July 1990

UILU-ENG-90-2228
CSG-127

COORDINATED SCIENCE LABORATORY
College of Engineering

DEPEND: A DESIGN ENVIRONMENT FOR PREDICTION AND EVALUATION OF SYSTEM DEPENDABILITY

Kumar K. Goswami
Ravishankar K. Iyer

UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN

SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1a. REPORT SECURITY CLASSIFICATION Unclassified	1b. RESTRICTIVE MARKINGS None
2a. SECURITY CLASSIFICATION AUTHORITY	3. DISTRIBUTION/AVAILABILITY OF REPORT Approved for public release; distribution unlimited
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE	
4. PERFORMING ORGANIZATION REPORT NUMBER(S) UILU-ENG-90-2228 (CSG 127)	5. MONITORING ORGANIZATION REPORT NUMBER(S)
6a. NAME OF PERFORMING ORGANIZATION Coordinated Science Lab University of Illinois 6b. OFFICE SYMBOL (If applicable) N/A	7a. NAME OF MONITORING ORGANIZATION NASA and JSEP
6c. ADDRESS (City, State, and ZIP Code) 1101 W. Springfield Ave. Urbana, IL 61801	7b. ADDRESS (City, State, and ZIP Code) NASA Langley Research Center, Hampton, VA 23665 and Arlington, VA 22217
8a. NAME OF FUNDING/SPONSORING ORGANIZATION NASA JSEP 8b. OFFICE SYMBOL (If applicable)	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER NASA: NAG-1-613 JSEP: N00014-90-J-1270
8c. ADDRESS (City, State, and ZIP Code) NASA Langley Hampton, VA 23665 JSEP Arlington, VA 22217	10. SOURCE OF FUNDING NUMBERS PROGRAM ELEMENT NO. / PROJECT NO. / TASK NO. / WORK UNIT ACCESSION NO.

11. TITLE (Include Security Classification)
"DEPEND: A Design Environment for Prediction and Evaluation of System Dependability"

12. PERSONAL AUTHOR(S)
Kumar K. Goswami and Ravishankar K. Iyer

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT (Year, Month, Day)	15. PAGE COUNT
Technical	FROM _____ TO _____	1990 July	15

16. SUPPLEMENTARY NOTATION

17. COSATI CODES			18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
FIELD	GROUP	SUB-GROUP	fault-tolerance, design, evaluation, simulation, fault-injection, distributed systems

19. ABSTRACT (Continue on reverse if necessary and identify by block number)

This paper describes the development of DEPEND, an integrated simulation environment for the design and dependability analysis of fault-tolerant systems. DEPEND models both hardware and software components at a functional level, and allows automatic failure injection to assess system performance and reliability. It relieves the user of the work needed to inject failures, maintain statistics and output reports. The automatic failure injection scheme is geared toward evaluating a system under high stress (workload) conditions. The failures which are injected can affect both hardware and software components. To illustrate the capability of the simulator, a distributed system which employs a prediction-based, dynamic load-balancing heuristic is evaluated. Experiments are conducted to determine the impact of failures on system performance and, to identify the failures to which the system is especially susceptible.

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT ☒ UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT. ☐ DTIC USERS	21. ABSTRACT SECURITY CLASSIFICATION Unclassified
22a. NAME OF RESPONSIBLE INDIVIDUAL	22b. TELEPHONE (Include Area Code) / 22c. OFFICE SYMBOL

DD Form 1473, JUN 86 *Previous editions are obsolete.* SECURITY CLASSIFICATION OF THIS PAGE

UNCLASSIFIED

DEPEND:

A Design Environment for Prediction and Evaluation of System Dependability

Kumar K. Goswami *Ravishankar K. Iyer*

July 1990

Center for Reliable and High Performance Computing
Coordinated Science Laboratory
University of Illinois at Urbana-Champaign
Urbana, IL 61801
USA

Abstract

This paper describes the development of DEPEND, an integrated simulation environment for the design and dependability analysis of fault-tolerant systems. DEPEND models both hardware and software components at a functional level, and allows automatic failure injection to assess system performance and reliability. It relieves the user of the work needed to inject failures, maintain statistics and output reports. The automatic failure injection scheme is geared toward evaluating a system under high stress (workload) conditions. The failures which are injected can affect both hardware and software components. To illustrate the capability of the simulator, a distributed system which employs a prediction-based, dynamic load-balancing heuristic is evaluated. Experiments are conducted to determine the impact of failures on system performance and, to identify the failures to which the system is especially susceptible.

Keywords: Fault-tolerance, Design, Evaluation, Simulation, Fault-Injection Distributed Systems.

1. Introduction

The application of computers in commercial, military, health and industrial environments has increased rapidly and along with it has risen the need for these computers to be reliable and offer high performance. Tools are now needed to assist in the design and dependability analysis of reliable computer systems. Currently there are a few tools which allow some automated design and evaluation. Analytical tools like SHARPE [Sahner 87], SAVE [Goyal 86], and METASAN [Sanders 86] have been in use for some time. Recent research has been directed toward the creation of simulators and test environments. For example, FIAT [Segall 88] is a testing environment which is designed to inject errors into a software application in order to validate error detection and recovery mechanisms. OODRA [Hwang 89] is a visually-oriented workbench that is used for evaluating the performance and reconfiguration capabilities of highly concurrent application specific architectures. FOCUS [Choi 89] is a hierarchical mixed-mode simulator that is used to evaluate the fault-tolerance and reliability of VLSI systems with specific emphasis on transient errors. In [Kubiak 89], the authors describe an event-driven simulator called GRACE and use it to study the dependability of a bit-serial processing element.

This paper describes the development of DEPEND, an integrated simulation environment for the design and dependability analysis of fault-tolerant systems. DEPEND models both hardware and software components at a functional level, and allows automatic failure injection to assess system performance and reliability. Complex hardware/software interactions can also be studied. The environment is designed to expedite and simplify the process of simulating a fault-tolerant architecture. It relieves the user of the work needed to inject failures, maintain statistics and output reports. The automatic failure injection scheme is geared toward evaluating a system under high stress (workload) conditions. The failures which are injected can affect both the hardware and the software components. To illustrate the capability of the simulator, a distributed system which employs a prediction-based, dynamic load-balancing heuristic is evaluated. Experiments are conducted to determine the impact of failures on system performance and, to identify the failures to which the system is especially susceptible.

2. The DEPEND Simulation Tool

In DEPEND a library of objects is used to simulate hardware components (e.g., CPUs, communication channels and disks). The fault-tolerant characteristics of an object are specified by the user. The

degradation), the type of failures injected (permanent or transient) and the method by which failures are injected. Each object contains routines which automatically inject failures, maintain a record of all failures injected, keep error statistics (e.g., mean time between failures) and output reports. The software components are modeled by C++ routines written by the user.

The simulation environment is shown in Figure 1. It is based on CSIM [Schwetman 86] which is a process-based simulation language written in C. The user sees an object-oriented interface because the DEPEND library is written in C++. The simulator contains a viewing system called PARAGRAPH, which graphically displays the key performance indicators during a simulation [Lee 89]. Both actual programs or trace files from actual workloads can be used in the simulations. The next subsection

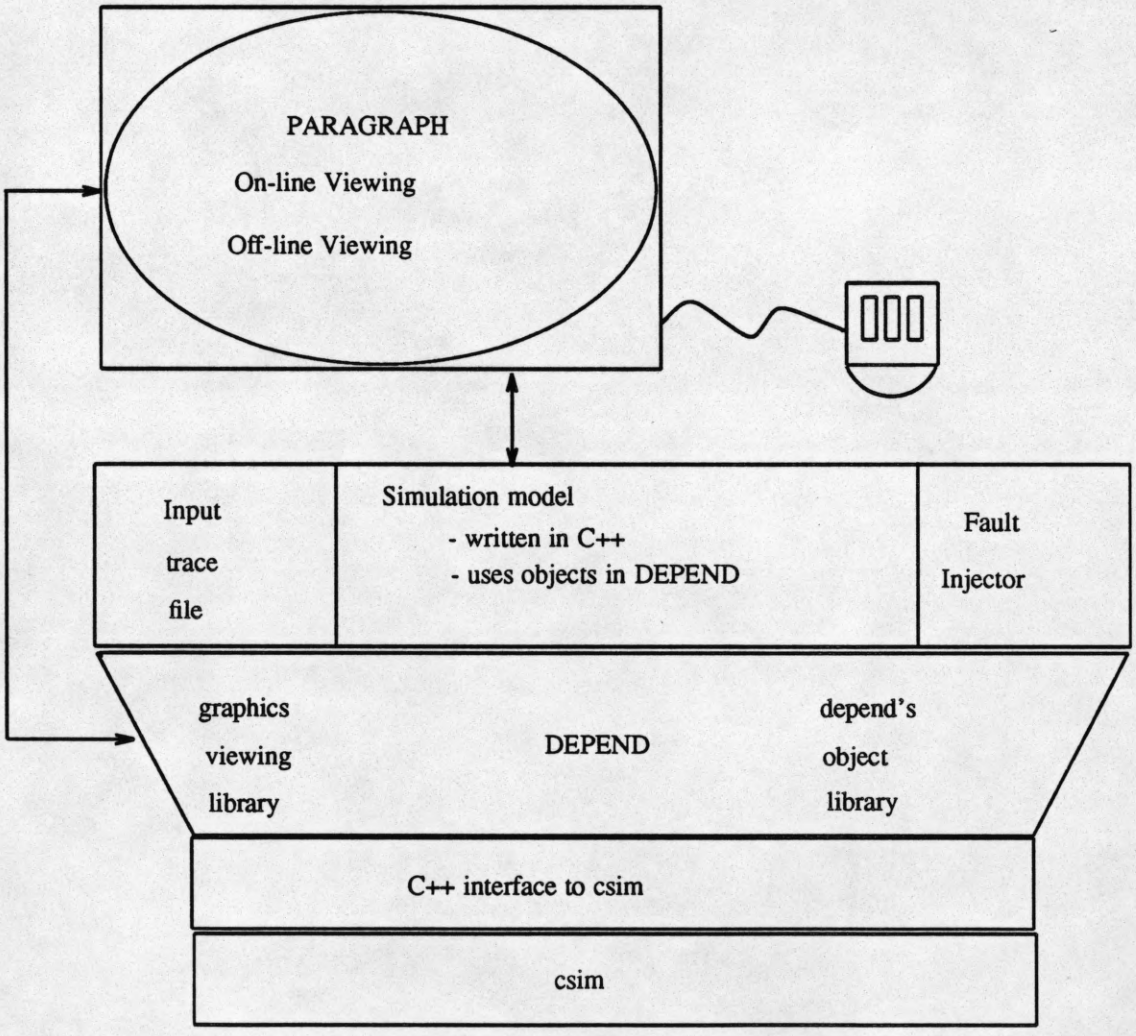

Figure 1. The DEPEND simulation environment.

describes the main objects defined in the DEPEND library.

2.1 The Objects

The most basic object in the DEPEND library is called *Basic_svr*. This object is used to simulate servers like CPUs and disks and it is also used to build more complex objects. *Basic_svr* consists of methods which can be invoked by a user to simulate the functions of a server, inject failures and repair servers. For example, the *Fault*() method is used to inject a failure into a server. Both transient and permanent failures can be injected. When a server is injected with a failure it becomes inoperative, (i.e., all processes using the server are deleted and no others are accepted until the server is repaired). In addition, event flags associated with the server are set to notify the user of a change in the server's status. These event flags can be monitored by calling methods like, *wait_for_fault*() and *wait_for_repair*() and then can be used to trigger remedial action such as reconfiguration. The *No_fault*() method is used to repair a server or its spare (for stand-by redundancy). By controlling the time between a call to *Fault*() and *No_fault*(), the duration of a failure can be controlled. The *reserve*(), *use*() and *release*() methods are used to simulate the acquisition, use and release of a server. In addition to these methods, there are many others which allow the user to check the server's status and acquire performance measurements like the server's utilization, queue length and throughput. More complex objects like the *Distributed_system* objects and the *Communication_channel* objects are built using *Basic_svr* objects.

A *Distributed_system* object simulates a distributed set of processors. This object does not specify the connectivity of these processors. The connectivity is specified by *Network* objects which are used in conjunction with the *Distributed_system* object. A *Distributed_system* object consists of many instances of *Basic_svr* and a set of failure injection routines. The instances of *Basic_svr* are used to simulate the processors. The failure injection routines automatically inject transient and permanent failures into the processors based on the specifics of the injection strategies described in Section 2.2. These routines are also responsible for maintaining a record of all injected failures and for keeping performance measurements. They also provide a full report of all injected failures, (e.g., where and when failures were injected, the mean time between failures and the mean failure/recovery duration).

A *Network* object is used to define the connectivity of the processors in a *Distributed_system*. A *Communication_channel* object is a type of a *Network* object which simulates a single bus communication

channel. It consists of a *Basic_svr* object (to simulate the communication channel), several *Port* objects (to simulate the I/O ports) and failure injection routines. Currently, three types of channel failures can be simulated. The first simply makes the *Communication_channel* inoperative, (i.e., no messages can be sent via the channel). The second causes the communication channel to occasionally lose messages. The third failure type causes the channel to intermittently corrupt messages. The latter two failure types simulate a 'noisy' communication channel.

2.2 The Fault Injection Schemes

Three fault injection strategies are currently available in DEPEND. All three are incorporated into the *Distributed_system* and the *Communication_channel* objects described above. In the first scheme, faults are injected at a constant rate. In the second scheme, faults are injected based on an exponential distribution; the duration of transients is based on exponential or normal distributions. The third approach injects faults such that there is a high probability of injections under heavy workloads. On one hand this ensures that the system is tested under stress conditions. On the other, it models the workload/failure dependency observed in [Iyer 82] & [Castillo 82]. The duration of transients is based on exponential and normal distributions.

In order to implement a workload dependent injection strategy, a statistical clustering algorithm is first used to identify high-density regions of the workload. These regions (defined as states) are used to build a state transition diagram to characterize the workload [Hsueh 88]. Associated with each state is a visit counter which counts the number of visits to that state. Also associated is a fault rate, λ, which the system experiences in that state. Periodically, the workload is monitored to identify the workload state and to update the appropriate visit counter. Based on the *injection interval*, the information from the state transition diagram is used to estimate a weighted average failure arrival rate (*Wgt_lambda*) as follows:

$$Wgt\_lambda = \sum_{i=1}^{N} visit\_ratio_i \times \lambda_i \qquad (1)$$

where:
N = the number of states

$$visit\_ratio_i = \frac{counter \text{ for } state_{j_i}}{total\ visits\ to\ all\ the\ states}$$

Once *Wgt_lambda* is determined, it is used to compute the probability of a failure injection ($P\_inject(t)$) over the last interval t (= *injection interval*) as follows:

$$P\_inject(t) = 1 - e^{-Wgt\_lambda \times t}$$

3. The Simulated Distributed System

This section briefly describes the distributed system used to demonstrate some of the features of DEPEND. A detailed description can be found in [Goswami 89].

Figure 2 is a framework for the distributed system. The simulated system contains a homogeneous set of processors connected by a single communication channel. The system is assumed to have a reconfiguration mechanism that repairs faulty processors and restarts the processes, which were executing on the processor, within a short period of time (e.g., in less than 2 minutes). Processor 0 contains the central scheduler which consists of a predictor and a scheduler. The predictor uses a statistical pattern-recognition method to predict the CPU, I/O and memory requirements of a program prior to its execution [Devarakonda 89]. The scheduler executes a load-balancing heuristic called MINQ which uses predicted process resource requirements to determine the processor load and send incoming

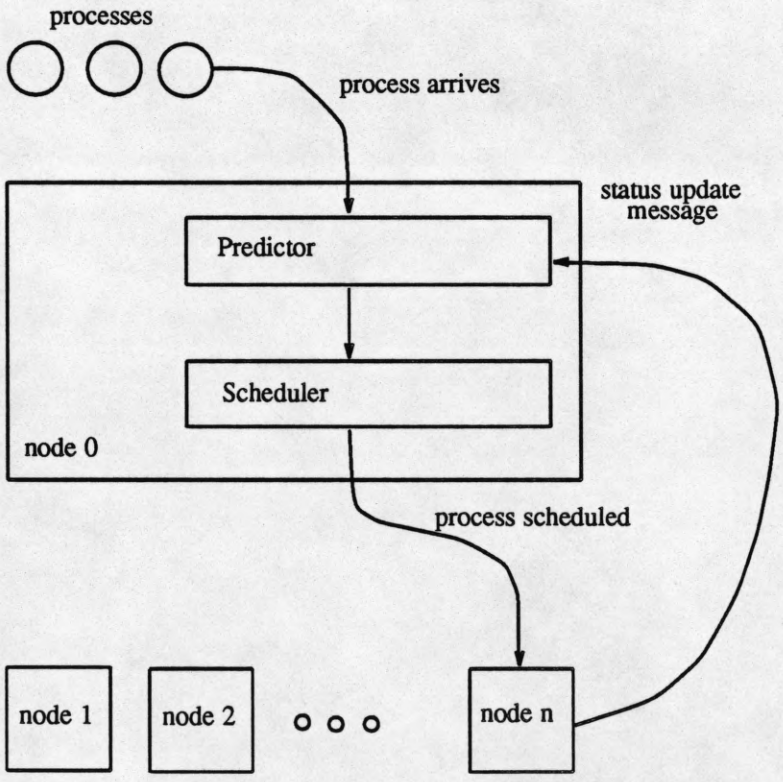

Figure 2. MINQ: Centralized load-sharing with prediction.

processes to the processor with the least load. The formula used by MINQ to estimate the processor load is as follows:

$$CPU\_LOAD_i = \sum_{j=1}^{N} \frac{CPUREQ_j}{CPUREQ_j + IOREQ_j} \qquad (2)$$

where:
N_i = the number of processes in processor i
$IOREQ_j$ is the predicted I/O requirement of process j in units of time
$CPUREQ_j$ is the predicted CPU requirement of process j in units of time

When a process completes its execution the processor sends the *actual* resources used by the process to the central scheduler via a status update message. This message is used to update the databases maintained by the predictor and decrement the *CPU_LOAD* value of the processor which sent the message.

In this study, we define the workload on a processor to be its CPU utilization. To characterize the workload, each processor has its own state transition diagram in which a state represents a specific utilization level of the processor. Each processor's utilization is measured every second and, its state transition diagram is appropriately updated. Every 20 seconds the state transition diagram of a processor is used to compute the processor's weighted average fault rate by equation 1. This value is then used to determine the probability of a fault injection ($P\_inject(t)$). Since this procedure is followed for each processor independently, multiple processors can fail at a given time.

4. The Fault Models

In DEPEND, components are simulated at a functional level, therefore, the impact of physical faults is modeled by a change in the functional behavior. The fault models used in this study simulate failures in the processor and the communication channel. Both transient and permanent failures can be injected. The duration of transients and intermittents is selected based on a normal distribution.

The *processor fault model* used to inject failures into a processor is defined as follows:

1. All processes executing on the processor are ejected.
2. Ejected processes hang until the processor is revived and then are restarted from the beginning.
3. All messages sent to the processor, while it is failed, are collected but not processed until the processor is revived.

4. If the processor contains the central scheduler, the databases maintained by the predictor and the scheduler are erased.

Two fault models are used for communication channel faults. In the first, a *message loss fault model*, the communication channel is assumed to incur intermittent failures that cause a specified percentage of all messages processed by the communication channel to be lost. A message that is lost is simply destroyed and not delivered to its destination. In the second, a *message garble fault model*, the communication channel is assumed to incur intermittent failures that corrupts pre-specified bytes in a message. Only messages that are processed by the communication channel when the channel is faulty (or 'noisy') are garbled.

These fault models were selected because they can be used to inject faults in areas that are crucial to the functionality of the distributed system discussed above. A centralized load-balancing heuristic is especially vulnerable to failures in the processor which houses the scheduler and, to failures that affect the status update messages received by the scheduler.

5. The Experiments

Our experiments showed that, for the type of system studied, a single failure, even in the processor containing the central scheduler, has an insignificant impact on the response time so long as reconfiguration is achieved within a small period of time. The problems that impact system performance quite significantly are due to intermittents [Iyer 90] occurring in close succession (e.g., 5 to 10 failures per hour). In this paper, we consider the impact of intermittent failures.

Table 1. The transition diagram used by all processors.

State	Low Util.	High Util.	Lambda (failure/hr)
1	0.0	0.4	0.1
2	0.4	0.8	0.3
3	0.8	1.0	1.0

The experiments were conducted on a ten processor system. An actual trace file containing processes run on a VAX-11/780 was used as input to the simulation. The trace file was also used to derive the state-transition diagram, shown in table 1, to characterize the workload for failure injection

purposes. The failure rates shown in the table were selected to create frequent intermittents. For each experiment, the simulation was executed five to six times with different random seeds and an average of these results is shown in the graphs below. The main performance metric used in the study is the response times for all of the processes.

5.1 Processor Failures

The *processor fault model* was used to inject failures into the processors in the system. Figure 3 shows the response times of MINQ for the 10 processor system, when transients of 0, 10, 30, 60, 90 and 120 second duration were injected. Each simulation lasted about 1.5 hours and approximately 7 failures were injected during this period. Of these failures, typically 16% were injected to the processor containing the central scheduler.

Results in figure 3 show that persistent intermittents degrade system performance considerably. For two minute transients, there was a 46% increase in the response time. However, as stated earlier, in simulations where only one or two failures were injected, there was little or no performance degradation.

5.2 Corrupted Status Update Messages

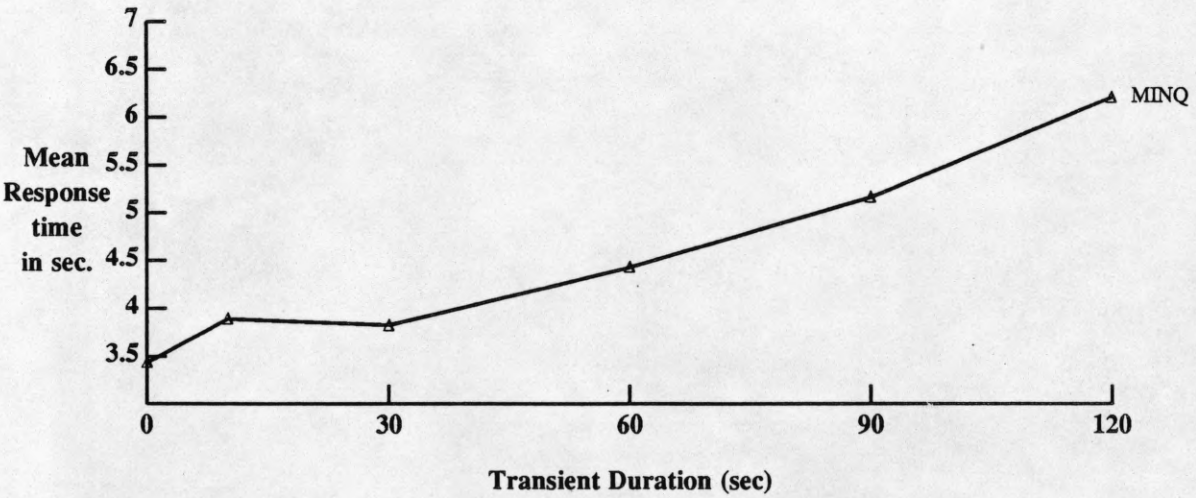

Figure 3. Impact of transient failures.

An important issue in the design of these systems is the impact of corrupted messages. To evaluate this effect, the *message garble fault model* was used in conjunction with the constant fault injection scheme to corrupt status update messages. Specifically, the fault injections were designed to corrupt the $CPUREQ_X$ field in the status update message. The $CPUREQ_X$ field is used by MINQ to decrement the *CPU_LOAD* value. Corruption of the $CPUREQ_X$ field has the most adverse impact on the database maintained by the scheduler and hence allows a worst case evaluation of the system. Figure 4 shows the results from experiments where 0, 5, 10 and 20% of the messages were corrupted. There is a 15% degradation in the response time when 5% of the messages are corrupted. The degradation more than doubles to 35% when 10% of the messages are corrupted.

5.3 Losing Status Update Messages

The *message loss fault model* was used in this experiment to destroy the status update messages sent to the central scheduler. Figure 5 is a graph of MINQ's response times when 0, 5, 10, 20 and 30% of the status update messages were destroyed. Relative to figures 3 and 4, MINQ seems to be extremely sensitive to lost status update messages. With only 10% of the messages destroyed there is nearly a 300% increase in the response time.

Upon close examination it became apparent that the poor performance was not due to the predictor or the scheduler but due to an implementation detail. MINQ uses status update messages, sent by

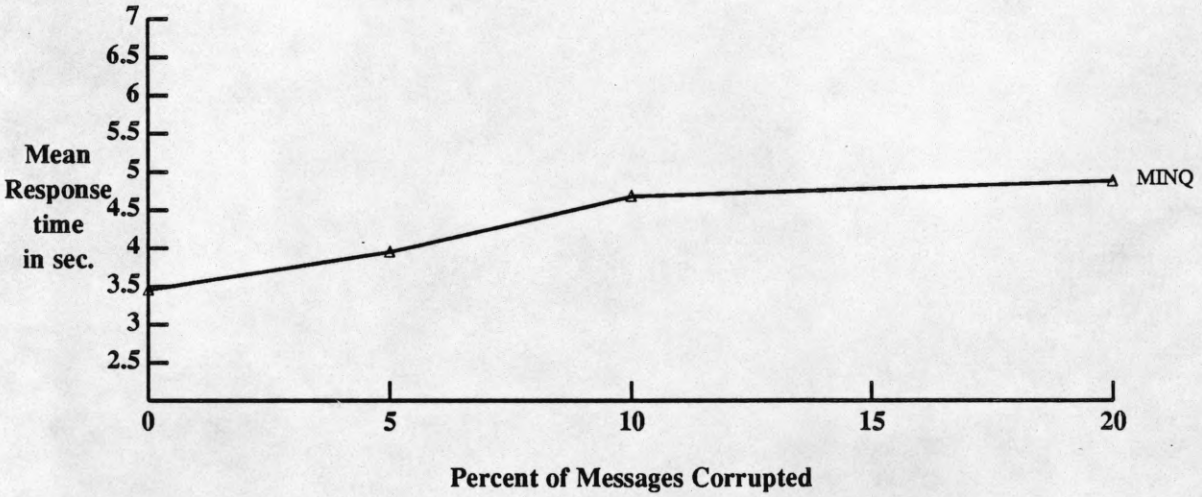

Figure 4. Impact of corrupted status update messages.

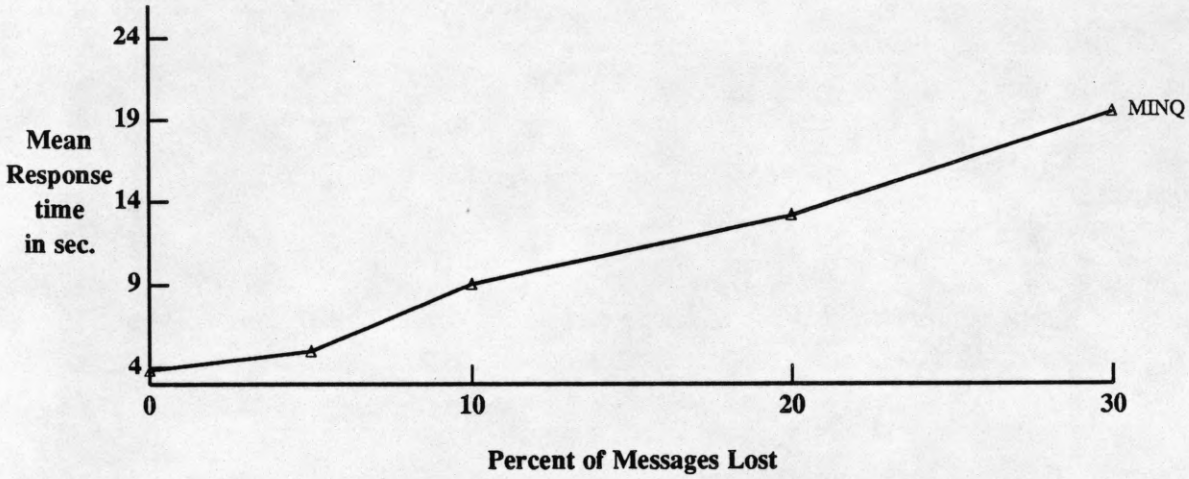

Figure 5. Impact of lost status update messages.

the processors, to decrement the processor load. When messages are lost, due to a faulty communication channel, the load values are not decremented. Processor 0, however, houses the scheduler and does not use the communication channel to send status update messages. Hence, processor 0's load is always decremented and appears lower than that of the other processors. This causes MINQ to assign a disproportionate number of processes to processor 0, resulting in the extremely poor performance shown in Figure 5. In fact, the simulations showed that processor 0 had 4 to 20 times more processes assigned to it than the other processors. Thus, faults that impair the communication channel for a reasonable period of time and prevent status update messages from reaching the scheduler can cause severe problems unless the implementation is changed.

To reduce this problem, processor 0 was forced to use the communication channel when sending status update messages. The experiment was re-run with this set up. The results for the ten processor system are shown in Figure 6. The sensitivity seen in Figure 5 has disappeared because, now all the processors lose their status update messages. MINQ (with the new set up) shows only a 16% increase in the response time when 10% of the messages are lost as opposed to the 300% increase seen in Figure 5.

An additional result can be deduced from figures 4 and 6. After approximately 10% of the status update messages are lost or corrupted, the increase in the response time levels out. At this stage, the database used by MINQ is so corrupted that MINQ seems to schedule processes randomly. Thus, increasing the number of destroyed or corrupted messages does not further degrade system performance.

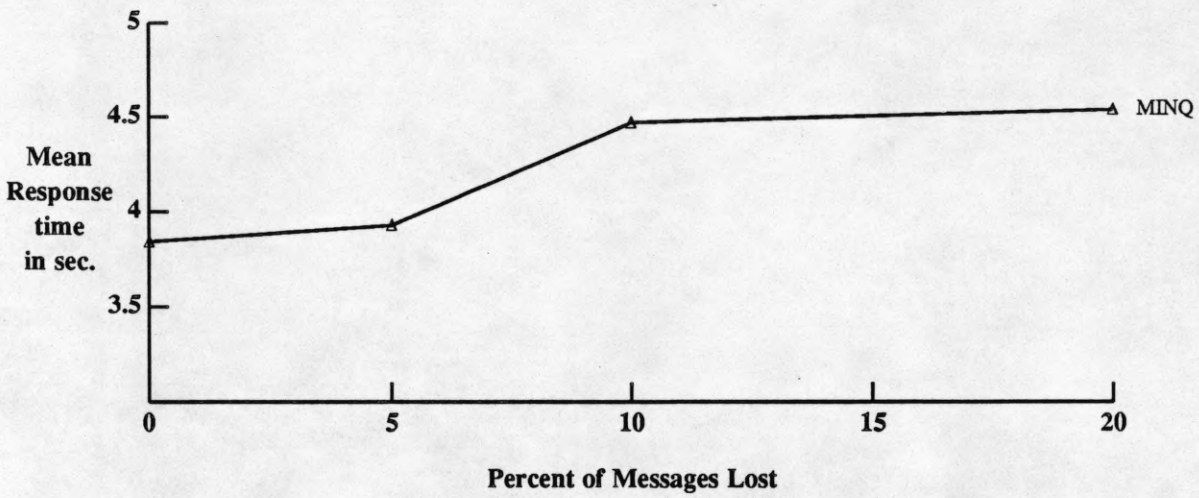

Figure 6. Impact of uniform message loss.

In the experiments, where up to 50% of the status update messages were destroyed the response time was still approximately 4.5 seconds.

6. Conclusion

This paper presented DEPEND, a simulation-based tool for design and reliability analysis of computer systems. DEPEND consists of a library of basic objects that simulate components like CPU's, communication channels and disks. The failure characteristics of an object, such as the type of fault-tolerance mechanism (e.g. stand-by redundancy or graceful degradation) used and the type of failures (transients or permanent) injected can be specified by the user. These objects serve as the building blocks with which a complex system can be simulated for dependability evaluation.

DEPEND also features a workload-based fault injection scheme which ensures an increased probability of fault injection under heavy workload conditions. One of the advantages of DEPEND is that it readily allows the user to simulate complex architectures as well as simulate the interaction between the hardware and the software.

To illustrate some of the features of DEPEND, a distributed system employing MINQ, a prediction-based centralized load-balancing heuristic, was modeled and analyzed to determine its susceptibility to intermittent failures. The results show that frequent intermittents cause significant performance degradation. MINQ was moderately sensitive to corrupted status update messages. However,

simulations using DEPEND helped identify an implementation problem which made MINQ extremely susceptible to failures that destroy status update messages.

7. Acknowledgements

This work was supported by the National Aeronautics and Space Administration under NASA grant NAG-1-613 and in part by the Joint Services Electronics Program (U.S. Army, U.S. Navy, U.S. Air Force) under grant N00014-90-J-1270. Special thanks are due to Bob Dimpsey and Inhwan Lee for their helpful suggestions.

8. References

[Castillo 82]
X. Castillo and D. Siewiorek, "A Workload Dependent Software Reliability Prediction Model," *Digest, 12th Int. Symp. on Fault-Tolerant Computing,* Santa Monica, Ca., June 22-24, 1982.

[Choi 89]
G.S. Choi, R.K. Iyer and V. Carreno, "FOCUS: An Experimental Environment for Validation of Fault Tolerant Systems, Case Study of a Jet-Engine Controller, *IEEE International Conference on Computer Design: VLSI in Computers & Processors (ICCD)*, Cambridge, MA, October 2-4, 1989, pp. 561-564.

[Devarakonda 89]
M. Devarakonda and R. K. Iyer, "Predictability of Process Resource Usage: A Measurement-Based Study of UNIX," *IEEE Trans. on Software Eng.*, Vol. 15, No. 12, December 1989.

[Goswami 89]
K. Goswami, R. Iyer, and M. Devarakonda, "Load Sharing Based on Task Resource Prediction," *Proc. 22nd Annual Hawaii International Conf. on System Sciences,* Volume 2, January 1989, pg 921-927.

[Goyal 86]
A. Goyal, W. C. Carter, E. de Souza e Silva, S. S. Lavenborg, "The system availability estimator," *Proc. 16th Int. Symp. Fault-Tolerant Comput.*, Vienna, Austria, July 1986, pp. 84-89.

[Hsueh 88]
M. C. Hsueh, R. K. Iyer and K. S. Trivedi, "Performability Modeling Based on Real Data: A Case Study," *IEEE Tran. on Comput.*, Vol., 37, No. 4, April 1988.

[Hwang 89]
D. K. Hwang and W. K. Fuchs, "CSP-Based Object-Oriented Description of Parallel Reconfigurable Architectures," *Proc. IEEE Intl. Conf. on Wafer-Scale Integration,* Jan. 1989, pp. 111-120.

[Iyer 82]
R. K. Iyer, S. E. Butner, E. J. McCluskey, "A Statistical Failure/Load Relationship: Results of a Multicomputer Study," *IEEE Trans. on Computers,* Vol. SE-8, No. 4, July 1982, pp. 354-370.

[Iyer 90]
R.K. Iyer, L.T. Young, and P.V.K. Iyer, "Automatic Recognition of Intermittent Failures: An Experimental Study of Field Data," *IEEE Transactions on Computers, Special Issue on Fault Tolerant Computing*, Vol. 39, No. 4, April, 1990, pp. 525-536.

[Kubiak 89]
K. Kubiak and W. K. Fuchs, "Reliability Analysis of Application-Specific Architectures," *Internt' l Workshop on Defect & Fault Tolerance in VLSI Systems,* Oct. 1989.

[Lee 89]
K. D. Lee, "PARAGRAPH: A Graphics Tool for Performance and Reliability Analysis," *UIUC Coordinated Science Laboratory Tech. Report # UILU-ENG-89-2239,* Nov. 1989.

[Sanders 86]
W. H. Sanders and J. F. Meyer, "METASAN: A Performability Evaluation Tool Based on Stochastic Activity Networks," *1986 Fall Joint Comp. Conf.*, Dallas, TX, Nov. 1986, pp. 807-816.

[Schwetman 86]
H. Schwetman, "CSIM: A C-BASED, Process-Oriented Simulation Language," *Proceedings Winter Simulation Conference,* 1986.

[Sahner 87]
R. A. Sahner and K. S. Trivedi, "Reliability modeling using SHARPE," *IEEE Trans. Reliability,* Vol R-36, No. 2, June 1987, pp. 186-193.

[Segall 88]
Z. Segall, D. Vrsalovic, D. Siewiorek, D. Yaskin, J. Kownacki, J. Barton, R. Dancey, A. Robinson, T. Lin, "FIAT - Fault Injection Based Automated Testing Environment," *FTCS-18,* June, 1988, pp. 102-107.