

**EVALUATING THE RELIABILITY OF DISTRIBUTED REAL-TIME
SYSTEMS**

A Thesis Presented

by

GOPINATH DURAIRAJ

Submitted to the Graduate School of the
University of Massachusetts Amherst in partial fulfillment
of the requirements for the degree of

MASTER OF SCIENCE IN ELECTRICAL AND COMPUTER ENGINEERING

February 1999

Department of Electrical and Computer Engineering.

**EVALUATING THE RELIABILITY OF DISTRIBUTED REAL-TIME
SYSTEMS**

A Thesis Presented

by

GOPINATH DURAIRAJ

Approved as to style and content by:

C.M. Krishna, Chair

Israel Koren, Member

Ian Harris, Member

Seshu B. Desu, Department Head
Electrical and Computer Engineering

ACKNOWLEDGMENTS

I wish to thank Prof. C. M. Krishna for his invaluable guidance and his patience in helping me shape this work at each step. I thank him dearly for providing me an opportunity to work with him. I have learnt immensely from his lectures and our numerous discussions. I consider myself fortunate in having him as my student advisor.

I wish to thank Prof. I. Koren for inspiring me and helping me focus better on the research topic of interest. I thank him for sparing me his invaluable time and helping me in understanding this wonderful field. It is a great privilege to have worked with him. Without his guidance, this work would not have been possible.

I wish to thank Vijay Lakamraju, Osman Unsal and Josh Haines for their valuable insights and their help in shaping this work. The development of the simulator was a team effort and each one of the above have contributed heavily towards its success. I wish to thank them for sharing with me their wonderful ideas and for making my stay at UMass a memorable one.

I wish to thank Seth Muthukaruppan for being a constant source of encouragement and support. He helped me in keeping myself focussed even when I was away from school.

ABSTRACT

EVALUATING THE RELIABILITY OF DISTRIBUTED REAL-TIME SYSTEMS

FEBRUARY 1999

GOPINATH DURAIRAJ, B.E., R.E.C. TIRUCHIRAPALLI, INDIA

M.S., UNIVERSITY OF MASSACHUSETTS AMHERST

Directed by: Professor C. M. Krishna

Computers are increasingly being used in life-critical applications and their need to be reliable also increases dramatically. A distributed system architecture is a very attractive proposition to meet these reliability and fault tolerance requirements. Such a system is very complicated and needs to be validated before building and deploying it.

A simulator test bed is built at UMass to model a variety of such systems quickly from a few basic building blocks. Since these systems are very complex, many independent simulation runs are needed to estimate their reliability to a reasonable level of confidence.

Importance sampling is a technique commonly used to speed up such rare event simulations. In this technique, the probabilistic dynamics of the system are altered so that the rare events (such as the system failure) occur much more frequently. The sample outputs then need to be adjusted to compensate for the bias introduced.

This thesis talks about building the simulator test bed and concentrates on incorporating and validating importance sampling to speed up the reliability estimations.

Two importance sampling heuristics called ‘forcing’ and ‘failure biasing’ are incorporated in the test bed. The implementation is validated by comparing the reliability estimates with that of the normal simulation. The effect of the failure bias on the dynamics of the scheme are also investigated to provide some guidance on choosing the failure bias.

The tool is applied to see how the reliability estimates of a system changes with the change in failure rate. Finally the tool is used to demonstrate that using an optimal failure recovery algorithm can significantly improve the reliability of a distributed real-time system.

TABLE OF CONTENTS

ACKNOWLEDGMENTS	iii
ABSTRACT	iv
LIST OF TABLES	x
LIST OF FIGURES	xi
CHAPTER	
1. INTRODUCTION	1
1.1 Thesis goal	3
1.2 Thesis Outline	4
2. IMPORTANCE SAMPLING	5
2.1 Introduction	5
2.2 Basic Idea	7
2.3 Choosing an Optimal Sampling Density	9
2.4 Likelihood Ratio	10
2.5 Failure Biasing	12
2.5.1 Simple Failure Biasing	12

2.5.2	Balanced Failure Biasing	13
2.6	Forcing	15
2.7	Other Importance Sampling Results	16
2.8	A Sample Application	18
2.9	Other Variance Reduction Techniques	21
3.	IMPLEMENTATION OF THE TEST BED	24
3.1	Simulation Model	24
3.1.1	Computing Nodes	26
3.1.2	Tasks and Messages	27
3.1.3	Scheduling and Allocation Algorithms	28
3.1.4	Network	30
3.1.5	Fault Injection	32
3.2	Distributed Simulation Issues	32
3.3	RAMP Algorithm	33
3.4	Simulator Implementation	35
3.4.1	Platform	35
3.4.2	Implementation Model	36
3.4.3	Other Simulator Entities	38

3.4.4	Fault Injection	40
3.4.5	Fault Detection and Recovery	41
4.	SIMULATION ANALYSIS	44
4.1	Reliability	44
4.2	Normal Simulation for Estimating Reliability	45
5.	IMPLEMENTING IMPORTANCE SAMPLING	48
5.1	Outline	48
5.2	Implementation	52
5.2.1	Forced transitions	53
5.2.2	Failure biasing	54
5.2.3	Analysis	56
5.3	Expected Behaviour of Importance Sampling	57
5.4	Validation of the Model	61
5.5	Selecting the Bias Parameter	67
5.6	Some Typical Usages	69
5.6.1	Varying the Transient Failure Rates	69
5.6.2	Comparing the Recovery Policies	70
6.	CONCLUSIONS	72

BIBLIOGRAPHY 74

LIST OF TABLES

1. Normal Simulation	64
2. Importance Sampling with Bias 0.3	64
3. Acceleration Factor	66
4. Sample variance for different failure bias values	68
5. Effect of the Transient Failure Rates on the Unreliability	69
6. Comparison of the Recovery Policies	70

LIST OF FIGURES

1. A typical distributed real-time system	25
2. The Implementation Model	37
3. The Implementation Model	49
4. RE of Normal Simulation and Importance Sampling	58

CHAPTER 1

INTRODUCTION

Computers are wonderful machines. They are increasingly being used in ways never imagined possible before. From their traditional uses in number crunching and huge databases they have evolved rapidly and are now used to control everything from cars to factories and fly-by-wire aircraft.

As computers begin to be used in these life-critical applications, their need to be reliable increases dramatically. These computers are expected to perform their tasks in a time-bounded fashion. It is not enough if these machines deliver outputs that are logically correct they should also be timely. Catastrophic results can occur if these task deadlines are not met (Imagine an aircraft not putting out its wheels when it is fast approaching the runway). These computers are also increasingly expected to perform their best even in the presence of faults. They should be able to tolerate faults in hardware, software or anywhere!

A distributed system architecture is a very attractive proposition to meet these reliability and fault tolerance requirements. It has multiple, independent computing entities, which provide scalable computing power. Since they are independent, there is a high chance that the faults will be isolated and localized to a subset of nodes. When a fault is detected in a node, the tasks from that node can be moved to other

active nodes that can handle the additional load. Such an architecture can also exploit the potential parallelism that might exist among the various units. Does it look like we have solved all the tough problems? Alas, there is no such thing as a free lunch! The distributed system architecture is very complex and there are quite a few issues to be solved.

- We need good task admission, allocation/scheduling algorithms to make sure that the accepted tasks meet their deadlines.
- We need a good resource management algorithm to efficiently manage the available redundancy. When a node fails, there are a few possible recovery actions. An optimal failure recovery action must be chosen so that the possibility of a task missing its deadline is minimized.
- We need a good network architecture to interconnect the distributed nodes and to provide high connectivity and performance even in the presence of faults. This network should also provide for predictable delays as the tasks might be exchanging messages that are time constrained. As the number of computing nodes are increased, the network should also be able to scale well.

Each of the above problems constitute a major research area in itself and a lot of work has gone into solving them. At UMass an optimal resource management algorithm [30] has been developed to suggest the failure recovery action that will minimize the probability of a critical task missing its deadline.

Once we have some solutions for each of the problems, we would like to put together a complete system and find out the reliability of the system to validate the

design. Such a system is very complex and the algorithms/policies might interact in ways that may not be obvious. The reliability evaluation is usually carried out by analytical techniques or simulation. Analytical modeling is very tough in all but the simplest of cases and simulation is the preferred way to evaluate complex models.

1.1 Thesis goal

We have endeavored to create a simulator test bed that will facilitate building such a complex system and to evaluate its reliability. The simulator test bed will enable the user to model a variety of systems quickly using a few basic building blocks. Adding new building blocks as plug-ins should also be easy. Examples of such building blocks include scheduling algorithms, allocation algorithms, failure recovery policies, new network types etc.

Once we have created the system under evaluation, we need efficient techniques to evaluate its reliability. Means should be found to integrate such techniques with the simulator test bed. This will enable the user to configure a system of choice and find out its reliability quickly.

Since the modeled systems are very complex and their reliability is quite high, many independent simulation runs will be needed to estimate their reliability to a reasonable level of confidence. Importance sampling is a technique commonly used to speed up such rare event simulations. In this technique, the probabilistic dynamics of the system are altered so that the rare events of interest occur much more frequently. The sample outputs then need to be adjusted to compensate for the bias introduced.

This thesis talks about building such a simulator test bed and concentrates on incorporating importance sampling in the test bed to speed up the reliability estimations.

1.2 Thesis Outline

The rest of the thesis is organized as follows: Chapter 2 talks about efficient simulation techniques for reducing the sample variance. Chapter 3 talks about the simulation model and some of the interesting issues involved in the design and implementation of such a simulator.

Chapter 4 clarifies some issues regarding reliability evaluation and gives an overview of the simulation analysis that will be used. Chapter 5 talks in detail about the implementation of importance sampling, its validation and the effect of the bias parameter.

CHAPTER 2

IMPORTANCE SAMPLING

Different measures are used to evaluate the modeled systems depending upon whether they are mission oriented systems or continuously operating systems. Some of the dependability measures that are very commonly used are steady-state availability, reliability, mean time to failure (MTTF), expected interval availability etc.

We will discuss a technique called Importance Sampling which is widely employed to speed up rare-event simulations in queueing and reliability models. A variety of heuristics have been proposed to implement importance sampling to estimate the different measures mentioned above. Depending upon the measure that we are interested in estimating, we could choose to implement some of them. Since we are primarily interested in measuring the reliability of the system, we will look closely at those heuristics that are applicable.

2.1 Introduction

To analyze the given system for reliability, we need to model the system in a way that closely mirrors reality and which makes the analysis simple. The system is considered to be a collection of components which can fail and possibly get repaired.

It is considered operational if at any given moment the operational components satisfy some minimum system operational requirements. The failure times and the repair times of the components are assumed to be exponentially distributed so that the system may be modeled as a continuous time Markov chain (CTMC).

Typically numerical methods are used to solve Markov chains. Although many modeling packages have been built (eg [10]), the size of the system modeled is typically small because the number of states in the system increases exponentially with the number of components. Techniques such as state lumping and unlumping and state aggregation and bounding can reduce the size of the state space substantially. However large systems with a large number of redundant components are still out of the range of the solution capabilities of current numerical methods primarily due to storage or computational limitations.

An alternative approach for the solution of large models is Monte Carlo simulation. By nature this approach has the immediate advantage of having relatively small storage requirements. In addition it might be easier to model very complex systems using simulation. On the other hand, since the failure events are very rare, it is apparent that the analysis by simulation of large models with a high degree of redundancy will require many long independent replications in order to attain reasonable confidence intervals. Our goal is to investigate the variance reduction methods that might be easily implemented in the current model.

In importance sampling, we change the probabilistic dynamics of the system for simulation purposes. The new probability measure introduces system failures to occur more frequently. We then make adjustments to the sample outputs to unbiased the

estimator before computing it. A good reference for the mechanics of this technique is Hammersley and Handscomb [13]. Generalizations to stochastic systems are given in Glynn and Iglehart [9]. The heuristic of failure biasing was first proposed in Lewis and Bohm [18] in the context of reliability estimation of nuclear reactors. In Goyal, Heidelberger and Shahabuddin [11] it was adapted to the estimation of unavailability for highly reliable Markovian systems. In Shahabuddin et al. [23], it was used for the estimation of the MTTF using a regenerative method. Generalizations of these heuristics along with new ones have been investigated in the unifying paper Goyal et al [12].

2.2 Basic Idea

Let X be the random variable that has the probability density function $p(x)$. We are interested in estimating the probability θ that X is in some set A (the failed state)

$$\theta = \int_{-\infty}^{+\infty} 1_{\{x \in A\}} p(x) dx = E_p[1_{\{X \in A\}}]$$

where the subscript p denotes sampling from the density p and $1_{x \in A}$ is the indicator of the set A . i.e.,

$$1_{\{x \in A\}} = \begin{cases} 1 & \text{if } x \in A \text{ and} \\ 0 & \text{Otherwise} \end{cases}$$

Consider estimating θ by simulation. The standard approach would be to draw n samples X_1, \dots, X_n from the density p , set $I_i = 1_{X_i \in A}$ and form the estimate

$$\bar{X} = \frac{1}{n} \sum_{i=1}^n I_i$$

If $E(X^2) < \infty$ then using the central limit theorem, we can construct the standard confidence interval on the estimator. This confidence interval is given by $[\bar{X} - HW, \bar{X} + HW]$ where HW is called the half width and is given by $z_{\alpha/2}S/\sqrt{n}$ (the quantity $z_{\alpha/2}$ is the 100(1 - $\alpha/2$) percentile point of the standard normal distribution and S is the standard deviation of the estimator). **The relative error (RE)** is defined to be HW/θ .

Multiplying and dividing the integrand by another density function $p'(x)$, we obtain

$$\begin{aligned}
 \theta &= \int 1_{\{x \in A\}} \frac{p(x)}{p'(x)} p'(x) dx \\
 &= E_{p'} \left[1_{\{X \in A\}} \frac{p(X)}{p'(X)} \right] \\
 &= E_{p'} [1_{\{X \in A\}} L(X)]
 \end{aligned} \tag{2.1}$$

where $L(x) = p(x)/p'(x)$ is called the **likelihood ratio** and the subscript p' denotes sampling from the density p' . The above equation is valid for any density p' provided that $p'(x) > 0$ for all $x \in A$ such that $p(x) > 0$. i.e., a non-zero feasible sample under density p must also be non-zero feasible sample under p' .

The above equation is the key for importance sampling. Draw n samples X_1, \dots, X_n using the density P' and define $\delta_i = L(X_i)I_i$. Then by equation 2.1 $E_{p'}[\delta_n] = \theta$. Thus an unbiased estimate of θ is given by

$$\bar{X}(p') = \frac{1}{n} \sum_{i=1}^n \delta_i = \frac{1}{n} \sum_{i=1}^n I_i L(X_i)$$

i.e., θ can be estimated by simulating a random variable with a different density and then unbiaseding the output (I_i) by multiplying by the likelihood ratio. Sampling with a different density is sometimes called a “change of measure” and the density p' is called the importance sampling density.

2.3 Choosing an Optimal Sampling Density

Since essentially any density p' can be used for sampling, what is the optimal density i.e., what is the density that minimizes the variance of $\bar{X}(p')$? Selecting $p'(x) \equiv p^*(x)$ as follows

$$p^*(x) = \begin{cases} p(x)/\theta & \text{for } x \in A \text{ and} \\ 0 & \text{Otherwise} \end{cases}$$

has the property of making $\delta_i = I_i P(X_i)/P^*(X_i) = \theta$ with probability one. Since the variance of a constant is zero, $p^*(x)$ is the optimal change of measure. However, there are several practical problems with trying to sample from this optimal density p^* . First, it explicitly depends upon θ , the unknown quantity that we are trying to estimate. If in fact θ were known, there would be no need to run the simulation experiment at all. Second, even if θ were known, it might be impractical to sample efficiently from p^* .

Since the optimal change of measure is not feasible, how should one go about choosing a good importance sampling change of measure? Since $E_{p'}[\delta_i] = \theta$ for any density p' , reducing the variance of the estimator corresponds to selecting a density

p' that reduces the second moment of δ_i

$$\begin{aligned} E_{p'}[Z_i^2] &= E_{p'}[(I_i L(X_i))^2] \\ &= \int 1_{x \in A} \left(\frac{p(x)}{p'(x)} \right)^2 p'(x) dx \\ &= \int 1_{x \in A} \frac{p(x)}{p'(x)} p(x) dx \\ &= E_p[I_i L(X_i)] \end{aligned}$$

Thus to reduce the variance, we want to make the likelihood ratio $p(x)/p'(x)$ small on the set A. Since A is a rare event, roughly speaking, $p(x)$ is small on A. Thus to make the likelihood ratio small on A, we should pick p' so that $p'(x)$ is large on A. i.e., the change of measure should be chosen to make the event A more likely to occur.

Importance sampling does not always lead to a reduction in variance and can produce arbitrarily bad results if not applied carefully. Essentially all work on using importance sampling in practical applications deals with choosing an importance sampling distribution that leads to actual variance reduction.

2.4 Likelihood Ratio

In order to apply importance sampling, it must be possible to compute the relevant likelihood ratio. Essentially, each time a change of measure is performed, the likelihood ratio for that variable is computed (using the original and new densities) and incorporated into a running product.

In the case of sampling from a single probability density function as described above, the likelihood ratio $L(X) = P(X)/P'(X)$ This equation is also valid if X is drawn from a discrete distribution. i.e., if

$$P(X = a_i) = P(a_i) \quad i = 1, \dots, n \text{ and}$$

$$P'(X = a_i) = P'(a_i) \quad i = 1, \dots, n$$

We require that $P'(a_i) > 0$ if $P(a_i) > 0$ but note that we can have $P'(a_i) > 0$ even if $P(a_i) = 0$ since the likelihood ratio is zero in this case. i.e., no weight is given to an impossible (under p) sample path. Suppose $\mathbf{X} = (\mathbf{X}_1, \dots, \mathbf{X}_n)$ is a random vector where X_i is drawn from density $P_i(x)$ and X_i is independent of $X_j (j \neq i)$. If under importance sampling, X_i is drawn from density $P'_i(x)$ and again X_i is independent of $X_j (j \neq i)$, then

$$L(X) = \prod_{i=1}^n \frac{P(X_i)}{P'(X_i)}$$

Suppose $\{X_i, i \geq 0\}$ is a discrete time Markov chain (DTMC) on the state space of non-negative integers where X_0 has the (initial) distribution $P_0(i)$ and the step transition probabilities are given by $P(i, j) = P(X_m = j | X_{m-1} = i)$. Let $\mathbf{X}_m = (\mathbf{X}_0, \dots, \mathbf{X}_m)$. If, under importance sampling, X_0 is drawn from $P'_0(i)$ and the process is generated with the one-step transition probabilities $P'(i, j)$ then

$$L(X_m) = \frac{p_0(X_0)}{p'_0(X_0)} \prod_{i=1}^m \frac{P(X_{i-1}, X_i)}{P'(X_{i-1}, X_i)}$$

2.5 Failure Biasing

In the last few sections, we have seen the theoretical foundations of importance sampling. There are quite a few heuristics for the implementation of this technique, applicable for estimating different dependability measures. Failure biasing is an important heuristic used heavily in the estimation of transient measures such as reliability, mean time to failure etc.

For estimating the transient measures, we would like to apply importance sampling so as to sample most often from the most likely paths to failure. However, in complex systems, it may not be easy to identify these most likely paths. Thus we need to develop heuristics that are simple to implement and search many different paths to failure. It should sample often enough from the most likely failure paths so as to obtain variance reduction. Failure biasing is designed to do this.

The objective of the biasing is to derive results with reduced variance. This can be accomplished by causing more trials to contribute to the result than in the binomial case. As this happens, there will be fewer zero weight trials, but those that do contribute will have weights much smaller than one.

Failure biasing has a few variants that work well for certain types of systems. We will now look at a few important ones.

2.5.1 Simple Failure Biasing

Simple failure biasing was introduced in Lewis and Bohm [18]. At each transition of the system state, it biases the system towards more faults so as to drive the system

to the failure state. Whenever such a bias is made, the corresponding likelihood ratio is calculated and updated into a running product.

Let the failure and repair rates of a component i be exponentially distributed with rates λ_i and μ_i respectively. The total failure rate out of a particular state is the sum of the failure rates of the currently active components. Let us assume this is λ . Similarly the total repair rate out of the state is the sum of the repair rates of the components that are suffering a temporary fault. Let us call this quantity μ . γ is the total transition rate out of the current state and is equal to the sum of λ and μ .

In this technique, the probability of an additional failure is set to be a fixed probability ϕ . Typically ϕ is chosen to be in the range $0.20 \leq \phi \leq 0.80$. Note that in the original system the probability of this event is λ/γ . Since the repair rates are usually orders of magnitude greater than the failure rates, this ratio is very small. Thus the probabilistic dynamics of the system are altered with a heavy bias towards more failures.

When a failure transition is selected, the component getting the failure j is selected proportional to the original transition rates, i.e., with probability λ_j/λ . If a repair transition is selected, with probability $(1 - \phi)$, then the component getting repaired j is selected with probability μ_j/μ .

2.5.2 Balanced Failure Biasing

The above mentioned technique of simple failure biasing works very well for balanced systems. While the simple failure biasing takes the system to the set of failure

states with reasonable probability, it does not push the system along the more common failure paths often enough.

Consider a system with two types of components. There is one component of type 1 that has failure rate ϵ^2 and three components of type 2 that each have failure rate ϵ . The system is considered operational if at least one component of each type is operational. Under simple failure biasing, it is a type 1 failure with probability $\epsilon^2/(N_2\epsilon + \epsilon^2)$ where N_2 is the number of operational type 2 components; this probability is of order ϵ . Similarly, the probability of a type 2 failure is $(1 - O(\epsilon))$.

Thus, under simple failure biasing, when the system ends up in a failure state, most of the time it gets there by having three type 2 failures. It only rarely ends up in a state in which component 1 is failed. However, the path with a single component 1 failure is the most likely path to system failure; its probability is of order ϵ , whereas any other system failure path has a much smaller probability of the order ϵ^2 . Thus, while simple failure biasing takes the system to the set of failure states with reasonable probability, it does not push the system along the right failure path often enough. The result of this is that simple failure biasing applied to unbalanced systems may result in estimates having unbounded relative error.

To overcome this, another variant of failure biasing called “balanced failure biasing” was introduced [24] and was shown to result in bounded relative error.

In balanced failure biasing, a failure transition is again chosen with a probability ϕ . Now however, given that a failure event has occurred, the probabilities of all failure transitions are equalized, i.e., a failure transition from i to j is conditionally selected with probability $1/F(i)$ where $F(i)$ is the number of failure transitions possible out

of state i . The selection of the repair transition is similar to the case of simple failure biasing.

Under balanced failure biasing, many unlikely paths to system failure may be generated, but enough of the most likely such paths are generated so as to guarantee good estimates. Further analysis that characterizes when these and more general failure biasing schemes are efficient is given in Nakayama [19].

2.6 Forcing

At each transition step of the simulation, we essentially need to make two decisions.

- When to have the next transition?
- What should be the next state?

The technique of failure biasing influences the second decision, i.e., we force the system towards additional faults. The technique of “forcing” [18] influences the first decision. It increases the probability of the system having another transition before the end of the mission time.

This technique is widely used to estimate transient measures such as unreliability, mean time to failure etc. Note that this heuristic is essentially independent of failure biasing and each of them can be applied in isolation. The likelihood ratios due to of them are calculated separately and multiplied together to give the cumulative likelihood ratio.

This technique is applied only when the mission time is sufficiently small that the probability of another transition occurring before the end of the mission time is

very small. When failed components are present, the total transition rate is usually sufficiently large that this heuristic need not be applied.

2.7 Other Importance Sampling Results

The above mentioned techniques of failure biasing and forcing are the most widely used importance sampling heuristics. These are also important to us as they are heavily used in estimating the reliability of systems. There are a few other results related to importance sampling and we will look at some of them here.

Regenerative Simulation

Some times, we are interested in the steady state performance measures such as the steady-state unavailability, i.e., the long run fraction of the time that the system is in a failure state. If the system is regenerative then the regenerative method can be used to estimate steady state performance measures.

Let X_s be the process at time s . We assume there is a particular state, call it 0, such that the process returns to state 0 infinitely often and that, upon hitting state 0, the stochastic evolution of the system is independent of the past and has the same distribution as if the process were started in state 0.

Let β_i denote the time of the i -th regeneration ($\beta_0 = 0$) and $X_0 = 0$. Let $\alpha_i = \beta_i - \beta_{i-1}$ denote the length of the i -th regenerative cycle. If $E[\alpha_i] < \infty$ then under certain regularity conditions, $X_s \Rightarrow X$ as $s \rightarrow \infty$ (where \Rightarrow denotes convergence in distribution and X has the steady state distribution). Let h be a function on the

state space and define $Y_i = \int_{\beta_{i-1}}^{\beta_i} h(X_s) ds$ Then $(Y_i, \alpha_i), i \geq 1$ are i.i.d. and

$$E[h(X)] = \frac{E[Y_i]}{E[\alpha_i]}$$

The above equation forms the basis of the regenerative method; simulate N cycles and estimate $E[h(X)]$ by $\bar{Y}_N/\bar{\alpha}_N$ where \bar{Y}_N and $\bar{\alpha}_N$ are the averages of the Y_i -s and α_i -s respectively.

For Large Time Horizons

When the mission time is fixed, balanced failure biasing and forcing produce bounded relative error estimates of the unreliability $U(t)$. For very large time horizons, the empirical effectiveness of forcing decreases and a somewhat different approach has to be taken.

This problem is analyzed in Shahabuddin [26] and the regenerative structure of the system is exploited to estimate tight upper and lower bounds on $U(t)$. A different approach to estimating $U(t)$ is presented in Carrasco [5]. Instead of estimating $U(t)$, its Laplace transform $\hat{U}(s)$ is estimated at a number of values of s . The regenerative structure is again exploited by deriving a renewal equation for $\hat{U}(s)$ in terms of quantities defined over a single cycle that can easily be estimated. Numerical inversion of the estimated Laplace transform is then used to recover estimates of $U(t)$.

Failure Distance Biasing

Carrasco [4] considers another failure biasing approach, termed “failure distance biasing” which attempts to improve on the efficiency of balanced failure biasing by

giving more weight to sample paths that are closer to the set of system failure states F .

In this approach, failure transitions are grouped into classes based on their estimated distance to F and more weight is given to the classes corresponding to shorter distances. Once a class is chosen, the individual transitions can be chosen either proportional to their original rates (unbounded relative error may occur in unbalanced systems) or they can be equalized (as in balanced failure biasing, then the resulting estimate has bounded relative error).

In practice, the success of this approach depends on the ability to correctly and efficiently assign the distances. The class of systems for which this can be done is unclear. In some cases, significant improvements over balanced failure biasing have been obtained for systems with a large number of component types.

2.8 A Sample Application

We will now look at a practical application of the importance sampling for evaluating the reliability of a complex system. Boyd and Bavuso [3] talk about the modeling of a highly reliable fault-tolerant guidance, navigation and control system for long duration spacecraft. It is of considerable interest to us, as it is the type of system that we aim to model using the simulator.

System Model

The system consists of a 3-dimensional hypercube configured as two fault-tolerant 2-dimensional modules, each with a spare processing node. This spare could be a

hot or cold spare. Each processing node communicates with the other processing nodes in the system through four ports. For the system to be operational all eight processing nodes must be operational and must all be able to communicate with each other. Therefore, the system will be considered failed if any processing node fails and a spare node is unable to take over or if any two nodes in the hypercube are unable to communicate with each other.

The mission time of the system was assumed to be 10 years. It was clear from preliminary studies that the system with a traditional constant failure rate model will not meet the high reliability requirements. More recently acquired empirical data provided evidence that decreasing fault rates are common in spacecraft applications. Hence, they were interested in the effect of assuming that the components having a Weibull decreasing fault rate instead of the usual constant failure rate that is characteristic of the time-homogeneous Markov models. They were also interested in assessing the improvement in system reliability, if any, that can be achieved by using a cold spare processor in the processing nodes instead of a hot spare.

Simulation

The inclusion of decreasing failure rates with cold spares requires the use of a non-Markovian reliability model which is substantially more difficult to solve than the Markovian model that assumes a constant failure rate. Since the analytical solution of the system was extremely tough it was planned to use simulation.

Since the system failure events are extremely rare, a large number of trials will be needed to evaluate the reliability. To speed up the reliability evaluation, impor-

tance sampling techniques such as forced transitions and simple failure biasing were implemented.

Results

Results using constant failure rates indicated that the proposed architecture would be inadequate, with the probability of system failure exceeding 60%. Initial attempts to evaluate the model with HARP [6] (which uses analytical solution techniques) were not successful due to the large size of the model.

The simulator was able to handle the system model well. The effect of assuming Weibull decreasing fault rate clearly resulted in decreasing system unreliability. There was a difference of about three orders of magnitude in the system unreliability from 0.631 ± 0.013 when all the components have constant fault rate to about $0.777 \times 10^{-3} \pm 0.41 \times 10^{-3}$ when all the components have Weibull distribution.

From their experience in implementing the importance sampling for a complex model they have the following recommendations: Simulation techniques such as importance sampling are able to evaluate models that are beyond the reach of analytical techniques both in terms of memory and execution time. If only ballpark estimates are desired, simulation may be able to produce the required results relatively quick. However, if the accuracy of the evaluation is important, the execution time required by the simulation increases rapidly. Therefore, they advocate using the analytical methods if this is feasible and use simulation to evaluate the more complex cases.

2.9 Other Variance Reduction Techniques

Importance sampling is not the only variance reduction method used in simulation. There are quite a few other techniques that could be used for variance reduction. We will here try to present a sampling of such schemes. For an excellent discussion of the topic see [7] and [22].

Common Random Numbers

Common random numbers method is normally used when estimating the difference between the expected performance measures of two or more systems. It is perhaps the most widely used variance reduction method in practice. Suppose we want to estimate $\eta_1 - \eta_2$, where η_1 and η_2 are two unknown quantities, estimated by X_1 and X_2 respectively. Let $Z = X_1 - X_2$ and suppose that $E[Z] = \eta_1 - \eta_2$. The variance of Z is then

$$Var[Z] = Var[X_1] + Var[X_2] - 2Cov[X_1, X_2]$$

If X_1 and X_2 are generated independently, the covariance term disappears. But if we manage to introduce a positive covariance between X_1 and X_2 without changing their individual distributions, then the variance of Z will be reduced. The standard way of inducing such a covariance is to use the same underlying random numbers to drive the simulation for both X_1 and X_2 and to make sure that these random numbers are used at exactly the same place for both systems

Antithetic Variates

The idea of antithetic variates resembles that of the common random numbers mentioned above. Now, we want to estimate a single mathematical expectation η , using a pair of unbiased estimators (X_1, X_2) . The unbiased estimator of η will be the average $X = (X_1 + X_2)/2$. Whose variance is given by

$$Var[X] = \frac{Var[X_1] + Var[X_2]}{4} + \frac{Cov[X_1, X_2]}{2}$$

If the $Cov[X_1, X_2] < 0$, then X has a smaller variance. A standard way of inducing the negative correlation is to use a sequence of underlying iid uniforms $\omega \equiv \{U_k, k \geq 1\}$ to drive the simulation for computing X_1 and use the antithetic sequence $1 - \omega \equiv \{1 - U_k, k \geq 1\}$ to drive the simulation when computing X_2 . The rationale is that disastrous events in the first simulation should be compensated by the “antithetic” lucky events in the second one, thus reducing the variance of the average.

Conditional Monte Carlo

The general idea of Conditional Monte Carlo (CMC) is to replace the estimator $X = h(\omega)$ by its conditional expectation given another random variable Z . Roughly, if Z contains much less information than X , then the CMC estimator $X_{cm} \equiv E[X|Z]$ should have much less variability than X . More specifically, one has $E[X_{cm}] = E[X]$ and $Var[X_{cm}] = Var[X] - E[Var[X|Z]]$, so the variance can only decrease.

Indirect Estimation

Suppose that the mean μ of interest can be expressed as a known function of some other quantity η , say $\mu = f(\eta)$. Then it may be more efficient to estimate η instead of μ , then apply f to the estimator of η . This is called indirect estimation.

Assuming that we want to estimate the steady-state average queue length L_q . For the standard estimator, we simulate the system for a long time horizon and take the sample time-average. An alternative indirect estimator is based on Little's law $L_q = \lambda w_q$ where λ is the arrival rate and w_q is the average waiting time in the queue.

CHAPTER 3

IMPLEMENTATION OF THE TEST BED

As mentioned earlier, we want to build a simulator test bed where the user can configure a system of his/her choice and find out its reliability. This simulator should be modular so that it will be easy in future to modify or add new building blocks of the system. In this chapter we will take a close look at the simulation model, the overall design of the simulator and some interesting issues in building such a distributed test bed.

3.1 Simulation Model

Before we set out to design and build the simulator, we need to look at the system model that we are trying to simulate. A distributed real-time system can often be structured as a collection of computing nodes that are connected by a communication network. This can be modeled as illustrated in Figure 1.

From this model, we can identify some of the key elements of the system. They are

- The computing nodes

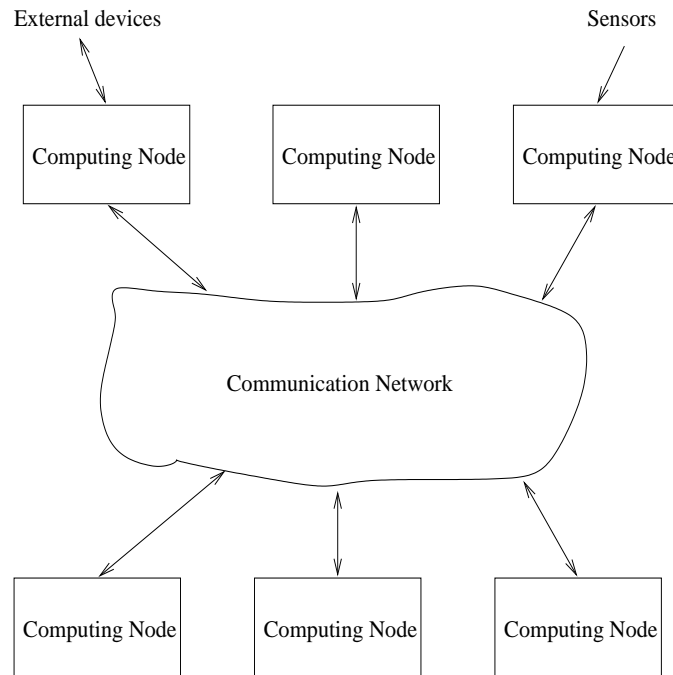


Figure 1: A typical distributed real-time system

- The tasks that run on the nodes
- The algorithms to be used for allocation/scheduling
- The messages that are exchanged by the nodes as part of execution or maintenance
- The network interconnect

We will take a look at each of these entities and see how they can be effectively modeled.

3.1.1 Computing Nodes

Each of the computing nodes in the system can be modeled as a single entity that is self sufficient in itself. It has private memory, it's own scheduling algorithm, checkpointing scheme, etc. Each of these can be individually changed without affecting the others. Once we have such a collection of nodes, we need to have some mechanism by which they work together. One of the most common solutions is to have a master-slave relationship among the nodes.

There is a single master node in the system whose only responsibility is to make sure that the rest of the nodes work together to execute the tasks even in the presence of faults. The responsibilities of the master can be enumerated as follows

- Task allocation. The jobs arrive from the outside world at the master which has to send it out to the appropriate node for execution.
- Fault Recovery. When there is a fault on one of the nodes, the recovery policy has to be consulted to choose the recovery action. The master then has to implement this action to the best of its ability.

To make this scheme work, the slave nodes have to periodically record their state in a checkpoint and send out 'alive' messages that the master can monitor. They also have to monitor the working of the master. In case the master malfunctions, a new master node has to be elected from the currently active slaves. Of course, this means that any slave node will have to be able to take on the master functionality smoothly.

3.1.2 Tasks and Messages

Many real-time applications and systems are highly structured, much more so than the general purpose systems. The set of tasks and their properties are generally known beforehand at least approximately. These tasks can be classified by nature of their recurrence. They are

- *Periodic Tasks:* Many tasks in real-time systems are executed periodically. For example, most of the monitoring tasks are periodic and these are executed at regular intervals. The periodicity of these tasks are known to the designer and so can be prescheduled.
- *Aperiodic Tasks:* These are tasks that occur only occasionally. The arrival pattern cannot be predicted.
- *Sporadic Tasks:* Aperiodic tasks with a bounded inter arrival time are called as sporadic tasks.

In a typical real-time system, there will be a mixture of these task types, the majority of them being periodic. In the current version of the simulator, we have allowed periodic tasks which can be described by attributes such as period, deadline, phase, redundancy, priority, messages to be sent and received during its execution etc. User could also insert a task during the simulation run.

The computing nodes exchange messages among themselves either as a result of the tasks that they are executing or as a part of the maintenance functions. Depending upon the functionality, they can be classified as data messages (which are sent as part

of the processing for the tasks) and control messages (which are used to monitor the state of the system etc).

3.1.3 Scheduling and Allocation Algorithms

Given a list of tasks that are defined as mentioned above, it is the task of the allocation/scheduling algorithms to make sure that the task deadlines are met. These algorithms can be characterized by the following parameters.

- Hard real-time (needs tough performance guarantees) versus soft real-time (can live with a best efforts approach)
- Preemptive (allows tasks to be suspended temporarily when a higher priority task arrives) versus non-preemptive scheduling (runs each task to completion)
- Dynamic (scheduling decisions are made during execution) versus static (scheduling decisions are made in advance)
- Centralized (one node collecting information and making the decisions) versus decentralized

Given a set of tasks, task precedence constraints, resource requirements, task characteristics and deadlines, the real-time computer has to come up with a feasible allocation/schedule. A task assignment/schedule is said to be feasible if all the tasks start after their release times and complete before their deadlines.

Allocation Algorithms

On systems with more than two processors, the task assignment/scheduling problem is NP-complete [16]. So, we use the following heuristic : First the master assigns the tasks to the processors and then runs the uniprocessor scheduling algorithm for each of the slave nodes to see if the allocation was feasible. If one or more of the schedules are infeasible, we must either return to the allocation step and change the allocation or declare that a schedule cannot be found.

Different allocation algorithms are used. They vary from a pure round-robin allocation to schemes which aim to keep the utilization of the individual processors below a limit. This limit depends on the characteristics of the tasks and the uniprocessor scheduling algorithm being run on the individual processors.

Uniprocessor Scheduling Algorithms

These algorithms typically assign priorities to the tasks and execute the task with the highest priority. The relative priorities of the tasks are a function of the nature of the tasks themselves and the current state of the system. The following two algorithms are implemented in the simulator.

Rate-Monotonic is an optimal and popular static-priority algorithm. It is used to schedule periodic, preemptible tasks whose deadlines equal the task period. The basic idea of the rate monotonic algorithm is to assign different and fixed priorities to tasks with different execution rates, highest priority being assigned to the highest frequency tasks. At any time, the low-level scheduler simply chooses to execute the

highest priority task. A task set of n tasks is schedulable under RM if its total processor utilization is no greater than $n(2^{1/n} - 1)$.

EDF is an optimal dynamic-priority scheduling algorithm. It is used to schedule preemptible tasks. The task with the earliest deadline has the highest priority. If a task set is not schedulable on a single processor by EDF, there is no other uniprocessor scheduling algorithm that can successfully schedule that task. If all the tasks are periodic and have relative deadlines equal to their periods, the test for task-set schedulability is simple: If the total utilization of the task set is no greater than 1, the task set can be feasibly scheduled on a single processor by the EDF algorithm.

3.1.4 Network

Communication in real-time distributed systems is different from communication in other distributed systems. While high performance is always welcome, predictability and determinism are the real keys to success.

Achieving predictability in a distributed system means that communications between processors must also be predictable. LAN protocols that are inherently stochastic, such as Ethernet are unacceptable because they do not provide a known upper bound on transmission time. As a contrast, consider a token ring LAN. Whenever a processor has a packet to send, it waits for the circulating token to pass by, then it captures the token and sends its packet. Assuming that each of the k machines on the ring is allowed to send at most one n byte packet per token capture, it can be guaranteed that an urgent packet arriving anywhere in the system can always be

transmitted within kn byte time plus overhead. This is the kind of upper bound that a real-time distributed system needs.

A faithful simulation of the interconnect network is crucial in getting a realistic cost of the 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 complete system.

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

W is the time spent waiting in the queue before starting transmission. The factors affecting it are the medium access control protocol used and the length of the queue ahead of the current packet.

S is the service time – the time taken to transmit all the bits in the packet (proportional to the size of the packet). It depends on the data rate of the channel.

T is the propagation delay - the time taken for a single bit to travel to the destination.

It is a characteristic of the channel used and is proportional to the distance between the source and the destination.

As we can see here, simulation of different topologies, channels and the medium access control protocols give rise to the different delays experienced by a packet.

In our simulator, the computing nodes send their messages out to a network which then inserts a certain delay (characteristic of the network type) and then forwards the message to the destination.

3.1.5 Fault Injection

In the current model, only the computing nodes can get the faults. The user can specify the faults in a variety of ways including: the Poisson rates for the fault arrival and repair, a table of fault arrival events and repair etc. It can be extended to include other possibilities.

There is a fault generator (we will see later on its implementation) which computes the fault arrival/repair events and send them out to the computing nodes.

When the fault arrival event comes up, the nodes simulate the fault by stopping the current task and also clearing further events from the event queue. Until the fault is cleared and some recovery action has been performed on the node, it will not respond further.

3.2 Distributed Simulation Issues

The test bed is a event-driven simulator and hence the simulation proceeds by executing events from an event queue ordered by the time of their occurrence. During the execution of an event, the state of the system changes and new events might also be created. These events are then inserted into the event queue to be executed later. In a distributed simulation, there are multiple such event queues in the system. The key to success is to let the simulation proceed as efficiently as possible without violating the causality constraints [28].

Each of the nodes/network have a separate event queue. We now have to decide on how to let the simulation clock proceed. One possibility (and which we have followed

in our simulator) is to have a central clock that collects the next event time from all the event queues, computes the next event time for the entire system and then broadcasts this time to all. This ensures correctness so that the events proceed in an orderly fashion. The nodes execute the events that are supposed to occur at that instant. As part of this process, more events get generated and get inserted into the respective event queues. The clock collects the next event times from all the nodes and the process repeats.

Other possibilities include some kind of optimistic execution in which the simulation time proceeds in parallel until there is an interaction between nodes that might violate causality. It is more complex to implement but definitely improves the performance of the system as a whole.

3.3 RAMP Algorithm

When a fault has been detected on a slave node, there are a few possible recovery actions that can be done. Some of the most commonly used ones are

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 tasks on a spare node.

Disconnect Use the latest checkpoint from the faulty node to distribute the tasks running on that node to the other non-faulty nodes in the system.

Each of these actions can have different overheads in terms of the time needed to complete them. Their success also depends on the current state of the system, the workload, fault characteristics etc. Given all this information, the master node has to decide which action to take. The policy that the master uses could vary from a fixed action or something that gives certain weightage for all these parameters and then decides on a action.

RAMP is an example of a resource management algorithm that can be used. It is intended to suggest the most suitable recovery action that will minimize the probability of the system losing a critical deadline over the remaining mission time. For this, it takes into consideration the fault characteristics, workload, check-pointing interval etc.

To make the computation time of the algorithm feasible, a method called the Reduced State Space Markov Decision Process (RAMP) [31] was developed. This method reduces the system state space to a manageable level without significantly compromising precision. A dynamic programming technique [20] 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 the algorithm consists of the set of actions to take for all possible system configurations and workload characteristics at a given point of time (with a desired resolution) in the mission.

Whenever a fault is detected and a recovery action needs to be chosen, the RAMP algorithm is consulted. Complete information on the current system state such as the

number of nodes that are alive and faulty, the remaining mission time etc are passed to the algorithm. The RAMP algorithm then suggests an appropriate recovery action after looking at these values and the precomputed alternatives.

3.4 Simulator Implementation

The design of the simulator is closely tied to the kind of systems that are to be modeled. We have seen earlier about the simulation model. The simulator models this system by having a set of processes that communicate with each other by means of a portable message passing library. There is a process to model each of the computing nodes, the network interconnect, central clock and the console. We will now take a look at each of these.

3.4.1 Platform

The simulator is intended to be run on multiple machines in a transparent manner. That way, we could add more machines for the simulation when we need more computational capacity. For this reason, PVM is chosen as platform on which to build the simulator.

PVM [8] is a portable message passing library that can run on multiple physical machines in a manner that is transparent to the applications. It uses the native message passing mechanism of the underlying machines (example UNIX sockets) to provide an abstract view of a single virtual machine.

Machines can be added to the PVM at will, thereby making it easy to increase the computing capacity. The processes that run on top of PVM can exchange messages

through well defined mechanisms and semantics. PVM is available on a wide variety of systems and the virtual machines can be made of multiple, heterogeneous machines. The simulator code is developed in C++.

This chosen platform has the following advantages

- Using a network of workstations for the simulator makes available a scalable computing platform. This will be helpful as the simulated system becomes more complicated and requires more computing power.
- Implementing the various components using the PVM processes facilitates distributing the load across machines evenly.
- PVM, implemented on a flavor of UNIX provides a portable message passing interface. So the simulator can easily be adapted to a wide variety of machines.
- C++ is a powerful object-oriented language which can be easily used to model the complex systems that are being simulated.
- Since C++ is a popular language, it will be easy for users to develop their own algorithms and integrate them with the simulator.

3.4.2 Implementation Model

There are two parts to the simulator implementation. How the actual system is simulated and how it is controlled from the users perspective. This can be modeled as shown in Figure 2.

The console acts as the controlling interface to the rest of the simulator. We can enumerate its functionality as follows

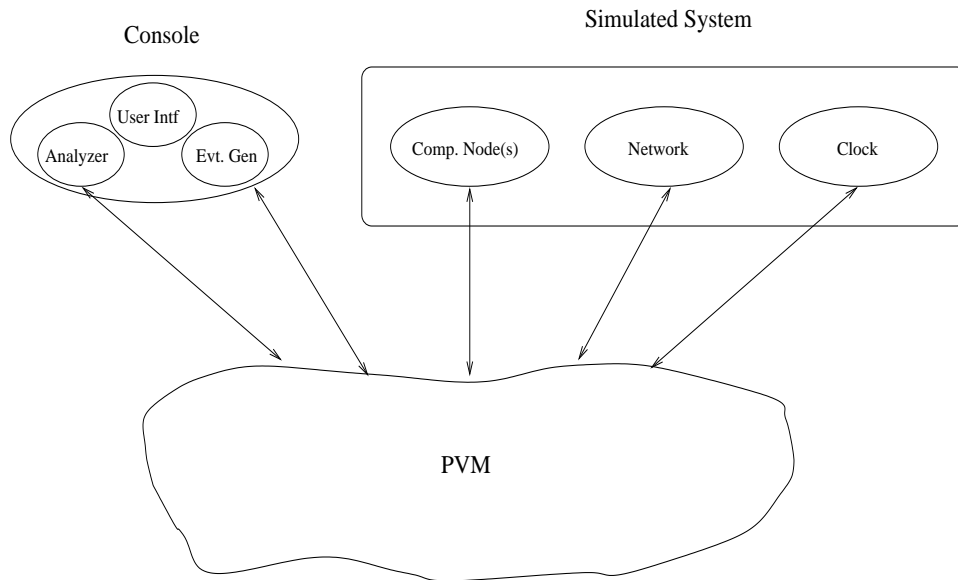


Figure 2: The Implementation Model

- Provides the GUI to help the user specify the system
- Provides the controls for the simulator
- Collects information about the task description, fault arrival patterns etc.
- Generates the fault arrival/repair events and send them out to the nodes
- Spawns the required processes
- Collects the information about the current state of the system
- Provides the status to the user via the GUI.

3.4.3 Other Simulator Entities

Virtual Nodes

Each virtual node represents a computing element in which the tasks run and exchange messages. The nodes run in a master-slave configuration. One of the nodes acts as the Master and does the control tasks such as task allocation, fault detection, fault recovery etc. Usually, the node with the highest id acts as the Master. If this node goes faulty, then some other node takes over. The slave nodes schedule and execute tasks which are allocated by the master.

Facilities are provided for plugging in the various algorithms that the nodes use such as the task allocation, scheduling etc. In the current version of the simulator, some of these algorithms are already in place and the future ones can be incorporated with minimum effort.

Virtual Network

The nodes can be connected via broadcast or point-to-point links. A variety of protocols such as FDDI and IEEE 802.5 Token Ring can be used on the broadcast links. In the case of point-to-point links, the nodes maintain routing tables to forward messages to the appropriate destination. A single process simulates the network connect in the system. It inserts the appropriate delay into the message passing depending on the protocol that is simulated in the network. We will now take a look at some special considerations for the simulation of individual networks.

Token Ring

Efficiency is an important issue in simulating the network protocols. In a token passing protocol, the token keeps on circulating even where is no message in the network. Each of these token passing events have to be simulated and hence it slows down the system considerably. The alternative is to stop the token passing when there are no messages in the system and restart it only when a new message comes into the system. To implement this, we record the position and the time when the token was last seen and regenerate the token at the appropriate place when a new message arrives.

The parameters of the network such as the operating speed, the number of nodes, token length, node latency, the token holding time etc, can be varied to reflect the network under consideration.

FDDI

FDDI medium access control [15] is similar to the token Ring in that it also depends on token passing in a ring. However it follows a timed token protocol which allows each node to reserve a portion of the available bandwidth to send Synchronous data (which is the real-time traffic) at periodic intervals. Each node can calculate the minimum amount of data that it needs to put out every P_i time units and send this information to the network. The network can calculate the fraction of the bandwidth needed for each node and assigns them to the nodes. This amount of bandwidth is guaranteed for each node over the requested period. Any unused bandwidth can be used by Asynchronous data which is the non-real time traffic.

For a more detailed discussion on the mechanism of bandwidth allocation, and the cycle time properties of the FDDI algorithm, see [16]. The efficiency considerations that occurred for the token ring also applies in this case and we follow the same approach.

Central Clock

The virtual nodes and network have their own local version of virtual time and run independently of others except when there is a need for interaction. They must satisfy causality constraints [17] to ensure correctness of the simulation.

The RAPIDS simulator uses a variation of the Breathing Time Buckets technique [28]. Each process in the simulator has a notion of the Local Event Horizon, which is the time it can proceed to, without violating causality. The central clock maintains the Global Event Horizon which is always the minimum of the Local Event Horizons, and broadcasts this as the current time of the entire system.

3.4.4 Fault Injection

As we have seen earlier in the implementation model, the console has an event generator which is in charge of generating the fault arrival and repair events. The fault injector reads the fault generation information in one of two possible formats.

- Poisson fault arrival, repair rates
- Table of fault arrival times and repair times

Looking at this information for each of the computing nodes, it generates the two types of events (fault arrival and repair). By not sending out the fault repair event

for a node, the permanent faults can be simulated. A predefined number of these events are generated and sent out to the concerned nodes. These nodes insert the events into their respective event queues and simulate the events at the appropriate time.

These events are periodically replenished whenever the number of events falls below a low-water mark. Since the nodes send their event information back to the console, a precise estimate can be made about the number of events still to be executed.

3.4.5 Fault Detection and Recovery

Fault on the Slave Nodes

The master node is responsible for detecting the failure of a slave, and invoking the appropriate recovery actions. Each fault-free slave node periodically records its state in a checkpoint. It also periodically sends an “I am alive” message to the master. The master node periodically examines these slave messages to decide if any of them are faulty. The period of the alive messages and the master invoking the maintenance functions are design parameters that can be specified.

The master node maintains the following information about the system to help in allocation and reassignment of tasks etc: The utilization of each node, whether the node is currently alive, whether the node is a spare, whether the node has sent an Alive message recently, the recommended recovery action for the node.

On detecting a fault, the master invokes a recovery algorithm to decide on the appropriate recovery action.

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. User can also specify the algorithm to be used to generate the recovery actions during the mission time. This can be one of the following: a fixed action, a heuristic or an optimal recovery policy algorithm similar to the RAMP algorithm.

Interfacing with the RAMP algorithm

The RAMP algorithm precomputes and outputs the set of actions to take for all possible system configurations and workload characteristics at a given point of time (with a desired resolution) in the mission. The selection of a particular action depends on the fault characteristics (transient and permanent fault rates, transient fault duration), the given workload, the checkpointing interval and the remaining system mission time.

Some of the parameters for the RAMP algorithm are needed at the beginning, to generate the tables. These are either taken from the user, or estimated by doing a pre-simulation on the user inputs. Others are run-time parameters indicating the current time in the mission, number of nodes that are currently up, the utilization of these nodes etc. These are calculated at runtime and passed onto the algorithm. The algorithm uses this information to select the appropriate recovery action and gives this information to the simulator to implement.

Fault on the Master Node

The master node also sends an "I am alive" message to the slaves and the slaves monitor this message. The non-receipt of this message indicates that the master has gone faulty, and a new master has to be elected.

The slaves follow a 'Bully' algorithm for the election in which the slave with the highest id becomes the new master. As soon as the failure is detected by a node, it sends an election message to the other nodes that have a higher id. The nodes getting the election message respond with an acknowledgement whereupon the original node(s) gives up and waits. If no node sends back an acknowledgement, then the node that initiated the election wins and becomes the new master. The node that sends out the acknowledgement has the responsibility of finding the new master and so it initiates another election (if it hasn't done so yet). Ultimately, the node with the highest id (that is currently alive) is found and becomes the new master.

CHAPTER 4

SIMULATION ANALYSIS

Real-time systems have to be carefully validated before they are put into operation. Specifically, we need to know the probability that the system will not fail over the mission time. Here we also need to be specific about what we mean by “failure of the system”. This chapter looks closely at this and tries to arrive at a framework to estimate it.

4.1 Reliability

When we have a mission-oriented system, we would like to know whether it will fail over its mission time. Reliability is the probability that the system will not fail over its mission time. Unreliability is the complementary probability (the probability that the system will fail over its mission time). To find this, we need to define what exactly we mean by ‘failure’ in our context. We will now try to define this in the context of a hard real-time system that we intend to model.

In a hard real time system we can have critical and non-critical tasks. Critical tasks need to be executed before their deadlines and if these deadlines are not met, catastrophes can occur. Non-critical tasks are not critical to the application. However,

they do deal with time-varying data and hence are useless if not completed within their deadline. The objective of a hard real time system is to meet all the critical deadlines and if possible, the non-critical deadlines too.

The Critical tasks are often executed at a higher frequency than is absolutely necessary. This constitutes time redundancy and ensures that one successful computation every n iterations of the critical task is sufficient to keep the system alive. The actual value of n will depend on the application, the nature of the task and its frequency.

We define the system failure state as the state where it failed to meet the required number of iterations out of the n iterations. Even if a single critical task does not meet its deadline, then it is deemed a failure of the system itself. For each critical task, we can have a moving window and check whether the system is able to meet the specified number of deadlines among the iterations.

In order to measure the unreliability, we can run the system for its mission time and see whether it fails. This experiment is repeated many times and the percentage of times the system failed gives the estimate of the unreliability of the system.

4.2 Normal Simulation for Estimating Reliability

A simulation study is usually undertaken to determine the reliability of the system. In cases where the value of the reliability is very close to 1, it is usually better to estimate the unreliability of the system which is $1 - reliability$. Lets call the unreliability of the system as θ which is the parameter to be estimated from a simulation

study. The system is simulated for the mission time to see if it fails. The output value X_i is a 1 or 0 depending on whether the system failed or not. The simulation is repeated to get more such data points. The mean of these data points \bar{X} provides an estimation of θ .

By the central limit theorem, a $100(1 - \alpha)\%$ confidence interval for θ is approximately $\bar{X} \pm z_{\alpha/2}S/\sqrt{n}$ where S is the observed standard deviation and $z_{\alpha/2}$ is defined by the equation

$$\alpha/2 = P(N(0, 1) \geq z_{\alpha/2})$$

$N(0, 1)$ denotes a normally distributed random variable with mean 0 and variance 1.

By definition,

$$X_i = \begin{cases} 1 & \text{with probability } \theta, \text{ and} \\ 0 & \text{with probability } 1 - \theta \end{cases}$$

and hence the variance of X_i is given by

$$Var(X_i) = \theta(1 - \theta)$$

Suppose we wish to estimate θ to within $\pm 10\%$ (about two significant digits of accuracy), i.e, we want the relative half-width of say, a 90% confidence interval for θ to be less than 0.1, that implies :

$$1.282\sqrt{\theta(1 - \theta)/n}/\bar{X} \leq 0.1$$

$$n \approx 1.282^2 * 100 * (1 - \theta)/\theta$$

From this, we can see that the required sample size is proportional to $1/\theta$. The smaller θ is, the larger the sample size must be. For example, if θ is of the order of 10^{-6} , a sample size of the order of 10^8 will be needed.

For the kind of systems that we are considering, the probability of failure is very low for the individual components and the unreliability of the complete system is very low. Therefore a very large number of samples is required to estimate this parameter to a reasonable level of accuracy.

CHAPTER 5

IMPLEMENTING IMPORTANCE SAMPLING

In the earlier chapters we have discussed the basics of importance sampling, how the simulator test bed is implemented and how the simulation outputs can be analyzed to estimate the reliability of the system.

This chapter talks about how all these are put together to actually implement the desired solution and how exactly we validate the implementation. It also takes a look at the parameters that affect the simulation output and provides some guidelines on how to choose them.

5.1 Outline

The implementation model of the system is as shown in Figure 3. To implement importance sampling in this model, we should not alter the simulated system, we only need to modify the generation of the fault arrival/repair events and the way the reports are analyzed. We intend to measure the unreliability of the system by repeating the simulations to get individual samples. The algorithm to be followed can be summarized as

- Initialize the weight of the simulation output to 1.

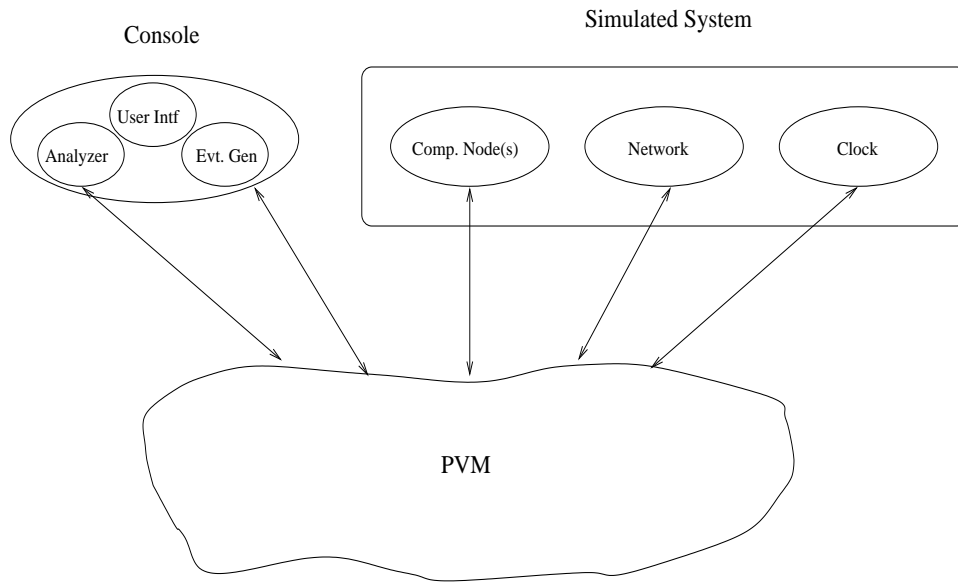


Figure 3: The Implementation Model

- Use the heuristics of “forcing” and “failure biasing” to generate the fault arrival/repair events.
- Send these events to the individual components and replenish them periodically.
- Monitor the reports from the simulated system.
- Update the simulation weight whenever a “change of measure” is performed.
- Output either the simulation weight, when the system fails before the end of the mission or zero, when the mission ends with the system still functioning. This forms a single sample of the simulation run.

The logical place to implement importance sampling is in the console. To be more precise, we can implement this in the event generator and the analyzer. The event generator has the following responsibilities

- Decide the time of the next system state transition. Implement “forcing” to accelerate the state changes.
- Decide whether the next transition is a fault arrival or repair. Implement “failure biasing” to push the system towards more component faults.
- Calculate the likelihood ratios associated with each “change of measure” and store this value along with the time it is supposed to happen.

The analyzer has the following responsibilities

- Receive the reports from the simulated system.
- If it corresponds to one of the above mentioned “change of measure”, update the current simulation weight.
- If the system fails within the mission time, set the simulation output to the current value of the likelihood ratio else set the simulation output to zero.

We will see later how each of these functions are implemented in the modules.

Theoretical Formulations

The equations governing system failure are constructed from two probability density functions [18]. Let

$f(t|t', k')$ \equiv Probability density that the system will make a state transition at

t given that it is at state k' at time t' ($t' \leq t$) and

$q(k|k') \equiv$ Probability that the system will enter state k , following a transition out of state k' .

Now, let $\psi_k^n(t)$ be the probability density that the system arrives in state k at time t after the n^{th} transition. Then, the probability density for arrival in state k is:

$$\psi_k(t) = \sum_{n=0}^{\infty} \psi_k^n(t).$$

A recursive relation for the states for which $n > 0$ follows immediately from the definitions of the probability densities:

$$\psi_k^n(t) = \sum_{k'} q(k|k') \int_0^t dt' f(t|t', k') \psi_{k'}^{n-1}(t')$$

Suppose we consider a modified or "biased" random walk where the probability density $f(t|t', k')$ and the discrete probabilities $q(k|k')$, both defined above, are replaced respectively by the modified distributions $\tilde{f}(t|t', k')$ and $\tilde{q}(k|k')$.

Similarly we can define $\tilde{\psi}_k^n(t)$ as the modified density that the system arrives at state k at time t after the n^{th} transition. We now require a relation between the earlier density and the new density function. To do this, we define a weight $w_k^n(t)$ such that

$$\psi_k^n(t) = w_k^n(t) \tilde{\psi}_k^n(t)$$

This weight $w_k^n(t)$ is the likelihood ratio associated with the current state. Substituting this in the earlier equations and simplifying we can get the recursive relation

$$w_k^n(t) = \frac{q(k|k') f(t|t', k')}{\tilde{q}(k|k') \tilde{f}(t|t', k')} w_{k'}^{n-1}(t')$$

The above is the key equation that we will need for the implementation. We can associate a weight with each random walk, which is initialized to one and then adjusted to correct for the bias at each sampling. The incremental likelihood ratio for each “change of measure” is given by

$$\frac{q(k|k')f(t|t', k')}{\tilde{q}(k|k')\tilde{f}(t|t', k')}$$

This has two parts, $q(k|k')/\tilde{q}(k|k')$ and $f(t|t', k')/\tilde{f}(t|t', k')$ each corresponding to the likelihood ratio due to the two heuristics of failure biasing and forcing respectively. Since these are independent, we can calculate the ratios due to each of them separately and then multiply them to give the likelihood ratio for the state transition.

The cumulative weight when the system enters a failed state is the sample value(X_i) for the run. If the system doesn't fail over the entire mission, the resultant sample value is 0.

5.2 Implementation

The major chunk of the work is implemented by the event (fault) generator. It generates the list of the fault arrival/repair events for all the computing nodes in the system. To do this, it maintains a set of variables for each node i in the system. The fault arrival rate is given by λ_i and the repair rate is given by μ_i .

λ is the total failure rate out of the current state and is equal to the sum of the failure rates of active nodes. μ is the total repair rate out of the current state and is equal to the sum of the repair rates of all the currently faulty nodes (but not

permanently faulty). Finally γ is the total transition rate out of the current state and is equal to the sum of λ and μ . As a node goes faulty or gets repaired the system state changes and these variables are updated.

5.2.1 Forced transitions

The fault generator module uses a random number generator to generate random numbers uniformly distributed between 0 and 1. To sample time intervals Δt between transitions, we can use the following equation

$$\int_{t'}^{t'+\Delta t} dt f(t|t', k') = e^{-\gamma_{k'} \Delta t}$$

A random number ε is sampled from a uniform distribution $0 < \varepsilon \leq 1$, and set equal to the cumulative distribution on the left. Inverting the equation then yields

$$\Delta t = -\frac{1}{\gamma_{k'}} \ln \varepsilon$$

where we have utilized the fact that ε and $1 - \varepsilon$ have the same probability densities.

This value Δt is added to the present time t' to give the next transition time t . Usually the failure rate of the nodes is very small when compared to the repair rate (for transient faults). Therefore γ is small when no failed components are present. In such a case, the transition rate is boosted by taking

$$\begin{aligned} \tilde{f}(t|t', k') &= \frac{f(t|t', k')}{1 - e^{-\gamma(T-t')}} \text{ for } t' \leq t \leq T, \\ &= 0 \text{ otherwise} \end{aligned}$$

where T is the mission time of the system.

This heuristic is applied only when γ is small, $\gamma(T - t') \ll 1$ and there is only a small chance that an additional transition will take place before the end of mission time T .

We then have to multiply the trial weight by

$$\frac{f(t|t', k')}{\tilde{f}(t|t', k')} = 1 - e^{-\gamma(T-t')}$$

This value is the partial likelihood ratio and is stored in a variable for use later.

5.2.2 Failure biasing

Once the next transition time is decided, we have to see whether it is going to be an additional node failure or a repair. Obviously if there are no failed nodes present, there cannot be a repair and hence the only possibility is a node failure. Similarly if all the nodes are faulty, repair is the only way to go. Barring these two extreme cases, we have to make some decision on the type of transition.

We would like to bias the transitions in such a way as to cause more failures and thus push the system towards system failure quicker than in the normal case. To recapitulate, system failure is defined as the state where the system is not able to meet all its critical task deadlines.

As more and more nodes go faulty, the load gets redistributed among the working nodes thereby increasing their loads. As the loads keep building there comes a point when the scheduler cannot meet all the task deadlines and hence the system will start missing some of the deadlines. If this condition persists (i.e., the faulty nodes remain down long enough) critical task deadlines will be missed and the system will

fail. Also, when the tasks are being moved around, there is an overhead involved in terms of time and deadlines could be missed even though enough computing capacity is available somewhere in the system.

Suppose for a particular system state k' we divide all possible transitions $k' \rightarrow k$ into two classes, the transition $k \in F$ corresponds to an additional component failure, while $k \in R$ corresponds to an additional component repair. Hence for non absorbing states

$$\sum_{k \in F} q(k|k') + \sum_{k \in R} q(k|k') = 1$$

We choose a fraction called the failure bias which indicates the percentage of the transitions that result in an additional fault (This affects the dynamics of the sample output and we will look at it in more detail later). Let us assume that we choose a value ϕ , then,

$$\sum_{k \in F} \tilde{q}(k|k') = \phi$$

and hence,

$$\sum_{k \in R} \tilde{q}(k|k') = 1 - \phi$$

We generate a random number ε' and compare it with ϕ . If it is smaller, then the next transition will be an additional fault. Otherwise, it will be the repair.

We choose to implement one of the variants of the failure biasing called “balanced failure biasing” as we have seen earlier. To decide on the exact node getting the fault, we maintain an ordered list of all the eligible nodes (lets say n) and generate a random number i uniformly distributed between 0 and $n - 1$. This i is the index of

the node getting the fault. The unbiasing fraction is then given by

$$\frac{q(k|k')}{\tilde{q}(k|k')} = \frac{n\lambda_i}{\phi\gamma}$$

If it is a repair, we decide the repaired node by looking at the repair rate of the individual faulty nodes (excluding those permanently faulty). For this we order the eligible nodes and determine the particular node i by using the following formula

$$\sum_{i'=1}^i \mu_{i'} < \frac{\mu(\varepsilon' - \phi)}{1 - \phi} \leq \sum_{i'=1}^{i+1} \mu_{i'}$$

The unbiasing fraction is then given by

$$\frac{q(k|k')}{\tilde{q}(k|k')} = (1 - \phi)^{-1} \frac{\mu}{\gamma}$$

5.2.3 Analysis

As we have seen in the last two subsections, we can implement the two techniques of forcing and failure biasing. In each case we calculate the unbiasing fraction also. The product of these two gives the likelihood ratio for that transition.

This value is associated with the transition event and is stored in a list. The overall likelihood ratio for the simulation run is initialized to one. As the event actually happens in the system, the fractional value is taken out of the list and multiplied with the running product to give the current value of the likelihood ratio.

The nodes report all the events happening in the system to the console. If the system is not able to meet its critical task deadlines, this constitutes system failure

and the current value of the likelihood ratio is the sample output. If the system does not fail over the mission time, then the sample value is a zero.

The simulator maintains a file to hold the sample outputs and another to store the seed for the random generator. Since the random generator is a crucial component of the simulator, its seed has to be maintained over the sample runs to make sure the numbers are uniformly distributed and also the simulation can restart from the last paused point.

5.3 Expected Behaviour of Importance Sampling

Before we carry out the validation of the implementation, it is imperative for us to understand the expected behaviour of the importance sampling scheme.

Importance Sampling is a heuristic, designed to speed up the occurrence of rare events. We wonder whether it can perform uniformly well in all regions of unreliability values. Intuitively we know that there should be some range of the unreliability values beyond which the usage of the normal simulation will be better than the importance sampling.

We will use the ‘Relative Error’ (RE) defined as follows, as the performance measure of the estimation scheme (normal simulation and importance sampling).

$$RE = \frac{z_{\alpha/2}S}{\sqrt{n}\theta}$$

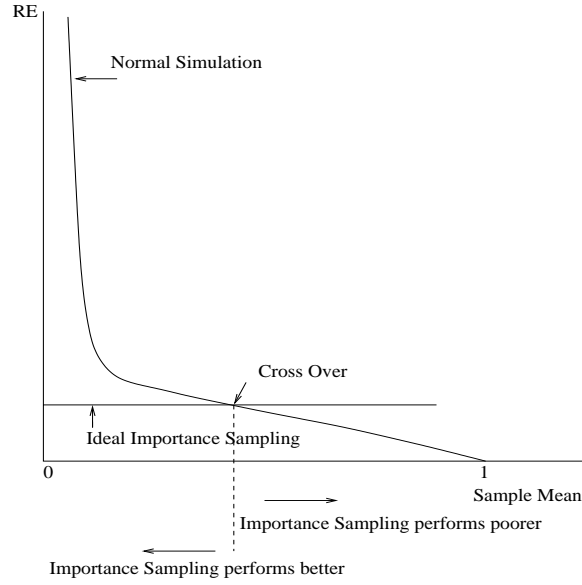


Figure 4: RE of Normal Simulation and Importance Sampling

For the case of the normal simulation, this becomes

$$RE = \frac{z_{\alpha/2} \sqrt{\theta(1-\theta)}}{\sqrt{n\theta}}$$

for a fixed number of samples, as the failure event becomes rarer (i.e., $\theta \rightarrow 0$) the $RE \approx z_{\alpha/2} / \sqrt{n\theta}$ becomes unbounded. Therefore, to obtain precise estimates, we need very large n .

For importance sampling, Shahabuddin [25] notes that the elements of the modified transition matrix should be independent of the ‘rarity’ (a parameter that he defines to reflect the highly reliable nature of the components) of the components. This is sort of an ‘ideal’ importance sampling scheme which will always lead to the

bounded RE property. He also proves in this case, the RE will have a form

$$RE = \frac{z_{\alpha/2} \sqrt{a_2 + o(1)}}{\sqrt{n} (a_0 + o(1))}$$

where a_2 and a_0 are positive constants depending on the implementation.

For a fixed number of samples, we can represent the behaviour of RE as shown in Figure 4, for the cases of normal simulation and importance sampling. This ‘ideal’ importance sampling scheme will be able to estimate the sample mean with a small, fixed number of samples. We also observe that there is a certain region of reliability values beyond which the normal simulation will be better than the importance sampling.

In practice, techniques such as ‘Balanced Failure Biasing’ approximate the behaviour of this ‘ideal’ case. They achieve variance reduction (and hence the RE) by pushing the system towards more faults and hence reducing the reliability of the underlying system. They then unbiased the sample output to get the correct value of the sample mean.

The RE offered by such heuristics may not exactly be a constant but could be a complicated function of the bias value and the type of the system under consideration. However as the reliability of the system keeps on decreasing, the bias value has to be kept arbitrarily low in order to maintain the RE close to that of the normal simulation. Hence beyond a point, the importance sampling is not very useful. It is important for us to experimentally determine this range of values so that the user can switch to the normal simulation instead of the importance sampling scheme.

Recapitulating the basic idea of the importance sampling scheme, the estimate of the sample mean is given by $E_{p'}[1_{\{X \in A\}}L(X)]$. Note that since we want to make the sample variance very low, we need to satisfy two things:

- More number of samples have to contribute to the result (this is achieved by pushing the system towards more faults and thereby increasing the chance of it failing and contributing a non-zero sample).
- The ‘most likely’ paths to failure have to be favored more since the likelihood ratio has to be well behaved (the proportion of the failure paths has to be proportional to the relative importance of these failure paths in the system under consideration)

In the case of the failure biasing heuristic, there is only one parameter with which to control both of these. If we increase the failure bias, more number of sample runs will result in system failure and hence it is good as far as the first condition is concerned. However failure bias also affects path that a sample run might take.

In the systems that we are interested in simulating, the most obvious failure path is the one where the system is unable to meet its critical task deadlines. This obviously depends on a variety of factors such as the number of computing nodes available, average task load on the computing nodes, the recovery overheads etc. Consider a rather crude example: let a system be composed of x computing nodes and to meet its critical task deadlines it needs atleast y of them. Increasing the failure bias has the effect of pushing the system towards complete breakdown (in other words, more of the nodes are pushed towards failure). In many cases this will be an overkill and

the likelihood ratio associated with these sample runs will be very low. Thus most of the sample runs will have a likelihood ratio that is very less and a few of them will have very large values. This effect causes the RE to blow up in cases where the bias value is very high.

Because of these conflicting effects associated with the bias value, each system might have a optimal bias value that pushes the system towards the failure path most of the time and still does not make it an overkill. Since it is not practical for us to locate this optimal value for each configuration, it is enough if we are able to guess a bias value that gives ‘good’ results over a range of systems.

5.4 Validation of the Model

To validate our implementation of importance sampling, we need to compare the reliability estimates with the ones generated using normal simulation (without any bias).

Choosing the Configurations

As we have seen earlier, normal simulation of the complex reliability models take a long time and need extremely large number of sample points to get reasonable confidence intervals. This is exacerbated by the fact that each of these simulation samples take a long time.

The simulator is primarily intended to model very complex system models and algorithms and hence the number of events to be simulated is very large. Because of this (and other factors such as the efficiency of the distributed simulation) each

simulation run takes on the order of minutes for mission times on the order of a couple of thousand units.

Because of these reasons, we need to be very careful in choosing the configurations that we use to validate the implementation. Some of the important considerations are

- We need to verify that the implementation works for a variety of configurations that can be modeled using the simulator
- We need to make sure that the accuracy of the results holds for configurations with various ranges of reliability values

Since we only aim to get ballpark estimates for comparing with the importance sampling scheme, we propose to repeat the simulations until we get the half width of the 90% confidence intervals to be around 30% of the sample mean.

We choose 4 different system configurations (with increasing reliability values) and estimate their unreliability without applying any bias. These configurations are as follows:

Conf1: Number of Nodes: 6

Network interconnect: Token Ring (4 Mbps)

Transient failure rates: $1.38 \times 10^{-3}(5/hr)$

Permanent failure rates: $2.77 \times 10^{-5}(1/hr)$

Mission time: 1000 units

Average load on the nodes: 0.4

Conf2: Number of Nodes: 7

Network interconnect: Fddi

Transient failure rates: $5.55 \times 10^{-4}(2/hr)$

Permanent failure rates: $5.55 \times 10^{-5}(0.2/hr)$

Mission time: 1500 units

Average load on the nodes: 0.3

Conf3: Number of Nodes: 8

Network interconnect: Rectangular mesh

Transient failure rates: $2.77 \times 10^{-4}(1/hr)$

Permanent failure rates: $2.77 \times 10^{-5}(0.1/hr)$

Mission time: 2000 units

Average load on the nodes: 0.25

Conf4: Number of Nodes: 8

Network interconnect: 3D hypercube

Transient failure rates: Half of the nodes had $1.38 \times 10^{-4}(0.5/hr)$

Other half had $2.77 \times 10^{-4}(1/hr)$

Permanent failure rates: $2.77 \times 10^{-5}(0.1/hr)$

Mission time: 2000 units

Average load on the nodes: 0.25

The repair rate for all of the configurations was 0.277 or (1000 /hr). This is to make sure that the nodes come out of the transient faults quickly. The sample output values are binary: If the system failed before the end of the mission time, the sample

Table 1: Normal Simulation

Configuration	Sample Mean	Sample Variance	Number of Samples	Conf. Interval
Conf1	0.0424	0.04062	2500	12.2%
Conf2	0.0032	0.0031904	5000	32%
Conf3	0.0007	0.000699	20000	34.25%
Conf4	0.0003	0.000299	40000	37%

Table 2: Importance Sampling with Bias 0.3

Configuration	Sample Mean	Sample Variance	Number of Samples	Conf. Interval
Conf1	0.0236	0.052	2600	24.29%
Conf2	0.00342	9.9×10^{-4}	1400	31.6%
Conf3	0.0006462	5.65×10^{-5}	1938	33.88%
Conf4	0.0002977	1.74×10^{-5}	2430	36.44%

value is 1. If the system did not fail before the end of the mission time, the sample value 0. This is repeated n times and the sample mean, variance and the confidence intervals are calculated as discussed in the last chapter.

Results

We run the simulations for the above mentioned configurations and the results are tabulated as in Table 1.

We then repeat the experiments with the importance sampling and a nominal bias of 30% (or 0.3). The results are as tabulated in Table 2. Comparing this with Table 1, we can observe the following.

For configuration 1, where the unreliability is quite high, the importance sampling did not provide any improvement. In fact, the importance sampling estimates

performed poorly. It shows clearly that importance sampling is not meant for such systems with a high unreliability. We might be able to reduce the bias value and see whether it performs better. But this may not offer much variance reduction over the normal simulation. This is the kind of limit that we discussed in the last section. So when the unreliability is on the order of 10^{-2} or better, it is better to go with the normal simulation itself.

For the rest of the configurations, the importance sampling estimates agree quite closely to that of the normal simulations. In fact, the number of samples taken is much less when compared to the normal simulation. This is the kind of systems for which the importance sampling heuristic was designed.

Acceleration Factor

In order to see how well the importance sampling schemes have performed, we define a factor called ‘Acceleration Factor’ (AF). It is defined as the ratio of the number of samples of the normal simulation (Num_{norm}) to that of the importance sampling scheme (Num_{imp}) required to achieve the same amount of confidence intervals.

$$AF = \frac{Num_{norm}}{Num_{imp}}$$

For the above mentioned configurations, the acceleration factors are as shown in Table 3. As we can see, the acceleration is much better as the system reliability increases.

Table 3: Acceleration Factor

Configuration	Acceleration Factor (AF)
Conf1	undefined
Conf2	3.57
Conf3	10.32
Conf4	16.46

Comparison of the Time Taken

It is also important to compare the actual time taken (for the normal simulation and for the importance sampling) to get the desired results.

We will use Conf2 (where we only got a nominal value for the acceleration factor) as an example to look at the savings in time that is possible.

Normal simulation method took an average of around 8 mins (rounded off to the nearest minute) for each of the sample points. The importance sampling scheme took an average of 6 mins for each of its sample points. Hence the ratios of the total time taken can be calculated (similar to that of the acceleration factor)

$$\begin{aligned} \text{Ratio of the time taken} &= \frac{Time_{norm}}{Time_{imp}} \\ &= \frac{8 \times 5000}{6 \times 1400} \\ &= 4.76 \end{aligned}$$

As we can see here, the actual gain in the time saved is higher than that indicated by the Acceleration Factor. This gets better with configurations having long mission times.

When using Importance sampling, the events are forced to occur more rapidly and the system is forced to go into the failure mode faster. Because of this, the actual time taken for the simulation run gets shorter when compared to the normal simulation where most of the simulation runs take the entire mission time.

Hence we gain both in terms of the reduction in the number of samples required and also the time taken for each of these sample points.

5.5 Selecting the Bias Parameter

As we have seen earlier, failure bias is an important parameter that alters the dynamics of the sample output. If it is too high or too low, the variance of the sample output will be high. There is usually some optimal value associated with each system.

Since the simulator has to work with a large variety of systems with varying unreliabilities, we have to identify a nominal value of failure bias that will work well with most system configurations. Intuitively, if the sample variance is low, the estimates converge faster and we need fewer samples to get the desired confidence intervals.

For each of the above mentioned configurations we vary the failure bias from 0.2 to 0.6 in steps of 0.1 and observe how the variance of the sample outputs change accordingly. The results of this are tabulated in Table 4.

From the above table we observe that the optimal bias value (the one that produces the least sample variance) is different for each configuration and in general, the higher

Table 4: Sample variance for different failure bias values

Config (mean)	Bias = 0.2	Bias = 0.3	Bias = 0.4	Bias = 0.5	Bias = 0.6
Conf1 (0.0424)	0.048	0.052	0.055	0.0586	-
Conf2 (0.0032)	9.4×10^{-4}	9.9×10^{-4}	1.02×10^{-3}	1.37×10^{-3}	-
Conf3 (0.0007)	6.04×10^{-5}	5.65×10^{-5}	6.21×10^{-5}	6.27×10^{-5}	7.83×10^{-5}
Conf4 (0.0003)	3.2×10^{-5}	1.74×10^{-5}	1.69×10^{-5}	1.83×10^{-5}	2.92×10^{-5}
Conf5 (2.1×10^{-5})	4.7×10^{-7}	2.3×10^{-7}	1.9×10^{-7}	1.976×10^{-7}	2.53×10^{-7}
Conf6 (5.02×10^{-7})	-	3.07×10^{-9}	2.93×10^{-9}	2.64×10^{-9}	2.86×10^{-9}

the unreliability of the system, the higher the value of this optimal bias. This is in accordance with expected behaviour as mentioned in the last section.

For Conf1, the normal simulation seems to be the best choice. For all the failure bias values, the sample variance goes much above those for the normal simulation. Conf2 seems to have an optimal value around 0.2, Conf3 around 0.3, both Conf4 and Conf5 have an optimal value around 0.4 and Conf6 around 0.5. It seems to stabilize around 0.5 for configurations with higher reliabilities.

We can use these experiments to choose a failure bias when we want to estimate the reliability of a particular system. The optimal value of the bias will vary across different configurations.

A bias value of around 0.4 seems to work well for most of the configurations. If the ‘guessed’ unreliability of the system is quite high (on order of 10^{-3}), we are better off by choosing a bias of around 0.2 – 0.3.

The simulator is intended to be used to systems whose unreliability values are quite low. So in most of the cases we can pick a bias value of around 0.4 – 0.5 which works reasonably well over a range of values.

Table 5: Effect of the Transient Failure Rates on the Unreliability

Transient Failure Rates /hr	Unreliability
1	6.4×10^{-4}
0.8	4.8×10^{-4}
0.5	1.5×10^{-4}
0.2	9.7×10^{-5}
0.1	5.3×10^{-5}

5.6 Some Typical Usages

Once we have validated the implementation and understand how to choose the failure bias, we are ready to use this simulator tool.

5.6.1 Varying the Transient Failure Rates

We will now use the simulator to check how the reliability of a system varies with the change in the failure rates of the individual components. For this purpose, we can take one of the configurations and alter the transient failure rates to observe the corresponding change in the reliability of the system.

We take the Conf3 and estimate its unreliability for various values of the transient failure rates. A failure bias of 0.3 was employed throughout the experiment. The results are tabulated in Table 5.

Here we observe that the unreliability of the system decreases gradually as the failure rates of the individual nodes decrease.

Table 6: Comparison of the Recovery Policies

Configuration	Fixed Recovery Action	RAMP Algorithm
Conf2	0.00342	0.00336
Conf3	0.0006462	0.0002841
Conf4	0.0002977	0.0001736
Conf5	2.1×10^{-5}	1.3×10^{-5}

5.6.2 Comparing the Recovery Policies

As we have seen earlier, the RAMP algorithm [30] suggests the optimal recovery action to be used whenever there is a fault in the system and a decision has to be made (regarding the choice of the recovery actions). Even though it is theoretically proved to work well, it will definitely help to know how well it performs for a typical system of our choice.

Since the simulator provides a control on choosing the recovery policy to be used for the system, we can proceed by running two experiments. We can do one set of simulation runs with a fixed recovery action and another with the RAMP algorithm and compare the results.

A Fixed Recovery Action

The three basic recovery actions are Retry, Replace and Disconnect. They had the overheads of 1, 2, and 3 units of time respectively. Based on these numbers an intuitive fixed recovery action can be formulated such as the following:

- When the node fails, try a Retry first.
- If the Retry failed, then try to Replace the faulty node by a spare node.

- If a spare does not exist, then as a final resort, Disconnect the faulty node and distribute its load to the other active nodes.

We used such a fixed recovery action in the configurations Conf2, Conf3, Conf4 and Conf5 mentioned earlier. The unreliability estimates of these configurations are shown in the Table 4. The failure bias used was 0.3

Comparison with RAMP Algorithm

Next, we run the same configuration, replacing only the fixed recovery policy with the RAMP algorithm to estimate the new unreliability. This experiment can be repeated for other configurations and the results are compared in Table 6.

From this we can see clearly that the RAMP algorithm performs better than the intuitive fixed recovery action for all of the considered configurations.

CHAPTER 6

CONCLUSIONS

This thesis work consisted of the implementation of an efficient variance reduction technique called importance sampling in a simulator testbed.

- The implementation of this technique is validated by running a series of simulations for different configurations and comparing the results with that of a normal simulation.
- The effect of the failure bias on the sample variance has been investigated to provide some guidelines in choosing a good failure bias probability for a given system.
- The tool is used to
 - Observe the change in the unreliability of the system with the change in the transient fault rates
 - Demonstrate that using an optimal failure recovery algorithm such as RAMP can significantly improve the reliability of a real-time system.

A couple of improvements can be done to the implementation to improve the usability of the tool.

- **The efficiency of the simulation.** Since the simulator was designed to model very complex interactions between the hardware and the algorithms, the number of events to be simulated is very large. A pessimistic algorithm is used to coordinate the different clocks in the nodes. Because of these reasons, each simulation run takes a very long time. It might be very beneficial if the events are studied closely to discover the potential bottlenecks.
- **Choice of the failure bias.** In the present model, it is left to the user to make a good guess of the failure bias. It might be possible to make the simulator learn from the past history and adapt the value of the failure bias.

Bibliography

- [1] S. Andradottir, D. Heyman and T. Ott, "On the Choice of Alternative Measures in Importance Sampling with Markov Chains," *Operations Research* vol.43, no.3, pp.509-519 1995.
- [2] M. Berg and I. Koren, "On Switching Policies for Modular Fault-Tolerant Computing Systems," *IEEE Trans. Computers*, Vol. C-36, pp. 1052-1062, Sept. 1987.
- [3] M.Boyd and S.Bavuso, "Simulation Modeling for Long Duration Spacecraft Control Systems," *1993 Proc. Annual Reliability and Maintainability Symposium*," pp 106-113 1993.
- [4] J.Carrasco, "Failure distance based simulation of repairable fault-tolerant systems," *Proc. of 5th International Conf. on Modeling Techniques and Tools for Computer Performance Evaluation*, 1991, pp 337-351.
- [5] J.Carrasco, "Efficient Transient Simulation of Failure/Repair Markovian Models," *Proc. of 10th Symposium on Reliable and Distributed Computing*, IEEE Computer Society Press, 1991, pp 152-161.
- [6] J.Dugan, K.Trivedi, M.Smotherman, R.Geist, "The Hybrid Automated Reliability Predictor," *AIAA Journal of Guidance, Control, and Dynamics*, 1986, vol. 9.
- [7] P.L'Ecuyer, "Efficiency Improvement and Variance Reduction," *Proc. of the 1994 Winter Simulation Conf.* pp. 122-132 1994.
- [8] Al Geist, A. Beguelin, J. Dongarra, W. Jiang, R.Manчек, V.Sunderam, *PVM: Parallel Virtual Machine*, MIT Press, 1994.

- [9] P.Glynn and D.Iglehart, "Importance Sampling for Stochastic Simulations," *Management Science*, vol. 35, no. 11, pp. 1367-1393, 1989.
- [10] A.Goyal and S.S.Lavenberg, "Modeling and analysis of computer system availability," *IBM J. Res. Develop.*, vol.31 pp.651-664, 1987.
- [11] A.Goyal, P.Heidelberger, P.Shahabuddin, "Measure Specific Dynamic Importance Sampling for Availability Simulations," *1987 Winter Simulation Conference Proceedings*, IEEE Press 1987.
- [12] A.Goyal, P.Shahabuddin, P.Heidelberger, V.F.Nicola and P.W.Glynn, "A Unified Framework for Simulating Markovian Models of Highly Dependable Systems," *IEEE Transactions on Computers*, vol.41 no.1 pp. 36-51, 1992.
- [13] J.M.Hammersley and D.C.Handscomb, *Monte Carlo Methods*, Meuthen, London, 1964.
- [14] P. Heidelberger, "Fast Simulation of Rare Events in Queueing and Reliability Models," *ACM Transactions on Modeling and Computer Simulation* Vol. 5, No. 1, 1995.
- [15] R. Jain, *FDDI Handbook*, Addison-Wesley, 1994 .
- [16] C. M. Krishna and K. G. Shin, *Real-Time Systems*, McGraw-Hill, 1997 .
- [17] L. Lamport, "Time, Clocks, and the Ordering of Events in a Distributed System," *Communications of the ACM*, Volume 21, 7, 1978.
- [18] E. E. Lewis and F. Bohm, "Monte Carlo simulation of Markov unreliability models," *Nuclear Engineering and Design*, Vol. 77, 1984.
- [19] M.Nakayama, "A Characterization of the simple failure biasing method for simulations of highly reliable Markovian Systems," *ACM Trans. Model. Comput. Simul.* vol. 4, no. 1, pp 52-88, 1994.

- [20] M. L. Puterman, *Markov Decision Processes*, John Wiley & Sons Inc., 1994.
- [21] S. M. Ross, *Applied Probability Models with Optimization Applications*, San Francisco: Holden-Day, 1970.
- [22] S. M. Ross, *Simulation*, Academic Press, 1997.
- [23] P.Shahabuddin, V.Nicola, P.Heidelberger, A.Goyal and P.Glynn, "Variance Reduction in Mean Time to Failure Simulations," *1988 Winter Simulation Conference Proceedings*, IEEE Press, 1988.
- [24] P.Shahabuddin, "Simulation and Analysis of Highly Reliable Systems," Ph.D. Thesis, Department of Operations Research, Stanford University, Palo Alto, California.
- [25] P.Shahabuddin, "Simulation of Highly Reliable Markovian Systems," *Management Science*, vol. 40, pp 333-352, 1994.
- [26] P.Shahabuddin and M.Nakayama "Estimation of reliability and its derivatives for large time horizons in Markovian systems", *1993 Winter Simulation Conference Proceedings*, IEEE Press, pp 491-499.
- [27] W. Stallings, *Handbook of Computer-Communications Standards*, Howard W. Sams & Co., 1988.
- [28] J.S. Steinman, "Breathing Time Warp," *Proceedings of the 1993 Workshop on Parallel and Distributed Simulation*, 1993.
- [29] K. K. Toutireddy, "A Testbed for Fault Tolerant Real-Time Systems," *M.S. Thesis*, Univ. of Mass. Amherst, 1996.
- [30] K. Yu, "RAMP and the Dynamic Recovery and Reconfiguration of a Distributed Real-Time System," *Ph.D. Thesis*, Univ. of Mass. Amherst, 1996.

- [31] K. Yu and I. Koren, "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.