redundancy. RAPIDS offers static redundancy by replicating tasks and dynamic redundancy through checkpointing. The master node is responsible for detecting the failure of a slave, and invoking the appropriate recovery actions. The user can input the recovery action that is to be taken upon each failure.

Detection. Each node also periodically sends an "I am alive" message to the master. Fault detection is triggered by the non-receipt of such a message by the master.

Recovery. Each fault-free slave node periodically records its state in a checkpoint. The checkpoint of a node consists of the following information: The time at which the checkpoint was taken, the set of subtasks that were running

on that node, the actual state of each of these subtask instances. The information is assumed to be stored reliably, in that it is not corrupted by the nodes going faulty or by any other event. The checkpoints occur at user-specified intervals.

On detecting a fault, the master invokes a recovery algorithm to decide on the appropriate recovery action. There are static and dynamic recovery algorithms that the user may choose from.

- A static recovery algorithm simply performs the same action, or series of actions for recovery, regardless of system state.
- A dynamic recovery algorithm chooses which recovery action to try based on any number of variables concerning the state of the system, time and location of the fault.

The recovery algorithm will then specify one or a combination of three basic recovery actions as follows:

- Retry - Restart execution on the same node from a consistent state as recorded in the latest checkpoint.
- Replace - Use the latest checkpoint from the faulty node to start executing all its tasks on a spare node.
- Disconnect - Use the latest checkpoint from the faulty node to distribute its tasks to the other non-faulty nodes in the system.

The three recovery actions have different penalties in terms of the time taken to perform them. The user can specify these values before the start of the simulation, and also specify the algorithm to be used to generate the recovery actions during the mission time. One of them is the optimal recovery policy algorithm, RAMP, described below.

4. REDUCED STATE SPACE MARKOV RECOVERY PROCESS - RAMP

When a fault occurs, the most suitable recovery action must be followed to achieve high reliability. To satisfy the service requirements of real-time tasks with deadlines, it might seem intuitive that a more powerful system would give the best result. However, higher processing capacity means a more complex system, more processing modules, and more electronic parts, which may result in more frequent faults and a higher risk that the system will fail to complete the real-time tasks prior to their deadline. Therefore, a dynamic, optimal recovery policy for complex real-time systems is needed. Such an algorithm was developed at the University of Massachusetts [14], and has been adapted for use with RAPIDS. Operation of the algorithm is explained.

The RAMP algorithm uses Markovian decision theory. Based on the system state and such parameters as fault rates, transient fault duration, checkpoint interval, and anticipated workload, it determines which of the three recovery actions should be taken to maximize reliablility. Reliability is the probability with which the system can service all tasks within their deadline throughout the system mission time.

The RAMP algorithm [16] uses a state aggregation technique to reduce the system state space to a sufficiently small size without significantly compromising precision. A dynamic programming technique [6] is then used to compute the optimal recovery action. This computation is done *a priori*, before the simulation is started. Given the reduced system space, the computation is done for all possible system configurations for the system mission time.

The output of this pre-simulation run consists of the set of system states and the actions to take for each possible configuration at each point of time (with a desired resolution) in the mission. During the simulation itself, RAMP runs in the background, waiting for a fault to occur. In order to recover from a fault, the master node passes the current state of the system as a parameter to the algorithm. The algorithm then selects the appropriate recovery action by consulting the results from the pre-simulation run and returns this action to the master node for implementation.

5. GRAPHICAL USER INTERFACE (GUI)

The GUI provides a convenient interface to change and view the various parameters of the simulator. The configuration of RAPIDS is broken into several subsections of the GUI.

- The Configuration section covers the system and network topology, allocation and scheduling algorithms, and recovery policies to be used.
- The system workload is specified in the Task Generator section.
- Faults that will strike the system are configured in the Fault Generator section.

Once the system to be simulated is completely specified, the user can simply press a button to start the simulation. The user can monitor the state of the system during simulation with three windows in the Information section of the GUI.

- The Schedule Window displays information regarding the execution of the tasks at each node. It shows the times when the subtasks have started, finished execution, been preempted, received/sent messages, checkpointed, when the nodes are struck by faults etc.

- The System Performance Window displays the performance of the system in terms of the number of subtasks that have started, finished successfully, missed their deadline and preempted during their execution.

- The Task Allocation Window displays how tasks have been allocated to the slave nodes by the master.

In addition, all simulation output is also provided to the user in the form of a parsable text file, should script-automated comparisons between runs be required. The user can then easily compare the performance of one system configuration to another, in order to validate a configuration or recovery policy.

6. DISCUSSION

We have described a distributed simulator testbed whose purpose is the validation of resource management algorithms. Such algorithms are likely to increase in importance in the future as embedded systems are increasingly being used where there are considerable constraints on costs. Here every unit of redundancy must "pay its way".

The simulator can be used to demonstrate the improvements in performance that are possible when efficient resource-management policies are used. The role of the simulator is to help designers identify whether such policies make a sufficiently large difference to the system reliability as to be worth using for a given application; and to fine-tune the design of the architecture and operating system.

The core of the simulator is designed such that it can be easily extended to validate other aspects of a distributed real-time system. For example, link failures could be introduced, new and different scheduling or allocation algorithms can be implemented, and even networking protocols maybe be added. Importance sampling has been incorporated into RAPIDS in order to decrease the amount of simulation time required to validate the effectiveness of fault recovery policies [15].

This work was supported in part by DARPA, under order B855, and managed by the Space and Naval Warfare Systems Command under Contract No: 00039-94-C-0165.

References

[1] Berg M. and I. Koren. 1987. On Switching Policies for Modular Fault-Tolerant Computing Systems, IEEE Trans. Computers, Vol. C-36, pp. 1052-1062, Sept. 1987.

[2] Geist Al; A. Beguelin; J. Dongarra; W. Jiang; R.Manchek; V.Suderam. 1994. PVM: Parallel Virtual Machine, MIT Press.

[3] Krishna C. M. and K. G. Shin. 1997. Real-Time Systems, McGraw-Hill.

[4] Lamport L. 1978. Time, Clocks, and the Ordering of Events in a Distributed System, Communications of the ACM, Volume 21, 7, 1978.

[5] Liu C. L. and J. W. Layland. 1973. Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment Journal of the ACM, Volume 20, 1973.

[6] Puterman M. L. 1994. Markov Decision Processes, John Wiley & Sons Inc.

[7] Ramamritham K. and J. Stankovic. 1984. Dynamic task scheduling in distributed hard real-time systems, IEEE Software, Volume 1.

[8] Ross S. M. 1970. Applied Probability Models with Optimization Applications, San Fransisco: Holden-Day.

[9] Shin K. G.; Y. H. Lee; and C. M. Krishna. 1989. Optimal Dynamic Control of Resources in a Distributed Fault- Tolerant System, IEEE Trans. Software Eng., Vol. 15, October 1989.

[10] Stallings W. 1988. Handbook of Computer-Communca- tions Standards, Howard W. Sams & Co.

[11] Anderson D. 1998. Firewire System Architecture, , Addison Wesley.

[12] Steinman J. S. 1993. Breathing Time Warp, Proceedings of the 1993 Workshop on Parallel and Distributed Simulation.

[13] Toutireddy K. K. 1996. A Testbed for Fault Tolerant Real- Time Systems, M.S. Thesis, Univ. of Mass. Amherst.

[14] Yu K. 1996. RAMP and the Dynamic Recovery and Reconfiguration of a Distributed Real-Time System, Ph.D. Dissertation, Univ. of Mass. Amherst MA.

[15] Durairaj G. 1999. Evaluating the Reliability of Distributed Real-Time Systems, M.S. Thesis, Univ. of Mass., Amherst MA.

[16] Yu K. and I. Koren, 1995. Reliability Enhancement of Real- Time Multiprocessor Systems through Dynamic Reconfiguration. Fault-Tolerant Parallel and Distributed Systems, D. Pradhan and D. Avresky (Editors), pp. 161- 168, IEEE Computer Society Press, Los Alamitos, CA, 1995.

[17] B. VanVoorst, R. Jha, L. Pires, M. Muhammad. Implementation and Results of Hypothesis Testing from the C^3I Parallel Benchmark Suite. Proceedings of the 11th International Parallel Processing Symposium, 1997.

[18] D.A. Castanon and R. Jha. Multi-Hypothesis Tracking (Draft). DARPA Real-Time Benchmarks, Technical Information Report (A006), 1997.