DOCTOR: An IntegrateD SOftware Fault InjeCTiOn EnviRonment for Distributed Real-time Systems *

Seungjae Han, Kang G. Shin, and Harold A. Rosenberg
Real-Time Computing Laboratory
Department of Elec. Engr. and Computer Science
The University of Michigan
Ann Arbor, Michigan 48109–2122.
email: {sjhan, kgshin, rosen}@eecs.umich.edu

Abstract

This paper presents an integrateD sOftware fault injeCTiOn enviRonment (DOCTOR) which is capable of generating synthetic workloads under which system dependability is evaluated, injecting various types of faults with different options, and collecting performance and dependability data. A comprehensive graphical user interface is also provided. The software-implemented fault-injection tool supports three types of faults: memory faults, CPU faults, and communication faults. Each injected fault may be permanent, transient or intermittent. A fault-injection plan can be formulated probabilistically, or based on the past event history. The modular organization of tools is particularly designed for distributed architectures. DOCTOR is implemented on a distributed real-time system called HARTS [1], and its capability has been tested through numerous experiments.

1 Introduction

In real-time systems the correctness of a computation depends not only on the logical correctness of the result but also on the time at which the result is produced [2]. There are a wide range of real-time applications, including continuous-media, transaction processing, and life- and mission-critical controls. Distributed architectures have proved to be well suited for meeting the timing and reliability requirements of these real-time applications. One of the major problems which the designers of distributed real-time systems face is the difficulty of evaluating their dependability. Numerous approaches have been proposed to evaluate system dependability, such as formal methods, analytical modeling, simulation, and experimental measurements.

Validating distributed real-time systems is a challenging task, since both performance and reliability constraints should be considered simultaneously, and their software and hardware architectures are very complex. In fact, the growing complexity of distributed real-time systems, due mainly to their intercomponent communications, makes most of the existing evaluation approaches intractable except for fault injection into actual prototype systems. With a common goal to accelerate the occurrence of faults or errors in the system to be tested during operation, numerous fault-injection tools have been developed using both software and hardware techniques [3, 4, 5, 6, 7, 8, 9]. Although hardware-implemented fault injectors closely mimic the real world by producing actual hardware faults, they require additional hardware which is often very expensive. Moreover, it is difficult to use them to force a distributed system into certain states, which are essential for testing distributed protocols, because the effect of hardware fault injection is usually unpredictable and hard to reproduce. Hence, more systematic error injection at a higher-level than hardware-component level is necessary for the validation of distributed real-time systems.

Based on the above observations, we have developed a software-implemented fault injection tool which can inject communication faults as well as traditional hardware faults such as memory and CPU faults. The temporal behavior of a fault may be specified as transient, intermittent, or permanent. Beside this basic fault model, it also provides a convenient

*The work reported here is supported in part by the Office of Naval Research under Grants N00014-91-J-1115 and N00014-94-1-0229, the National Aeronautic and Space Administration under Grant NAG-1493, and the National Science Foundation under Grant MIP-9203895. Any opinions, findings, and conclusions or recommendations expressed in this paper are those of the authors and do not necessarily reflect the views of the funding agencies.

user interface that allows the user to specify fault-injection timing, thus enabling the user to construct more complicated fault-injection scenarios. Another point we would like to emphasize is the importance of supporting tools for integrated fault-injection experiments. For example, using only a few application workloads is not sufficient to assess the effects of a wide range of applications on the underlying fault-tolerance mechanisms. The dependence of experimental results on the executing workloads has to be dealt with in a systematic manner.

For ease in generating workloads of various operational characteristics under which system dependability may be evaluated, we have developed a synthetic workload generation tool [10]. Also, to facilitate the collection of both performance and reliability data, an efficient data-collection tool is developed. We have been developing an automated test case selection tool [11] for systematic fault generation on a formal basis. All these tools are controlled through a unified graphic user interface. In contrast to others [4, 5, 9], we integrate tools in distributed environment.

In real-time systems where time is the most precious resource, especially, fault injection and relevant data collection must be performed with minimum overhead to the target system. Otherwise, the correctness of the validation itself becomes doubtful. To minimize the performance overhead of fault injection, only essential functions are performed on the same processor under test and relatively simple fault-injection techniques are employed, which enhances the portability of tools as well. To increase the accuracy and to minimize the overhead of data collection, we have designed a dedicated hardware for data collection.

The proposed software-implemented fault-injection environment, called an integrateD sOftware fault-injeCTiOn enviRonment, or DOCTOR for short, is implemented on HARTS. In the duplicate-match fault-detection experiment, the evaluated dependability measures such as detection coverage & latency are compared with other fault-injection tools. Communication fault injection is used to evaluate a probabilistic distributed diagnosis algorithm. The results show that the algorithm performs better than its predicted worst case, but it is quite sensitive to various coverage and inter-processor test parameters.

The paper is organized as follows. Section 2 presents the motive of our approach by discussing new requirements for fault injection in distributed real-time systems. Section 3 describes the organization of DOCTOR and its components. Section 4 presents the fault model used in DOCTOR. In Section 5, we discuss the implementation issues. Section 6 presents experiments and their results to demonstrate the usefulness of DOCTOR. The paper concludes with Section 7.

2 Fault-Injection Requirements

There are four major attributes of fault injection: a set of faults $\boldsymbol{F}$, a set of activations $\boldsymbol{A}$ which specify the workload used to exercise the system, a set of readouts $\boldsymbol{R}$, and a set of derived measures $\boldsymbol{M}$ which correspond to dependability measures such as MTTF [4]. The $\boldsymbol{FARM}$ sets for fault injection in distributed real-time systems are more complex than those for single processor systems, because the fault-tolerance mechanisms of distributed real-time systems utilize multiple processors connected by communication networks. Considerable complexities can be found in evaluating distributed diagnosis, processor group membership, replicated process group for fault masking or recovery, fault-tolerant communication, and so on. A sophisticated fault-injection scenario in both time and space dimension should be constructed to test execution paths that may occur very rarely during normal operation.

The requirements for fault injection in distributed real-time systems are enumerated below.

1. The fault model should include faults on communication links and communication adaptor circuitry as well as faults inside a processing node such as memory faults, CPU faults, or bus faults.

2. The fault injector should be able to coerce the whole target system to follow a certain intended execution path, which requires it to orchestrate all participants' behaviors. This is not achievable by randomly selecting fault type and injection timing. A systematic fault-selection aid tool and a flexible user interface are necessary for this purpose.

3. The operational characteristics of workload should be easily adjustable, especially in terms of the communication activities.

4. Fault injection or data collection must require as little modification to the target system code as possible. The performance overhead or interference by these two should also be minimized and quantifiable.

5. To obtain high-resolution timing data such as error-propagation delay or error-recovery latency, a special time-stamping technique should be employed, because clock-synchronization skews

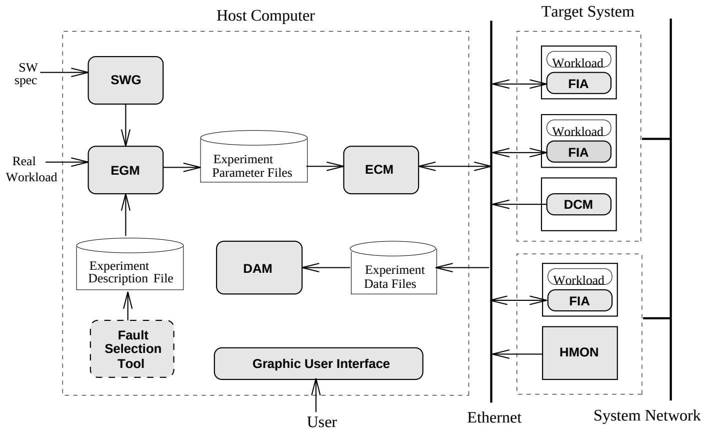

Figure 1: The organization of DOCTOR

among different processing nodes may cause unacceptably inaccurate time measurements. We solve this problem by using dedicated hardware.

3 An Integrated Experiment Environment

We provide a complete set of tools for automated fault-injection experiments. As mentioned earlier, this tool set is intended for use in distributed real-time systems (whereas most of other existing tools are intended for single processor systems). Figure 1 shows the organization of DOCTOR which forms a modular software architecture. In the distributed system architecture assumed, a host computer works as a console node and several processing nodes are connected via a system communication network and linked to the host node by an Ethernet. Each node can be a bus-based multi-processor group.

One distinct feature of this organization is the separation of components of the host computer from those of the target system. It has the advantage of reducing the run-time interference with the target system caused by fault injection, because each component runs separately and only essential components are executed on the target system. It also increases the portability of DOCTOR, since the highly system-dependent part is isolated from the rest.

Fault injector, the core part of DOCTOR, consists of three modules: Experiment Generation Module (EGM), Experiment Control Module (ECM), and Fault Injection Agent (FIA). Data Collection Module (DCM) collects experimental data during each experiment, and they are analyzed off-line after completing the experiment by Data Analysis Module (DAM). To obtain more accurate timing data with smaller performance overhead, Hardware MONitor (HMON) can be used in the place of DCM. Synthetic Workload Generator (SWG) [10] is provided to generate various artificial workloads. A tool for systematic fault selection [11] is currently under development. In addition, a comprehensive Graphic User Interface (GUI) and an automated multi-run experiment facility are provided to facilitate and automate the design and execution of fault-injection experiments. Fault-injection experiments are completely transparent to the workloads.

Each fault-injection experiment with specific workloads is called a *run*. In a fault-injection experiment, one of the factors that determine the quality of analysis results will be the number of runs. Therefore, it is very useful to automate multi-run experiments. The key problem in experiment automation is the synchronization and re-initialization of several processes involved. The level of re-initialization required depends on the status of the target system after completing each run. In some cases, it may be necessary to reset the whole system, and in some other cases, the restart of workloads may suffice. We support both levels.

3.1 EGM

The first role of EGM is to generate executable images of workloads which will be downloaded (from the host) to the target system. A workload could be run on a single processing node or be distributed among a number of nodes. The user can use real programs as workloads, or can rely on SWG for artificially-generated workloads. In either case, when the workloads are compiled and linked, the symbol-table information is extracted to be referenced by ECM.

The second role of EGM is to parse the experiment description file supplied by the user or the automatic fault-selection tool. The experiment description file describes the experiment plan which contains the information about the fault type and injection timing. EGM generates an experiment parameter file for each node involved in the experiment. These files are used by ECM to determine when to start fault injection, which type of fault to be injected, and how many times the experiment will be run, and so on.

3.2 FIA & ECM

FIA receives commands from ECM via Ethernet and executes them by injecting faults or making workloads wait/start/stop. It also reports its activities to DCM or HMON, such as the injection time, location, type, etc. FIA is a separate process which runs on the same processor as the workload.

ECM functions as an external controller. It synchronizes the start/end of each run among several nodes, and sets up an experiment environment by downloading executable images of the workload, FIAs, DCMs, and even system software if needed. ECM utilizes the experiment parameter files and the information received from FIAs to create proper commands to FIAs. For example, the symbol-table information which is contained in the experiment parameter files is used to decide memory fault-injection locations. At the same time, the information about run-time stack location from FIA is used. FIA and ECM share the responsibility of past event history management, which is particularly important for communication fault injection.

3.3 SWG

To evaluate the dependability of fault-tolerance mechanisms, we must measure dependability parameters like detection coverage and latency while executing appropriate workloads. A workload produces demands for the system resources, so the structure and behavior of the workload may affect the experimental result significantly. In DOCTOR, the user can use a synthetic workload produced by SWG instead of real programs, so that experiments can be conducted under various workload conditions. Because a synthetic workload is parameterized in the high-level description format, the user can easily control the workload characteristics.

3.4 DCM & HMON

The basic function of these tools is to log the events generated by the monitored object. The FIAs and fault-tolerance mechanisms under test generate such events, and if performance is monitored together with dependability, the event triggering instructions need to be placed in the operating system kernel. These events are the categorized, time-stamped information about the activities which we want to monitor. For example, in fault-detection experiments, two types of data are needed for the post analysis. One is the history of fault-injection reports, and the other is that of error-detection reports. Generation of events is the only overhead to the monitored object. DCM/HMON works continuously during experiments, and its function is fairly passive. [1]

If the goal of an experiment requires very high-resolution timing measurements, the time-stamp resolution supported by the underlying operating system or hardware may not be sufficient. Moreover, if the objects to be monitored are distributed among several nodes, the time-stamps of collected events are difficult to compare, because the tightness of clock synchronization among the nodes may be below the desired resolution. In order to obtain high-resolution time-stamps (e.g., 25 $n$sec), a hardware-implemented monitor (HMON) is developed. When a log request arrived through the backplane-bus, HMON obtains a time-stamp and stores the time-stamped event into local memory. It also maintains its own synchronization network so that necessary events are signaled to other HMONs. As a result, the measurement accuracy becomes independent of the system clock synchronization.

[1] To minimize performance impacts, DCM usually runs on a processor different from those on which workloads or fault-tolerance mechanisms run, but on the same backplane-bus(on the same node). The collected data are stored in files and used later for post data analysis.

Fault types		Location
Single bit	Set	Stack/Heap
Compensating	Reset	Global variables
Single byte	Toggle	User-code
Multi bytes	User defined	OS Kernel area
User defined		User defined

Table 1: Memory fault options

4 Fault Model

Hardware or software faults affect the various aspects of the system state or operational behavior, such as memory or register contents, program control flow, clock value, the condition of communication links, and so on. Modifying memory contents has been a basic technique used in software-implemented fault injectors. Faults are likely to (eventually) contaminate certain parts of memory, so memory faults can represent not only RAM errors but also emulate faults occurring in the other parts of the system. Though the memory fault model is quite powerful, some faults may affect system memory contents in a very subtle and nondeterministic way, and hence, it is very difficult to emulate such a faulty behavior with memory fault injection alone. A more sophisticated fault model is therefore required.

Currently, DOCTOR supports three types of faults: memory faults, CPU faults, and communication faults. The user can choose any combination of these three types to induce appropriate abnormal conditions. For each fault type, one can specify a number of options as shown in Tables 1, 2 and 3. We are also adding the capability of system-level error injection, such as making processes slow or fast, terminating or suspending processes, corrupting clock/timer services, corrupting system-call services, and so forth.

4.1 Memory Faults & CPU Faults

A memory fault can be injected as a single bit, two-bit (compensating), whole byte, or burst (of multiple

Fault types		Location
Single bit	Set	Data registers
Compensating	Reset	Address registers
Single byte	Toggle	Stack pointers
Whole word	User defined	Program counter
User defined		Status register

Table 2: CPU fault options

Fault types	Options
Lose messages	Faulty-link selection
Duplicate messages	Faulty-direction selection
Alter messages	Altered location
Delay messages	Altering operation
User defined	Delay control

Table 3: Communication fault options

bytes) error. The contents of memory at the selected address are partially or totally set, reset, or toggled. Beside the fault type, it is important to control the location of memory to be contaminated. The injection location either can be explicitly specified by the user, or can be chosen randomly from the physical memory space. It is sometimes desirable for a fault to be injected only into a memory section, such as the user program code, the user stack/heap, or the system area.

CPU faults can occur in data registers, address registers, the data fetching unit, control registers, the op-code decoding unit, ALU, and so on. The exact effect of faults in each processor component is highly architecture-dependent. Therefore, to emulate actual faults more directly, the utilization of detailed knowledge about the specific CPU architecture is required. However, depending on the underlying hardware and system software, accessibility to hardware components varies widely. One way to overcome this limitation is to inject erroneous effects rather than faults themselves. We chose to emulate the consequences of CPU faults in the architecture-independent level. For example, the control flow may be altered by bus line errors, instruction decoding logic errors, condition code flag errors, or control register errors (e.g., program counter). Instead of dealing with each possible case, the contents of CPU registers are used as the targets of fault injection.

4.2 Communication Faults

The communication faults in DOCTOR can cause messages to be lost, altered, duplicated, or delayed. If a node has multiple incoming and outgoing links, as in point-to-point architectures, different fault types can be specified separately for each link. The user can specify whether outgoing, incoming, or all messages are lost at the faulty link. Messages can be lost intermittently, with a probability distribution specified by the user, or alternately, every message can be lost during a certain period. Messages may be altered in a similar manner as memory faults, i.e., by corrupting single bit, two-bit compensating, or burst errors. The user can specify whether the error is to be injected into the body of a message or into its header. For delayed messages, the delay time can either be deterministic or follow some probability distribution. In addition to this set of predefined communication fault types, the user can define additional communication faults. These user-defined faults may be combinations of the predefined fault types, and may be based on the contents of individual messages or on the past message history. This variety of communication failures, and the ability to combine existing fault types and define new fault types, allow for the injection of a variety of failure semantics, including Byzantine failures.

4.3 The Control of Injection Timing

One important aspect of our fault model is its fine controllability of the fault-injection timing. In fact, the capability of injecting a proper fault instance into a proper location at a proper time is essential to the fault-injection experiments. Our fault model supports three temporal types of faults: transient, intermittent, and permanent. A transient fault is injected only once, and an intermittent fault is injected repeatedly. When injecting an intermittent fault, the user can specify the probability distribution of the fault recurrence interval. Several types of distributions like uniform distribution, exponential distribution, normal distribution, Weibull distribution and binomial distribution are provided. The user can specify the necessary constants of each distribution type, and similar probability distributions are provided for fault durations. Besides its (pre-defined) probabilistic injection timing control, DOCTOR allows the user to design fault-injection scenarios with user-specified timing control in either time-based specification or history-based specification. So, the user can directly control injection timing and fault durations with absolute or relative specifications.

5 Implementation on HARTS

The first target system[2] of DOCTOR is HARTS. HARTS is comprised of multiprocessor nodes connected by a point-to-point interconnection network.

[2] We are currently porting DOCTOR to the VxWorks based real-time distributed system.

Each HARTS node consists of several Application Processors (APs) and a Network Processor (NP). The APs are used for executing application tasks, and the NP handles most of communication processing. In the current configuration, the nodes of HARTS are VMEbus-based Motorola 68040 systems. Each HARTS node has 1–3 AP cards, an NP card, and a communication network interface board. Each node of HARTS runs an operating system called HARTOS [12]. [3] A Sun Sparcstation serves as a console. Applications and system software are downloaded from this workstation through a dedicated local Ethernet. In implementing the fault injector on HARTS, we use three techniques to inject faults concurrently with the execution of workloads. Simple memory overwrites are used for injecting memory faults, a special fault-injection protocol layer is used for injecting communication faults, and modification of CPU registers is used for injecting CPU faults.

For memory fault injection, when a pre-determined time is reached, ECM sends the corresponding FIA a message, which contains the address, the fault duration and the mask pattern. The content of the addressed location is masked by the specified pattern using AND (reset) operation, OR (set) operation, or XOR (toggle) operation. HARTS does not have any memory protection,[4] so FIA can easily overwrite any memory area. If the type of fault to be injected is permanent, the problem is not so easy, because the only way to facilitate true-permanent memory faults is to make use of system-provided memory protection support. Because HARTS is not equipped with memory protection support, we use pseudo-permanent memory faults. A permanent fault is emulated as an intermittent fault with a very small recurrence interval. FIA refreshes the contents of the fault location periodically. Another issue in injecting a memory fault is the problem of deciding the location of injection. Again, because HARTS does not use virtual memory, the symbol-table information can be used in an absolute address form.

A transient/intermittent type of CPU fault requires fault injection at run-time. The way we use is invoking a trap to the associated process and performing fault injection while the process is frozen, then allowing the process to continue execution again, which is similar to [6, 9]. In HARTOS, some of the CPU register contents are saved in the task control block and others are saved in the run-time stack, when a context switch occurs. The necessary location information about the task control block and run-time stack is obtained through call-out functions provided by the operating system. Since FIA is assigned a higher scheduling priority than other processes, it can force a process to be context-switched and returns the control to the trapped process after modifying the saved register values. This can be done very quickly because the context-switching in real-time systems is usually very fast. However, the efficient injection of permanent CPU faults is difficult to achieve. One possible way (that we chose) is changing program instructions at compile time. For example, modifying all instructions using a faulty ALU can emulate the permanent ALU fault, and overwriting a register's contents in the middle of the program execution whenever it is used can emulate the permanent register fault. By replacing or adding instructions at the assembly language level, more types of permanent CPU faults can be emulated.

Communication faults are injected by a special protocol layer, which accepts commands from FIA to determine fault instances to be injected and to build the message history structure. The fault-injection layer may be placed between any two protocol layers in the protocol stack, but is normally inserted directly below the protocol or user program to be tested. The current implementation takes advantage of the features of the x-kernel operating system, in which our communication protocols are implemented. The fault-injection layer is transparent to other protocol layers and does not add or modify the message header or data at all. The fault-injection layer does not need to be modified when it is placed between different protocol layers. If more complex fault scenarios are desired, copies of the fault-injection layer may be placed in multiple places in the protocol graph. The fault-injection layer operates by intercepting the UPI [5] operations between the protocol under test and the lower layer protocols. If it detects an operation during which a fault should be injected, based on commands from the FIA, it performs the appropriate fault injection operation. All other operations are simply passed through without modification. Outgoing and incoming messages are lost by intercepting the appropriate send and receive operations, and then discarding the message. Messages are

[3] HARTOS is primarily an extension of the functionality of pSOS^{+m} [13]. While pSOS^{+m} provides system support within a node, an extended version of the x-kernel [14] operating system coordinates communication between nodes.

[4] Like most other real-time systems, HARTS does not employ virtual memory or memory protection to reduce the unpredictability in memory access caused by page faults.

[5] All protocols in the x-kernel are implemented using same interface between layers, called the Uniform Protocol Interface (UPI).

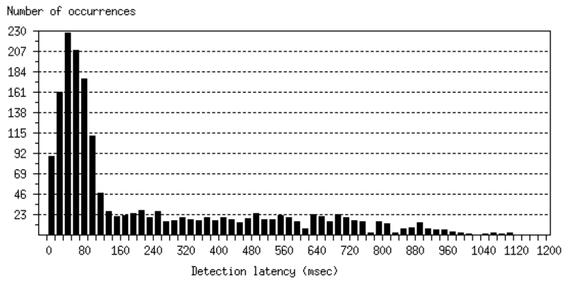

Figure 2: Error latency histogram of case 4

	case 1	case 2	case 3	case 4
matrix size	30x50x30	30x50x30	40x80x40	40x80x40
sampling freq	1/50	1/150	1/50	1/150
total runs	3000	1000	1000	3000
detection	2032	323	574	1756
error latency				
min(msec)	6	6	6	6
max(msec)	476	440	1018	1218
mean(msec)	89.10	118.49	206.76	251.57
variance	95.96	111.81	263.34	272.58

Table 4: Summary of error latency data

altered by intercepting the send or receive operation, and then changing the message contents before passing it on to the next protocol layer. Messages are delayed by stopping the current message and then scheduling a future message with the same contents, using the x-kernel event library. In order to support the user-defined fault classes, we allow messages to be stored in a message history All references to past messages in the user-defined fault description are translated into the x-kernel map library operations.

6 Experiments and Analyses

6.1 Error Latency Measurement

The goal of this experiment is to illustrate how the dependability parameters of a fault-tolerance mechanism can be measured with DOCTOR. Specifically, we measure error latency and analyze its probabilistic distribution. Error latency is the elapsed time between a fault/error injection and its detection. The experiment specification is described below. The error-detection scheme used in this experiment is the duplicate-match detection mechanism. Two identical workloads are executed simultaneously on two distinct APs of a HARTS node, and each time an element of the result matrix is generated, it is sent to the comparator program which is run on a third AP of the same HARTS node. In this experiment, a program for floating-point matrix multiplication is used as a workload. It consists of an infinite loop of the initialization step of input matrix data and the multiplication step. Since the software-implemented comparator has only limited capability of data comparison, buffers are used to store the data from workload executions and only part of the stored data are compared, so that others are discarded.

	Weibull			Exponential		Normal		
case	λ	α	error	λ	error	μ	σ	error
1	7.73	0.82	0.11	11.5	0.18	0.07	0.09	0.30
2	6.66	0.89	0.18	8.09	0.21	0.09	0.15	0.16
3	3.01	0.59	0.35	5.06	0.67	0.06	0.43	0.40
4	3.06	0.76	0.31	4.00	0.45	0.18	0.33	0.54

Table 5: Estimated distribution function parameters

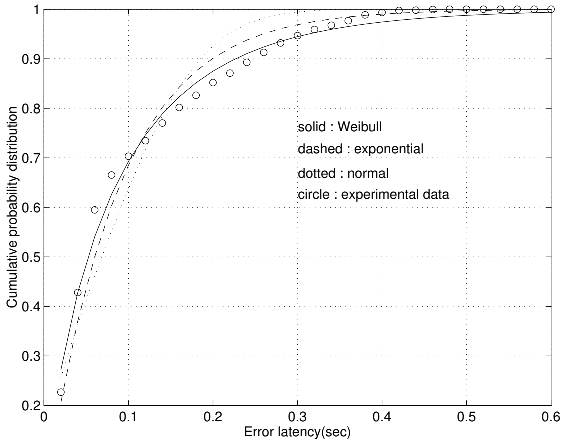

Figure 3: Fitting *cdf* of case 1

The size of matrices to be multiplied and the frequency of data sampling for comparison are the factors to be altered. Memory faults are injected into the memory section allocated for matrix data. For simplicity, we chose to inject one byte toggling transient memory faults. Because the workload keeps on re-initializing input matrix data after completing the whole multiplication, some of injected faults will be overwritten in the re-initialization step. It is why the injected faults are not always detected. Four cases are tested and measured, and the results are summarized in Table 4. One observation is that the larger matrix case has a larger mean error latency, even if the same sampling frequency is used. This is because it takes longer to generate each element to be compared. Increasing the sampling frequency also shortens the mean of error latency. Figure 2 shows the histogram of error latency for case 4 (screen dump of GUI). The

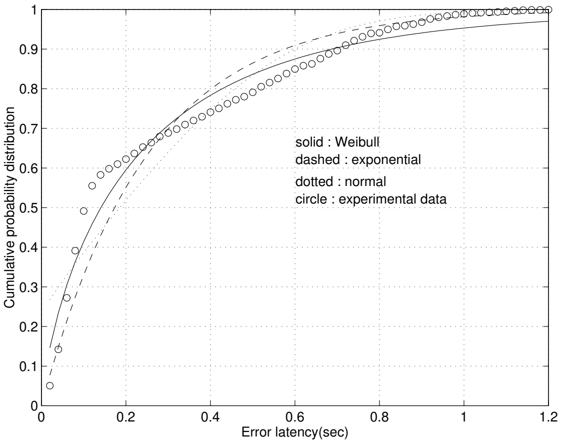

Figure 4: Fitting *cdf* of case 4

latency of fault recovery is often assumed to be distributed exponentially. However, the authors of [3, 5] observed that this was not true. In [3], Finelli showed that error latency did not fit an exponential distribution, but it rather followed the Gamma or Weibull distribution. On the other hand, in [5], Barton showed that error latency followed the normal distribution. To compare our experimental results with the others mentioned above, we performed the least-squares fit of our data to three types of distribution: normal, Weibull, and exponential distributions.

The estimated parameters of each distribution is given in Table 5 by minimizing the mean square errors. The experimental data and fitted cumulative distribution curves are plotted in Figures 3 and 4. The analysis shows that the Weibull distribution fits our data best except for case 2. Normal and exponential distributions have "inconsistent" fitting errors. Only when most errors can be detected soon after fault injection like case 1, the exponential distribution fits well, as expected. When matrix size is small but the result is compared infrequently as in case 2, or when matrix size is large but the result is compared frequently as in case 3, the normal distribution fits well, as compared to other cases. This may be because the randomness of latency increases in these cases. These results conflict Barton's result which also utilized software-implemented fault injection, but match well Finelli's result which was based on hardware-implemented fault injection. This difference can be explained by the detection mechanism and experiment tools. Although the size of data set is not large enough and the workload characteristics are varied only in a limited way, the experimental results indicate that the experimental accuracy of DOCTOR

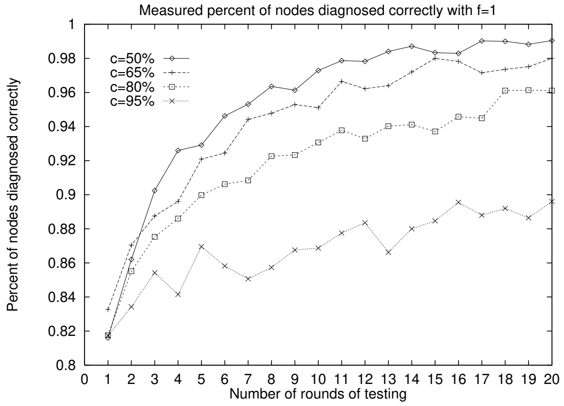

Figure 5: Percent of nodes diagnosed correctly with 1 failure mode, measured

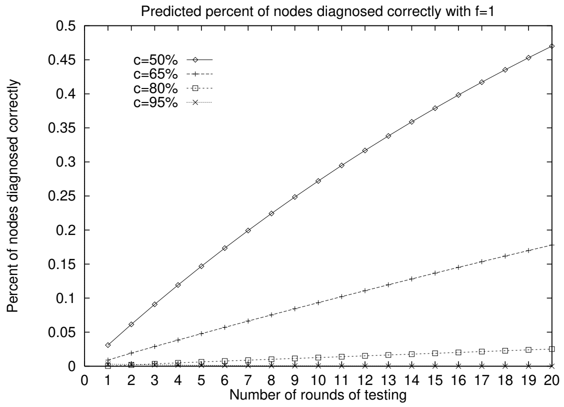

Figure 6: Percent of nodes diagnosed correctly with 1 failure mode, predicted

is close to that of hardware-implemented fault injector.

6.2 Evaluation of a Diagnosis Algorithm

In this section, we demonstrate the usefulness of software fault injection as a tool for validating dependability models of distributed protocols. By using the communication fault injection capabilities of DOCTOR, we are able to collect data on the behavior of a distributed diagnosis algorithm under a wide range of conditions. This data can then be used both to validate the predicted performance of the algorithm, and to assist in the selection of various parameters used during the execution of the algorithm.

The algorithm we chose to test is the probabilistic distributed diagnosis algorithm given in [15]. This algorithm is intended for the diagnosis of distributed systems of arbitrary connectivity. A run of the diagnosis algorithm consists of a number of rounds of testing. For the purposes of the diagnosis algorithm,

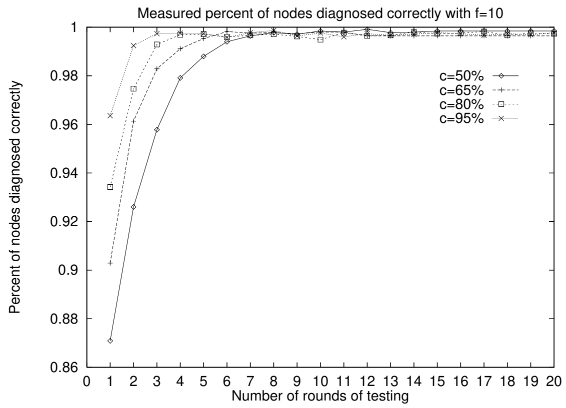

Figure 7: Percent of nodes diagnosed correctly with 10 failure modes, measured

a test graph, which is a subgraph of the undirected processor connectivity graph, is selected. Each node runs an identical test task on each round, and then exchanges the results with its neighbors in the test graph. The local result is then compared with the results received, and, if the number of mismatches is greater than some threshold, the node is considered to have failed that round. This is repeated for some number of rounds. If the number of rounds in which the node failed is greater than a second threshold, then the node is considered to be faulty.

This algorithm has a number of parameters that determine the effectiveness of the algorithm. Some of these parameters are selectable by the user, while others are functions of the system environment. The parameters that we look at in this experiment are: the number of rounds of testing (r), the coverage of interprocessor tests (c), and the number of failure modes of a test (f). The coverage of a test is the probability of a faulty processor generating an incorrect result on that test. The number of failure modes of a test is the number of possible incorrect results that a faulty processor can generate for that test. Other parameters, which we will fix for these experiments, are the probability of failure of a processor (p), the interconnection topology, and the test graph.

In the experiment, the diagnosis algorithm has two parameters to be altered, the probability of failure of a processor, and the interprocessor test coverage. The fault injection scenario is described in the following. Each time a run of experiment is initialized, the fault status of each node is independently chosen, with a probability of failure, p. On a faulty node, whenever a diagnosis-message is sent out by the diagnosis algorithm, the message history is checked to determine whether any previous diagnosis-message of the same round has been sent to another node. If not, the diagnosis-message contents are altered to a randomly selected value from the range of failure modes, with a probability equal to the test coverage. If any message of the same round had already been sent out, then the message history is used to ensure that all messages from the same round are same.

In our experiments, we chose to connect the processors of HARTS in a 9-node wrapped-square mesh. We fixed the probability of node failure at 25%. Selecting such a high figure allows us to test the algorithm under worse than expected conditions. The values of the other parameters were selected to be: c = 50%, 65%, 80%, and 90%; f = 1, 10, 20; r = [1..20]. We ran 500 iterations of the diagnosis algorithm with each combination of these parameters. The results of these experiments are summarized in Figures 5 through 8. There are a number of observations to be drawn from this data.

The first thing to notice is that in almost all cases, the measured diagnostic accuracy of the algorithm exceeded that predicted by the probabilistic model in [15], in many cases by a significant percentage. This is because the model makes a number of pessimistic assumptions, and therefore predicts only the worst-case performance of the algorithm. As a result, this distributed diagnosis algorithm may actually be appropriate for use in more systems than might be expected based only on the probabilistic model. As we see in Figure 7, using tests with 10 failure modes, even with interprocessor test coverage as low as 50%, the algorithm achieves nearly 100% correct diagnosis within 7 rounds of testing. When the test coverage is 95%, only 3 rounds are required to reach 100%. As predicted by the asymptotic analysis of the algorithm in [15], both the measured and predicted diagnostic accuracy converge to 100% as the number of tests increase, but the measured accuracy starts much higher, and converges more quickly than predicted.

One other interesting observation can be made by comparing the graphs in Figures 5 and 6 to those in Figures 7 and 8, respectively. In the cases where f, the number of failure modes, is 1, we observe that the accuracy of the diagnosis actually improves as the interprocessor test coverage decreases. This is because, when f=1, the faulty processors will always match when comparing their results with other faulty processors, and thus will be more likely to diagnose themselves as correct when the test coverage is high. This effect appears both in the predicted and observed behavior of the algorithm. When f is increased to 10,

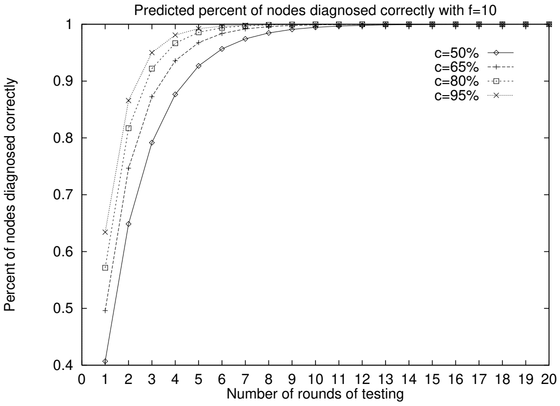

Figure 8: Percent of nodes diagnosed correctly with 10 failure modes, predicted

this effect disappears. These results indicate that tests with simple binary (e.g., good/bad) results are not a good choice when using comparison-based distributed diagnosis algorithms.

7 Conclusion

In this paper, we have presented an integrated flexible fault-injection environment called DOCTOR. It utilizes software-implemented fault injection and is intended for the validation and evaluation of distributed real-time systems. We implemented a fault injector which supports a wide range of fault type and injection options, and also developed several supporting tools such as the data-collection tool, the synthetic workload generator, and the graphic user interface. DOCTOR was implemented on a real-time distributed system, HARTS, and extensive experiments were conducted, demonstrating its power and utility. In addition, we are extending the functionality of DOCTOR in various directions. A hardware-implemented data collecting mechanism is developed, which provides high-resolution time-stamps with minimum performance overhead. We are also exploring the issues involved in formalizing both the specification of fault injection experiments, and the systematic selection of the faults to be injected. Once these extensions are completed, we will conduct more practical experiments, particularly in the area of fault-tolerant real-time communication.

References

[1] K. Shin, "HARTS: A distributed real-time architecture," *IEEE Computer*, vol. 24, no. 5, pp. 25–35, May 1991.

[2] J. Stankovic, "Misconceptions about real-time computing," *IEEE Computer*, vol. 21, no. 10, pp. 10–19, October 1988.

[3] G. Finelli, "Characterization of fault recovery through fault injection on ftmp," *IEEE Trans. Reliability*, vol. 36, no. 2, pp. 164–170, June 1987.

[4] J. Arlat, Y. Crouzet, and J. Laprie, "Fault injection for dependability validation of fault-tolerant computing systems.," in *Proc. FTCS*, pp. 348–355, June 1989.

[5] J. Barton, E. Czeck, Z. Segall, and D. Siewiorek, "Fault injection experiments using fiat," *IEEE Trans. Computers*, vol. 39, no. 4, pp. 575–581, April 1990.

[6] G. Kanawati, N. Kanawati, and J. Abraham, "FERRARI: A tool for the validation of system dependability properties," in *Proc. FTCS*, pp. 336–344. IEEE, 1992.

[7] K. Echtle and M. Leu, "The EFA fault injector for fault-tolerant distributed system testing," in *Workshop on Fault-Tolerant Parallel and Distributed Systems*, pp. 28–35. IEEE, 1992.

[8] H. Rosenberg and K. Shin, "Software fault injection and its application in distributed systems," in *Proc. FTCS*, pp. 208–217. IEEE, 1993.

[9] W. Kao, R. Iyer, and D. Tang, "FINE: A fault injection and monitoring environment for tracing the UNIX system behavior under faults," *IEEE Trans. Software Engineering*, vol. 19, no. 11, pp. 1105–1118, November 1993.

[10] D. Kiskis, *Generation of Synthetic Workloads for Distributed Real-Time Computing Systems*, PhD thesis, University of Michigan, August 1992.

[11] H. Rosenberg and K. Shin, "Specification and generation of fault-injection experiments," in *Proc. FTCS*. IEEE, 1995. Submitted for publication.

[12] K. Shin, D. Kandlur, D. Kiskis, P. Dodd, H. Rosenberg, and A. Indiresan, "A distributed real-time operating system," *IEEE Software*, pp. 58–68, September 1992.

[13] *pSOS+/68K User's Manual*, Integrated Systems Inc., 1992.

[14] N. Hutchinson and L. Peterson, "The x-Kernel: An architecture for implementing network protocols," *IEEE Trans. Software Engineering*, vol. 17, no. 1, pp. 1–13, January 1991.

[15] D. Fussell and S. Rangarajan, "Probabilistic diagnosis of multiprocessor systems with arbitrary connectivity," *Proc. FTCS*, pp. 560–565, 1989.