Software Fault Injection and its Application in Distributed Systems *

Harold A. Rosenberg and Kang G. Shin

Real-Time Computing Laboratory
Department of Electrical Engineering and Computer Science
The University of Michigan
Ann Arbor, Michigan 48109-2122.
email: {rosen, kgshin}@alps.eecs.umich.edu

Abstract

This paper describes a software fault injector (SFI) developed to facilitate the validation of dependability mechanisms on an experimental distributed real-time system called HARTS [1]. SFI introduces a number of extensions to previous work done on fault injection tools. In particular, it allows combinations of fault types to be injected in the nodes of a distributed system. It also allows control of all timing parameters of the injection at each node. A description is given of the features and implementation of SFI. As a demonstration of the utility of SFI, the results of some sample experiments are presented.

1 Introduction

Computing systems employed in life- and mission-critical applications must be highly dependable. In practice, the dependability requirements of these systems are met by employing a variety of fault-tolerance and fault-recovery mechanisms, both in software and in hardware. In a dependable distributed computer system, applications must be designed to tolerate faults in the nodes of the system. Reliable distributed applications may use mechanisms such as nested transactions [2] or process groups [3] to deal with faults in different nodes of the system. Any such mechanisms used must be rigorously validated to verify that the system meets its dependability requirements.

* The work reported here is supported in part by the Office of Naval Research under Contract N00014-91-J-1115, and the National Aeronautic and Space Administration under Grant NAG-1220, and the National Science Foundation under Grant MIP-9012549. 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.

Verification and evaluation of dependability mechanisms can be performed either statically by proof-of-correctness and probabilistic modeling, or dynamically by experimentation [4]. Complete static verification of large systems can be very difficult due to the complexity of the models. When models are used, their accuracy often depends on the dependability parameters used for the components of the system. These parameters may be difficult to estimate, and it is desirable to use parameters obtained from experimental testing of the system. As a result, dynamic verification is an important part of the validation process for dependable systems. However, dynamic verification can also be very difficult due to the large mean-time-between-failure (MTBF) of highly dependable systems. In order to measure or verify the dependability of fault-tolerant distributed systems experimentally, there must be some way to accelerate the occurrence of faults, errors, or failures in the nodes of the system.

Fault injection is the general name given to a number of techniques used to accelerate the occurrence of faults, errors, or failures in a system during dynamic verification. A fault injector is a tool which implements these techniques. Previously used fault injection techniques include hardware fault injection, software fault injection, and simulated fault injection. In hardware fault injection, faults are typically injected at the pin level [5, 6, 7, 8, 9]. One exception is [10], in which the faults were injected by exposing the system to heavy-ion radiation. In software fault injection experiments, errors have been injected by altering the content of registers or memory [11, 12, 13], or by altering sequences of instructions to emulate the behavior of hardware faults [14]. In simulated fault injection, faults or errors are injected into simulation models of the system [15, 16].

Fault injection can be used to experimentally determine the dependability parameters of the system, such as detection latency or fault coverage, and can aid in the measurement of other aspects of the system's behavior. Fault injection can also be used to test the operation of fault-tolerance mechanisms as part of the system design process. Methodologies useful in determining the locations that should be faulted in order to fully test fault-tolerance mechanisms are discussed in [17, 18].

In previously implemented fault injectors, the classes of faults and errors available to the user are either physical faults, or are chosen for their ability to represent some underlying physical fault model. In addition, the user's control over the timing of the injection is typically limited to specifying either permanent or transient faults. Even when intermittent faults can be injected (e.g., [9]), the available timing parameters are limited. These restrictions can complicate the testing of distributed dependability mechanisms, which are often designed to tolerate the occurrence of certain failure modes in the nodes of the system, without regard for the underlying causes of those failures [19]. For example, a reliable broadcast protocol may be designed to tolerate omission failures in the network, without regard to the type of fault that caused the failure. In order to validate the properties of such distributed dependability mechanisms, the user needs to be able to recreate the desired failure modes in the nodes of the system.

This paper presents SFI (Software Fault-Injector), a fault injection tool designed to facilitate the testing of distributed dependability mechanisms. SFI supports low-level fault injection for testing dependability mechanisms on single nodes. SFI also provides higher level injection methods that can be combined to emulate different failure modes in the nodes of a distributed system. This allows the user to cause the nodes of the system to exhibit the desired failure modes without having to determine the lower level faults that would cause those failures. With SFI, faults can be injected as transient, intermittent, and permanent faults, and the timing parameters of all fault types can be completely specified by the user. SFI has been implemented on HARTS, an experimental distributed real-time system [1], and it integrates quickly with any application developed for HARTS, without requiring any change in the application. Section 2 gives a brief description of the HARTS hardware and software environment. Section 3 describes the capabilities and implementation of SFI.

One important issue in the design of dependable distributed systems is detecting the occurrence of faults, either in nodes or connecting network links, and then using that information to correctly recover and maintain correct system operation. When faults only occur intermittently, the problem of detecting and correcting those faults becomes even more difficult. In Section 4 we present a sample experiment that demonstrates SFI by using it to explore the effect of intermittent communication failures on the communications between two nodes. Finally, Section 5 gives a conclusion and a discussion of future work.

2 HARTS environment

HARTS is an experimental distributed real-time system, which is being built in the Real-time Computing Laboratory at The University of Michigan [1]. It is comprised of multiprocessor nodes connected by a point–to–point interconnection network. Each HARTS node consists of several Application Processors (APs) which are used for running application tasks, and a Network Processor (NP). The NP contains the interface to the network, buffer memory, and a general-purpose processor which handles most of the processing related to communication.

In the current configuration, the nodes of the HARTS are VMEbus-based systems. Each node has 1–3 AP cards, a System Controller card, a Network Processor card, and an Ethernet processor card (ENP). The Ethernet serves as a link to the workstations used for software development. A custom NP board is currently under development.

Each of the nodes of HARTS runs the HARTOS operating system [20]. The first version of HARTOS is primarily an extension of the functionality of the uniprocessor pSOS[1] [21] real-time operating system kernel to work in a multiprocessor and distributed environment. pSOS services are enhanced to provide interprocessor communication (both unreliable datagram and RPC) and a distributed name service. While the custom NP is under development, the ENP is being used to execute the HARTOS communication software. Once the custom NP is completed, the HARTOS software will be ported to it.

Software for HARTS is developed on Sun workstations. A Sun 3/150 serves as the main connection to HARTS. HARTS applications and system software are downloaded from this workstation through the local HARTS Ethernet. The workstation is also connected to the campus computing facilities by a separate Eth-

[1]pSOS is a trademark of Software Components Group, Inc.

Memory faults		
Fault types	Interarrival time	Injection locations
Single bit Compensating Byte	Deterministic Exponential	Code Global variables Heap User defined

Table 1: Memory error options

Communication faults		
Fault types	Interarrival time	Injection locations
Lose outgoing messages Lose incoming messages Lose all messages Alter messages Delay messages	Deterministic Exponential Permanent	Header Data

Table 2: Communication failure options

Processor faults		
Fault types	Interarrival time	Injection effect
Adder Multiplier	Deterministic Permanent	User defined

Table 3: Processor failure options

ernet connection. In this way, programs developed and compiled on other workstations may be downloaded to HARTS, but HARTS executes with a dedicated local Ethernet. The workstation also serves as the console for the HARTS nodes.

3 Software Fault Injector

The Software Fault Injector (SFI) provides a suite of tools that simplify and automate the design and execution of dependability experiments. It simplifies the measurement of dependability parameters and the validation of dependability mechanisms in distributed systems, and provides lower-level injection methods to support testing of dependability mechanisms on single nodes.

SFI consists of two main components: the SFI Experiment Generator (SEG) and the SFI Control Modules (SCM). The SEG takes a user-supplied experiment description file to create the executable files and script files used to run the fault injection experiments. The SCM consists of the routines that provide the actual fault injection and behavior modification capabilities. The SEG compiles the appropriate portions of the SCM with the workload for each node. The relationship between these components is shown in Figure 1.

The types of faults, errors, and failures that can be injected with SFI are discussed in Section 3.1. Section 3.2 discusses how SFI is used to perform fault injection experiments. Section 3.3 describes how the injection is performed and how the SFI is implemented.

3.1 SFI fault models

Dependable distributed applications and operating systems must be designed to function in the presence of a variety of possible failure modes at the processing nodes and in the communication links between nodes. In order to simplify testing of such applications or other dependability mechanisms on HARTS, SFI has been designed to allow the injection of a variety of errors and failure types. The user can choose any combination of these necessary to create the errors or failure modes that are appropriate for the experiments to be run. The possible errors and failures are:

- Memory errors: Transient, intermittent, or (pseudo-)permanent errors in memory.
- Communication failures: Lost, altered, or delayed messages.
- Processor failures: Failures in functional units of the CPU.

Each error or failure has a number of possible variations that can be specified by the user in the experiment description file (see Tables 1, 2, and 3). A memory error can be injected as a single bit, two-bit compensating, or burst (byte) error. In addition, the duration of a memory error can be selected to emulate transient, intermittent or (pseudo-)permanent memory faults. A transient fault is only injected once, at a given time after the start of an experiment run. An intermittent fault is injected repeatedly at the same location. For an intermittent fault, the user can specify the distribution of the interarrival time between injections. The interarrival time can be deterministic, with a set time between injections, or can follow an exponential distribution with a given mean. Other interarrival distributions can be added if needed. If the interarrival time between injections is small, the injected fault will behave like a

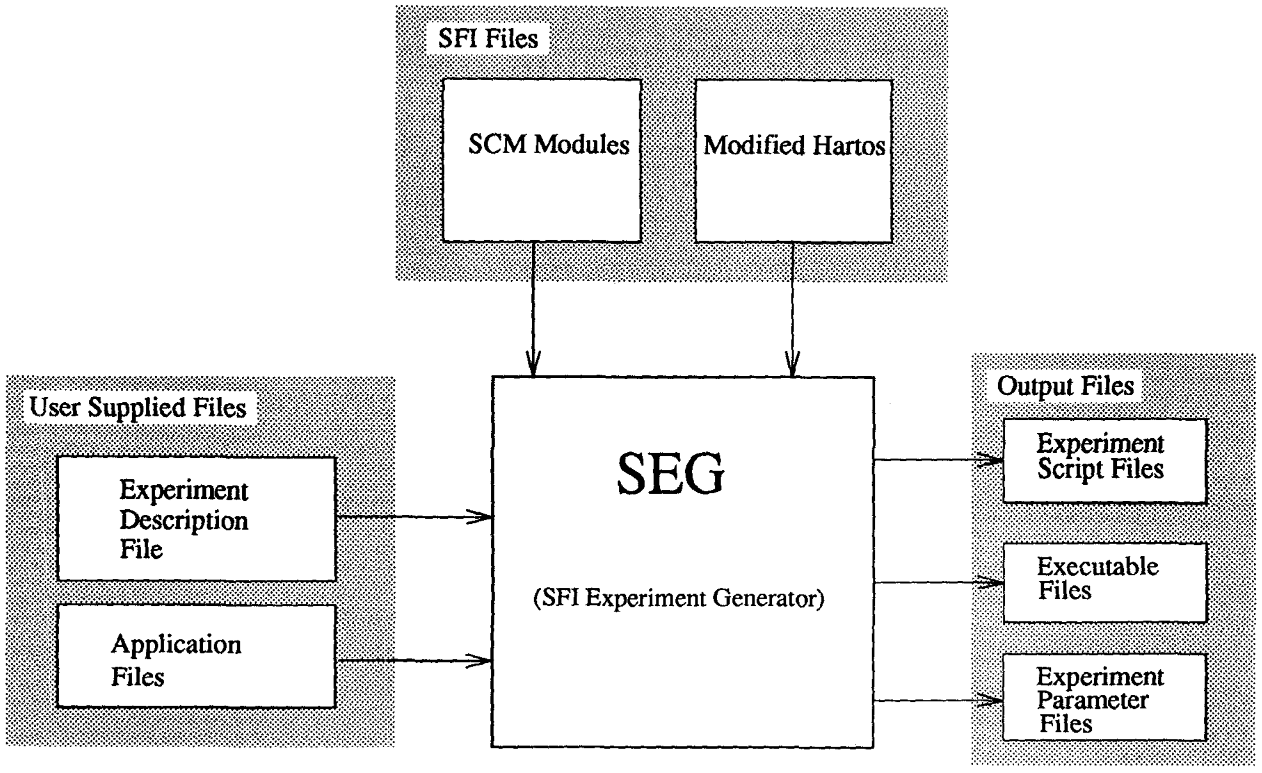

Figure 1: Relationship of SFI files.

permanent fault. Note that this is not a true permanent fault, as it is possible for the faulted location to be overwritten between injections.

The location at which a memory error is to be injected can either be explicitly specified, or can be chosen at random from a list of target locations generated by SFI. This list of target locations is created using the symbol table generated by the compiler, and can be anywhere in the memory space of the processor. If desired, the errors can be explicitly placed in data registers, global variables, code locations, or dynamically allocated memory. The user can specify which types of locations should be used. SFI can also create script files to perform multiple run experiments, each injecting a different location. For example, if the user wanted to test the effect of stuck-at faults occurring in the program code on the program's behavior, he would specify memory faults as active, with the fault type as a single bit stuck-at-zero (or stuck-at-one) fault, and the location as all program code locations. SEG would then generate script files that would run the experiment once for each memory location, injecting faults in a different location on each run.

Communication failures are specified in a manner similar to memory errors, except with some additional options. Messages can be lost, altered, or delayed. If the node has multiple incoming and outgoing links, as in a point-to-point architecture such as HARTS [1], different fault types can be specified separately for each link. Lost messages are simply not delivered to the recipient. The user can specify whether outgoing, incoming, or all messages are lost at the faulty link. Messages can be lost intermittently, with a probability specified by the user, or alternatively, every message can be lost. Messages may be altered in the same manner as memory locations, i.e., by inserting single bit, two-bit compensating, or burst errors. The user can specify whether the error is to be injected into the body of the message or into the header, which contains routing information. As before, the injection can be intermittent or permanent. For delayed messages, a method must be specified to determine how long each message will be delayed. The delay time can be either deterministic or can follow an exponential distribution with a user supplied mean. This variety of communication failures, and the ability to combine failure types, allows the injection of a variety of failure semantics, including Byzantine failures.

Processor failures emulate the effect of faults internal to the CPU, either in functional units such as

ALU, or in internal CPU data paths. These failures are implemented by allowing the user to specify alternate code sequences for any given operation or set of operations. Currently SFI implements failures in the arithmetic unit as built-in failure modes. For example, the result of all ALU operations can have any bit stuck-at a particular value. Additional fault locations can be added as needed.

In addition to the parameters that can be specified for each error or failure type, the user can specify global experiment timing parameters. These allow the user to specify the start and stop time of injection on each node. This allows experiments to be run in which a node runs correctly for some time, exhibits faulty behavior for another period of time, and then resumes correct operation. This can be used to model temporary conditions or cycles of failure and repair.

3.2 Running SFI experiments

In order to create and run an experiment using SFI, the user creates an experiment description file that provides SFI with the names of the HARTS nodes to be used, the location of the workload to be run on each node, and a description of the types of errors and/or failures to be inserted on each node. The workload can be any application that runs on HARTS. It can be a real application, or a synthetic workload generated with SWG, a synthetic workload generator developed for HARTS [22].

The experiment description file is used by the SEG to create all of the necessary files for an experiment. Based on the fault description in the experiment description file, the SEG compiles the workload together with appropriate SCM modules. In some cases the modules are versions of HARTOS system calls that have been modified so that their behavior represents the operation of the system in the presence of some underlying fault. In other cases a module may represent an injection task to be run concurrently with the workload. SEG uses the fault description and symbol table information from the compiler to create experiment parameter files that are used to initialize the injection processes. One parameter file is created for each run of the experiment. The parameters describe the fault types and locations to be used in the corresponding run. These parameter files are downloaded to the HARTS node with the executable files. By downloading the experiment parameters separately, we eliminate the need for the application to be recompiled for each experiment. Finally, SEG creates script files that are used to run the experiments. The script files download the executable and parameter files to the HARTOS nodes and run the experiments.

One important issue when running dependability experiments is the ability to collect relevant data from the experiment. SFI implements data collection by two methods. Data can be logged by writing to the console device directly, or by storing the data on the HARTS nodes, and downloading it after the experiment has completed. The data collection can be performed under the control of the executing workload, or can be done by using HMON, a monitor for distributed real-time systems being developed for HARTS [23].

3.3 SFI implementation

Due to the nature of computer systems, the accessibility of the various components in which we would like to inject faults varies considerably. As a result, a variety of methods must be used to inject different types of faults. In some cases it is possible to access the location to be faulted directly, while in other cases it is necessary to emulate the effect of a fault by causing a corresponding erroneous behavior. In implementing SFI we have used three different methods of fault injection. These are active injection, control flow alteration, and code replacement.

Active injection is performed by a process that runs concurrently with the executing workload. SFI uses active injection to inject memory errors. Control flow alteration is a technique that can be used when the functional behavior of the system is to be altered. When fault injection is activated, a running program executes an alternate instruction sequence, so that the intended function is performed incorrectly. This is particularly useful at the level of the operating system or communication protocols, where the services available to programs can be altered so that their functionality differs when fault injection is activated. SFI uses control flow alteration to inject communication failures. Code alteration can be used to inject faults in areas that are not otherwise accessible to executing programs. With this technique, faults in a functional unit can be emulated by altering program instructions that use that unit. SFI uses code alteration to emulate faults in the processor's functional units.

The injection of memory errors and communication failures is controlled by the SFI Control Process (The SCP is one of the SCM modules that is compiled together with the workload. The SCP is the first task to run, replacing the workload's startup process. When SCP starts, it checks the experiment parameters that are downloaded with the executable code. If the parameters indicate that the current experiment requires a memory error, SCP starts a high priority process, called the memory fault injection process, to perform the injection. This process injects the error at the selected address by reading, altering, and then writing back the contents of that location. Once the error has been injected, the memory fault injection process will pause itself for a time period determined by the user-defined interarrival time. This will repeat until the experiment is completed, or until a user-specified stop time.

If the current experiment requires a communication failure, SCP sets flags that are checked by the communications protocols to determine the desired behavior. When an experiment is created using the SEG, it will, based on the type of faults to be injected, be compiled with modules from the SCM that represent altered versions of some of the HARTOS system calls. The altered system calls check the flags set by the SCP, and change their operation based on those flags. When no communication failure is to be injected, the altered versions of the system calls behave identically to the original versions.

Once SCP has set up the fault injection mechanisms, it starts the workload. The workload then runs normally, with no knowledge of the fault injection processes that have been started.

The processor failures are not injected at run time. Instead they are emulated by changing all references to the faulty unit at the assembly language level. If a processor failure is to be injected, the compilation of the experiment is done in two steps. In the first step, all of the required files are compiled to the assembly language level. The SEG then searches for all instructions or instruction sequences that use the faulty functional unit, and replaces them with instructions that perform an appropriate incorrect operation. After the search-and-replace, compilation is completed using the altered files. Currently, processor failures can be injected to emulate adder and multiplier faults.

4 Experiments

One important issue in the design of dependable distributed systems is detecting the occurrence of faults/failures in the system, either in nodes or communication links. Once a failure is detected, the information can be used to recover and maintain correct system operation. When failures can occur intermittently, the problem of detecting and correcting those failures becomes even more difficult.

In this section, we present an example application of SFI to demonstrate how it might be used as an aid in both the modeling and development of fault tolerant distributed systems. We demonstrate SFI by using it to explore the effect of intermittent failures on the communication between two nodes. In Section 4.1 we develop and test a model of the effect of intermittent failures on the message delivery time between two adjacent HARTS nodes. In Section 4.2 we use SFI to test some alternate routing algorithms that use information gathered about intermittent faults.

4.1 Effect of intermittent message losses

In this section, we first develop a model to predict the effect of intermittent communication failures on message delivery times between two adjacent HARTS nodes, and then verify the accuracy of the model using SFI. For this experiment, we assume omission failure semantics, in which messages can be lost but not altered. This might represent a system operating in an environment that is outside its normal operating parameters. The poor operating conditions may cause messages to be intermittently corrupted, and thus discarded at the data link level. At higher layers of the communication subsystem, the messages would appear to be lost. We assume that since failures can be intermittent, the communication subsystem will resend a message until either a correct acknowledgment is received, or it is determined that the link has failed. We also assume that the communication subsystem has a predetermined timeout period during which it waits for an acknowledgement before resending. This timeout period is based on the transmission delay and maximum expected queuing delay at the destination node. Both the initial message or the acknowledgement can be lost.

When there are no failures, the total time required to send a message to an adjacent node and receive an acknowledgement (i.e., the round-trip time) depends only on the transmission time, the propagation delay, and the computation overhead and queueing delay at both nodes. Each time a message is lost, the round-trip time will be increased by one timeout period. When message loss is intermittent, the number of attempts required until a message is sent and acknowledged correctly can be expressed by a geometric random variable.

Let N be the random variable that represents the number of times a message is sent over a link until it is received and acknowledged correctly. Then N is a geometric random variable that gives the number of attempts until a success, where a success is the event that neither the message nor its acknowledgement is lost. If p represents the probability that any given

Probability of loss	Predicted mean time	Observed average time
0%	7.60	7.60
1%	8.12	8.18
2%	8.64	8.43
3%	9.16	8.45
5%	10.46	10.93
10%	13.58	14.15
20%	22.16	21.70
30%	34.64	36.80
50%	85.60	94.94
75%	397.60	427.18

Table 4: Predicted and observed average round-trip delay in milliseconds.

message is corrupted, then Prob[$success$] = $(1-p)^2$. Therefore,

$$\text{Prob}\{N = x\} = (1-p)^2 * (1-(1-p)^2)^{x-1}$$

and the expected number of attempts will be:

$$\overline{N} = \frac{1}{(1-p)^2}$$

We can use the probability function and mean for N to determine the expected round-trip time. The total time required to send a message and receive an acknowledgement will be the round-trip time with no failures, plus the time spent waiting for a reply every time a message is lost (i.e., the timeout period). Let RT represent the average round-trip time with no failures, and let TO represent the timeout period. If X is a random variable representing the total round-trip time in the presence of intermittent message loss, then the expected value of X will be:

$$\overline{X} = RT + (TO * (\overline{N} - 1)).$$

In order to verify this simple model, we ran experiments on HARTS using SFI. We used one node to send messages to an adjacent node. In order to match the failure semantics of the model, we chose to inject the faults at the sending node. We used SFI to inject communication faults such that messages would be lost with a given probability. We ran the experiment multiple times, each time changing the probability of losing a message.

In each run of the experiment, the sending node sent 5000 messages to the destination. We collected data on the average round-trip time, the average number of timeouts, and the number of attempts required by each message before successful transmission.

Probability of loss	Predicted mean attempts	Observed average attempts
1%	1.02	1.02
2%	1.04	1.04
3%	1.06	1.07
5%	1.11	1.11
10%	1.23	1.26
20%	1.56	1.50
30%	2.04	2.08
50%	4.00	4.02
75%	16.00	15.95

Table 5: Predicted and observed average number of attempts per message.

Number of attempts	Predicted frequency (p_i)	Observed frequency (o_i)
1	2812.5	2818
2	1230.5	1259
3	538.5	521
4	235.5	234
5	103.0	87
6	45.0	43
7	19.5	22
8	8.5	6
9+	7.0	9

Table 6: Predicted and observed frequency of number of attempts for 5000 iteration with the probability of message loss = 25%.

Table 4 shows the predicted mean and observed average round-trip delay for different message loss probabilities. The predicted values are calculated from the equation for $\overline{X}$ by using the measured value of 7.6ms for RT, and with the timeout value TO set to 26ms. Table 5 shows the predicted mean and observed average number of attempts required for different message loss probabilities. Table 6 shows both the predicted and observed number of messages requiring a given number of attempts for a probability of message loss of 25%.

The tables show that the observed data matches the results predicted by the model quite closely. This is not unexpected, given the simplicity of the model and the close modeling of the failure assumptions by the injected failures. To test the hypothesis that the observed number of attempts does follow a geometric distribution, we use a Chi-squared goodness-of-fit test on one of the data sets. The results from the other data sets are similar.

Table 6 shows the predicted and observed frequency of the number of attempts for 5000 messages sent with a probability of message loss of 25%. From this data, we can calculate the χ^2 value:

$$\chi^2 = \sum_{i=1}^{9} \frac{(o_i - p_i)^2}{p_i} = 6.12.$$

For a confidence level of 0.05, $\chi^2_{0.05} = 15.507$ with $\nu = 8$ degrees of freedom. Since $\chi^2 < \chi^2_{0.05}$, the observed data does come from a geometric distribution. This means that the experimental results we obtained using SFI validate the model.

4.2 Routing using failure data

In this section we present an example application of SFI to the system development process, and also demonstrate a few of its fault injection capabilities. In the example application, we use SFI to determine the effect of lost messages on different routing algorithms. This example also demonstrates the utility of SFI in testing distributed systems.

In many routing algorithms for point-to-point networks, routing decisions are made by determining the shortest path between two nodes. The length of a link in the path may be based on the transmission delay or congestion on that link. If a link is operating in an environment in which there is a high probability of message loss, it may be desirable to increase the length on the link to account for the lost messages. In this section, we present two simple algorithms for adding the effect of lost messages to the link length. We then test

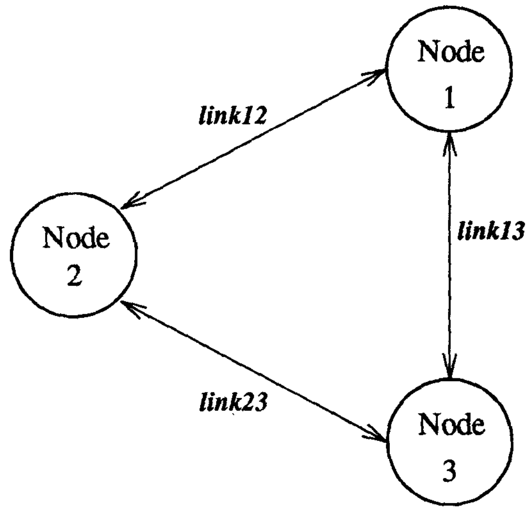

Figure 2: System configuration for routing experiments.

these algorithms using SFI, and compare the results of these tests with the results obtained when the effect of the message loss was not considered. The scenario we present is a simple one, but it demonstrates how SFI might be used in an actual application.

In the base algorithm we use for comparison, the length of a link is taken to be the transmission time of a message on that link. Each node calculates the shortest path to every other node based on this data. Messages are routed as datagrams, with each node choosing the best outgoing link for a message based on the shortest path calculations. This simple algorithm does not consider congestion or flow control.

We tested two alterations to this algorithm. The alterations employ different methods to add the effect of lost messages on a link to the length of that link. In the first method, each node keeps track of the average number of timeouts per message on each outgoing link. The average number of timeouts on a link multiplied by the timeout period gives the average incremental delay caused by lost messages for that link. This incremental delay is added to the transmission delay to calculate the current length of the link. The second method is to have each node periodically send out test messages to all of its neighbors. The delivery time for the test messages is used as the link length.

In order to test the algorithm and its modifications, we ran experiments on HARTS using SFI. The test system we used was a three node subset of HARTS, with each node connected to the other two. The test system arrangement is shown in Figure 2. Node 1 was designated the source node, and Node 3 was the destination node. Node 2 was an intermediate node that provided an alternate path from Node 1 to Node 3.

As in Section 4.1, we assumed omission failure semantics for the network. SFI was used to inject omission failures on the link connecting Node 1 to Node 3. We sent 5000 messages from Node 1 to Node 3, and measured the average delivery time. In order to determine the effect of different conditions on the algorithms, we ran two sets of experiments. In the first set of experiments, we set SFI to inject a 60 second burst of intermittent failures between two periods of fault-free operation. This represents a cycle of correct operation, followed by a failure and subsequent repair. When fault injection was active, the message loss probability was set to 20%. In the second set of experiments, fault injection was always active with a message loss probability of 20%. On each node, the link lengths in the routing table were initialized to the fault-free single link delivery time, which we measured to be 7.6 milliseconds in Section 4.1. The results of the experiments are summarized in Table 7. The times shown include both the message transmission times and the processing overheads.

In the base algorithm, the transmission times for fault-free operation are used as link costs. As a result Node 1 will always choose to send messages to Node 3 via *link13*. This is because the link lengths are not affected by message loss, and so the direct route is always shorter. Thus the average delivery time from Node 1 to Node 3 is the same as the delivery time without routing. When the link cost is augmented by the average time spent waiting for lost messages, which is the average number of timeouts per message multiplied by the timeout period, Node 1 will continue to send messages directly to Node 3 until the number of lost messages causes the length of *link13* to exceed the length of *link12 + link23*. Once the route through Node 2 becomes shorter, Node 1 will send all messages to Node 3 through Node 2, even when the link is no longer faulty. This is because it has no way of detecting a repair of the link. As a result, this algorithm does not adapt well to changing conditions.

When the link length is determined by the use of test messages, the system will adapt better to changing conditions. However, this method still has a number of flaws. If the test message is not affected by the intermittent fault, then the effect of that fault will not be considered. In addition, the frequency of the test messages will affect the correctness of the link length. If the test messages are sent frequently, the link length will better represent current conditions, but the test messages will add more overhead to the system. In this case, we sent test messages at a frequency of one test message for every hundred regular messages.

The results show that, given our assumptions about the system and its failure semantics, none of these methods is the best in all cases. To improve on these results, further refinements could be developed and then tested using SFI. This experiment demonstrates the usefulness of SFI for comparing different dependability mechanisms and testing distributed systems.

5 Conclusion

This paper has presented first an overview of SFI, a software fault injection tool developed for HARTS, a distributed real-time system. SFI improves upon previous fault injectors by allowing a wider range of injected fault types and injection options. It not only allows the injection of low-level faults, but also allows the direct injection of failures or faulty behaviors in order to simplify the validation of higher level dependability mechanisms in distributed systems. We also demonstrated the usefulness of SFI by presenting two experiments. These experiments show the application of SFI to both model validation and system testing. In the future, we intend to continue to extend the capabilities of SFI. We are working on developing fault injection methodologies for real-time systems, in which the fault injection mechanisms do not affect the timing characteristics of the system under test. SFI is currently being used as a tool in the development and testing of dependability mechanisms for HARTS.

References

[1] Kang G. Shin, "HARTS: A distributed real-time architecture", *IEEE Computer*, vol. 24, pp. 25–35, May 1991.

[2] E. Moss, "Nested Transactions: An Introduction", *in Concurrency Control and Reliability in Distributed Systems*, chapter 14, pp. 395–425. Addison Wesley, 1987.

[3] K. P. Birman and T. A. Joseph, "Reliable communication in the presence of failures", *ACM Trans. Computer Systems*, vol. 5, pp. 47–76, 2 1987.

[4] J. C. Laprie, "Dependability: Basic concepts and terminology", IFIP WG 10.4, October 1990.

Link length	Average delay with failure and repair	Average delay with no repair
Transmission delay	14.74	20.63
Transmission delay plus average timeout	14.23	15.16
Delivery time of test messages	13.74	18.43

Table 7: Average delivery delay for different method of calculating link costs (in milliseconds). Probability of message loss = 20%.

[5] K. G. Shin and Y. H. Lee, "Measurement and application of fault latency", *IEEE Trans. Computers*, vol. C-35, pp. 370–375, April 1986.

[6] G. Finelli, "Characterization of fault recovery through fault injection on ftmp", *IEEE Trans. Reliability*, vol. 36, pp. 164–170, June 1987.

[7] J. Arlat, Y. Crouzet, and J.-C. Laprie, "Fault injection for dependability validation of fault-tolerant computing systems.", *in Proc. Int'l Symp. on Fault-Tolerant Computing*, pp. 348–355, June 1989.

[8] J. Arlat et al., "Experimental evaluation of the fault tolerance of an atomic multicast system", *IEEE Trans. Reliability*, vol. 39, pp. 455–467, October 1990.

[9] J. Arlat et al., "Fault injection for dependability validation, a methodology and some applications", *IEEE Trans. Software Engineering*, vol. 16, pp. 166–182, February 1990.

[10] U. Gunneflo, J. Karlsson, and J. Torin, "Evaluation of error detection schemes using fault injection by heavy-ion radiation", *in Proc. Int'l Symp. on Fault-Tolerant Computing*, pp. 340–347, June 1989.

[11] Z. Segall et al., "Fiat – fault injection based automated testing environment", *in FTCS-18*, pp. 102–107, 1988.

[12] J. Barton, E. Czeck, Z. Segall, and D. Siewiorek, "Fault injection experiments using fiat", *IEEE Trans. Computers*, vol. 39, pp. 575–581, April 1990.

[13] R. Chillarege and N. S. Bowen, "Understanding large system failures — a fault injection experiment", *in Proc. Int'l Symp. on Fault-Tolerant Computing*, pp. 356–363, June 1989.

[14] G.A Kanawati, N.A. Kanawati, and J.A. Abraham, "FERRARI: A tool for the validation of system dependability properties", *in Proc. Int'l Symp. on Fault-Tolerant Computing*, pp. 336–344. IEEE, 1992.

[15] G. Choi, R. Iyer, and V. Carreno, "Simulated fault injection: A methodology to evaluate fault tolerant microprocessor architectures", *IEEE Trans. Reliability*, vol. 39, pp. 486–490, October 1990.

[16] R. Chillarege and R. Iyer, "Measurement-based analysis of error latency", *IEEE Trans. Computers*, vol. 36, pp. 529–537, May 1987.

[17] D. Avresky, J. Arlat, J.C. Laprie, and Yves Crouzet, "Fault injection for the formal testing of fault tolerance", *in Proc. Int'l Symp. on Fault-Tolerant Computing*, pp. 345–354. IEEE, 1992.

[18] K. Echtle and Y. Chen, "Evaluation of deterministic fault injection for fault-tolerant protocol testing", *in Proc. Int'l Symp. on Fault-Tolerant Computing*, pp. 418–425. IEEE, 1991.

[19] F. Cristian, "Understanding fault tolerant distributed systems", *Communications of the ACM*, vol. 34, pp. 56–78, February 1991.

[20] D. D. Kandlur, D. L. Kiskis, and K. G. Shin, "HARTOS: A distributed real-time operating system", *ACM SIGOPS Operating Systems Review*, vol. 23, pp. 72–89, July 1989.

[21] Software Components Group, Santa Clara, CA, *pSOS User's Guide*, 1986.

[22] D. L. Kiskis and K. G. Shin, "Generating synthetic workloads for real-time systems", *in Proc. IEEE Workshop on Real-Time Operating Systems and Software*, pp. 109–113, May 1991.

[23] P. S. Dodd and C. V. Ravishankar, "Monitoring and debugging distributed real-time programs", *Software–Practice and Experience*, vol. 22, pp. 863–877, October 1992.