Measurement-Based Analysis of System Dependability Using Fault Injection and Field Failure Data

Ravishankar K. Iyer and Zbigniew Kalbarczyk

Center for Reliable and High-Performance Computing
University of Illinois at Urbana-Champaign
1308 W. Main St., Urbana, IL 61801-2307
{iyer, kalbar}@crhc.uiuc.edu

Abstract. The discussion in this paper focuses on the issues involved in analyzing the availability of networked systems using fault injection and the failure data collected by the logging mechanisms built into the system. In particular we address: (1) *analysis in the prototype phase using physical fault injection to an actual system.* We use example of fault injection-based evaluation of a software-implemented fault tolerance (SIFT) environment (built around a set of self-checking processes called ARMORS) that provides error detection and recovery services to spaceborne scientific applications and (2) *measurement-based analysis of systems in the field.* We use example of LAN of Windows NT based computers to present methods for collecting and analyzing failure data to characterize network system dependability. Both, fault injection and failure data analysis enable us to study naturally occurring errors and to provide feedback to system designers on potential availability bottlenecks. For example, the study of failures in a network of Windows NT machines reveals that most of the problems that lead to reboots are software related and that though the average availability evaluates to over 99%, a typical machine, on average, provides acceptable service only about 92% of the time.

1 Introduction

The dependability of a system can be experimentally evaluated at different phases of its lifecycle. In the *design phase*, computer-aided design (CAD) environments are used to evaluate the design via simulation, including simulated fault injection. Such fault injection tests the effectiveness of fault-tolerant mechanisms and evaluates system dependability, providing timely feedback to system designers. Simulation, however, requires accurate input parameters and validation of output results. Although the parameter estimates can be obtained from past measurements, this is often complicated by design and technology changes. In the *prototype phase*, the system runs under controlled workload conditions. In this stage, controlled physical fault injection is used to evaluate the system behavior under faults, including the detection coverage and the recovery capability of various fault tolerance mechanisms. Fault injection on the real system can provide information about the failure process, from fault occurrence to system recovery, including error latency, propagation, detection, and recovery (which may involve reconfiguration). In the *operational phase*, a direct measurement-based approach can be used to measure systems in the

M.C. Calzarossa and S. Tucci (Eds.): Performance 2002, LNCS 2459, pp. 290–317, 2002.

field under real workloads. The collected data contain a large amount of information about naturally occurring errors/failures. Analysis of this data can provide understanding of actual error/failure characteristics and insight into analytical models. Although measurement-based analysis is useful for evaluating the real system, it is limited to detected errors. Further, conditions in the field can vary widely, casting doubt on the statistical validity of the results. Thus, all three approaches – simulated fault injection, physical fault injection, and measurement-based analysis – are required for accurate dependability analysis.

In the design phase, simulated fault injection can be conducted at different levels: the electrical level, the logic level, and the function level. The objectives of simulated fault injection are to determine dependability bottlenecks, the coverage of error detection/recovery mechanisms, the effectiveness of reconfiguration schemes, performance loss, and other dependability measures. The feedback from simulation can be extremely useful in cost-effective redesign of the system. For thorough discussion of different techniques for simulated fault injection can be found in [10].

In the prototype phase, while the objectives of physical fault injection are similar to those of simulated fault injection, the methods differ radically because real fault injection and monitoring facilities are involved. Physical faults can be injected at the hardware level (logic or electrical faults) or at the software level (code or data corruption). Heavy-ion radiation techniques can also be used to inject faults and stress the system. The detailed treatment of the instrumentation involved in fault injection experiments using real examples, including several fault injection environments is given in [10].

In the operational phase, measurement-based analysis must address issues such as how to monitor computer errors and failures and how to analyze measured data to quantify system dependability characteristics. Although methods for the design and evaluation of fault-tolerant systems have been extensively researched, little is known about how well these strategies work in the field. A study of production systems is valuable not only for accurate evaluation but also for identifying reliability bottlenecks in system design. In [10] the measurement-based analysis is based on over 200 machine-years of data gathered from IBM, DEC, and Tandem systems (note that these are not networked systems).

In this paper we discuss the current research in the area of experimental analysis of computer system dependability in the context of methodologies suited for measurement-based dependability analysis of networked systems. In particular we focus on:

- *Analysis in the prototype phase using physical fault injection to an actual system.* We use example of fault injection-based evaluation of a software-implemented fault tolerance (SIFT) environment (built around a set of self-checking processes called ARMORS, [13]) that provides error detection and recovery services to spaceborne scientific applications.
- *Measurement based analysis of systems in the field.* We use example of LAN of Windows NT based computers to present methods for collecting and analyzing failure data to characterize network system dependability.

2 Fault/Error Injection Characterization of the SIFT Environment for Spaceborne Applications

Fault/error injection is an attractive approach to the experimental validation of dependable systems. The objective of fault injection is to mimic the existence of faults and errors and hence to enable studying the failure behavior of the system. Fault\error injection can be employed to conduct detailed studies of the complex interactions between fault and fault handling mechanisms, e.g., [1] and [10]. In particular fault injection aims at (1) exposing deficiencies of fault tolerance mechanisms (i.e., fault removal), e.g., [3], and (2) evaluating coverage of fault tolerance mechanisms (i.e., fault forecasting, e.g., [2]. Number of tools were proposed to support fault injection analysis and evaluation of systems, e.g., FERRARI [14], FIAT [5], and NFTAPE [22].

This section presents an example of applying fault/error injection in assessing fault tolerance mechanisms of software implemented fault tolerance environment for spaceborne applications. In traditional spaceborne applications, onboard instruments collect and transmit raw data back to Earth for processing. The amount of science that can be done is clearly limited by the telemetry bandwidth to Earth. The Remote Exploration and Experimentation (REE) project at NASA/JPL intends to use a cluster of commercial off-the-shelf (COTS) processors to analyze the data onboard and send only the results back to Earth. This approach not only saves downlink bandwidth, but also provides the possibility of making real-time, application-oriented decisions.

While failures in the scientific applications are not critical to the spacecraft's health in this environment (spacecraft control is performed by a separate, trusted computer), they can be expensive nonetheless. The commercial components used by REE are expected to experience a high rate of radiation-induced transient errors in space (ranging from one per day to several per hour), and downtime directly leads to the loss of scientific data. Hence, a fault-tolerant environment is needed to manage the REE applications.

The missions envisioned to take advantage of the SIFT environment for executing MPI-based [19] scientific applications include the Mars Rover, the Orbiting Thermal Imaging Spectrometer (OTIS). More details on the applications and the full dependability analysis can be found in [31] and [32], respectively.

The remaining of this section presents a methodology for experimentally evaluating a distributed SIFT environment executing an REE texture analysis program from the Mars Rover mission. Errors are injected so that the consequences of faults can be studied. The experiments do not attempt to analyze the cause of the errors or fault coverage. Rather, the error injections progressively stress the detection and recovery mechanisms of the SIFT environment:

1. SIGINT/SIGSTOP injections. Many faults are known to lead to crash and hang failures . SIGINT/SIGSTOP injections reproduce these first-order effects of faults in a controlled manner that minimizes the possibility of error propagation or checkpoint corruption.
2. Register and text-segment injections. The next set of error injections represent common effects of single-event upsets by corrupting the state in the register set and text segment memory. This introduces the possibility of error propagation and checkpoint corruption.

3. Heap injections. The third set of experiments further broaden the failure scenarios by injecting errors in the dynamic heap data to maximize the possibility of error propagation. The results from these experiments are especially useful in evaluating how well intraprocess self-checks limit error propagation.

REE computational model. The REE computational model consists of a trusted, radiation-hardened (rad-hard) Spacecraft Control Computer (SCC) and a cluster of COTS processors that execute the SIFT environment and the scientific applications. The SCC schedules applications for execution on the REE cluster through the SIFT environment.

REE testbed configuration. The experiments were executed on a 4-node testbed consisting of PowerPC 750 processors running the Lynx real-time operating system. Nodes are connected through 100 Mbps Ethernet in the testbed. Between one and two megabytes of RAM on each processor were set aside to emulate local nonvolatile memory available to each node. The nonvolatile RAM is expected to store temporary state information that must survive hardware reboots (e.g., checkpointing information needed during recovery). Nonvolatile memory visible to all nodes is emulated by a remote file system residing on a Sun workstation that stores program executables, application input data, and application output data.

2.1 SIFT Environment for REE

The REE applications are protected by a SIFT environment designed around a set of self-checking processes called ARMORS (Adaptive Reconfigurable Mobile Objects of Reliability) that execute on each node in the testbed. ARMORs control all operations in the SIFT environment and provide error detection and recovery to the application and to the ARMOR processes themselves. We provide a brief summary of the ARMOR-based SIFT environment as implemented for the REE applications; additional details of the general ARMOR architecture appear in [13].

SIFT Architecture

An ARMOR is a multithreaded process internally structured around objects called elements that contain their own private data and provide elementary functions or services (e.g., detection and recovery for remote ARMOR processes, internal self-checking mechanisms, or checkpointing support). Together, the elements constitute the functionality that defines an ARMOR's behavior. All ARMORs contain a basic set of elements that provide a core functionality, including the ability to (1) implement reliable point-to-point message communication between ARMORs, (2) communicate with the local daemon ARMOR process, (3) respond to heartbeats from the local daemon, and (4) capture ARMOR state. Specific ARMORs extend this core functionality by adding extra elements.

Types of ARMORs. The SIFT environment for REE applications consists of four kinds of ARMOR processes: a Fault Tolerance Manager (FTM), a Heartbeat ARMOR, daemons, and Execution ARMORs

- *Fault Tolerance Manager (FTM).* A single FTM executes on one of the nodes and is responsible for recovering from ARMOR and node failures as well as interfacing with the external Spacecraft Control Computer (SCC).
- *Heartbeat ARMOR.* The Heartbeat ARMOR executes on a node separate from the FTM. Its sole responsibility is to detect and recover from failures in the FTM through the periodic polling for liveness.
- *Daemons.* Each node on the network executes a daemon process. Daemons are the gateways for ARMOR-to-ARMOR communication, and they detect failures in the local ARMORs.
- *Execution ARMORs.* Each application process is directly overseen by a local Execution ARMOR.

Executing REE Applications

Fig. 1 illustrates a configuration of the SIFT environment with two MPI applications (from the Mars Rover and OTIS missions) executing on a four-node testbed. Arrows in the figure depict the relationships among the various processes (e.g., the application sends progress indicators to the Execution ARMORs, the FTM is responsible for recovering from failures in the Heartbeat ARMOR, and the FTM heartbeats the daemon processes).

Each application process is linked with a SIFT interface that establishes a one-way communication channel with the local Execution ARMOR at application initialization. The application programmer can use this interface to invoke a variety of fault tolerance services provided by the ARMOR.

Error Detection Hierarchy

The top-down error detection hierarchy consists of:

- *Node and daemon errors.* The FTM periodically exchanges heartbeat messages with each daemon (every 10 s in our experiments) to detect node crashes and hangs. If the FTM does not receive a response by the next heartbeat round, it assumes that the node has failed. A daemon failure is treated as a node failure.
- *ARMOR errors.* Each ARMOR contains a set of assertions on its internal state, including range checks, validity checks on data (e.g., a valid ARMOR ID), and data structure integrity checks. Other internal self-checks available to the ARMORs include preemptive control flow checking, I/O signature checking, and deadlock/livelock detection [4]. In order to limit error propagation, the ARMOR kills itself when an internal check detects an error. The daemon detects crash failures in the ARMORs on the node via operating system calls. To detect hang failures, the daemon periodically (every 10 s in the experiments) sends "Are-you-alive?" messages to its local ARMORs.
- *REE applications.* All application crash failures are detected by the local Execution ARMOR. Crash failures in the MPI process with rank 0 can be detected by the Execution ARMOR through operating system calls (i.e., `waitpid`). The other Execution ARMORs periodically check that their MPI processes (ranks 1 through *n*) are still in the operating system's process table. If not, it concludes that the application has crashed. An application process notifies the local Execution ARMOR through its communication channel before exiting normally so that the ARMOR does not misinterpret this exit as an abnormal termination.

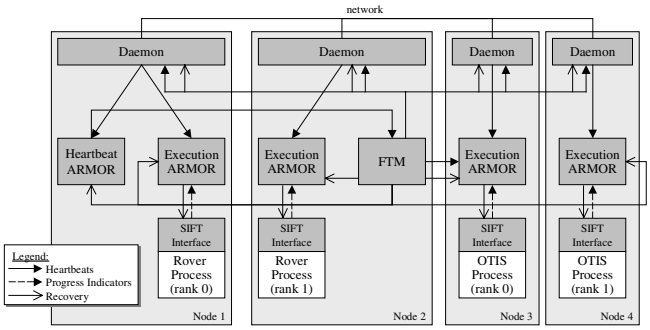

Fig. 1. SIFT Architecture for Executing two MPI Applications on a Four-Node Network.

A polling technique is used to detect application hangs in which the Execution ARMOR periodically checks for *progress indicator* updates sent by the application. A progress indicator is an "I'm-alive" message containing information that denotes application progress (e.g., a loop iteration counter). If the Execution ARMOR does not receive a progress indicator within an application-specific time period, the ARMOR concludes that the application process has hung.

Error Recovery

Nodes. The FTM migrates the ARMOR and application processes that were executing on the failed node to other working nodes in the SIFT environment.

ARMORs. ARMOR state is recovered from a checkpoint. To protect the ARMOR state against process failures, a checkpointing technique called *microcheckpointing* is used [30]. Microcheckpointing leverages the modular element composition of the ARMOR process to incrementally checkpoint state on an element-by-element basis.

REE Applications. On detecting an application failure, the Execution ARMOR notifies the FTM to initiate recovery. The version of MPI used on the REE testbed precludes individual MPI processes from being restarted within an application; therefore, the FTM instructs all Execution ARMORs to terminate their MPI processes before restarting the application. The application executable binaries must be reloaded from the remote disk during recovery.

2.2 Injection Experiments

Error injection experiments into the application and SIFT processes were conducted to: (1) stress the detection and recovery mechanisms of the SIFT environment, (2) determine the failure dependencies among SIFT and application processes, (3) measure the SIFT environment overhead on application performance, (4) measure the overhead of recovering SIFT processes as seen by the application.

1. Study the effects of error propagation and the effectiveness of internal self-checks in limiting error propagation.

The experiments used NFTAPE, a software framework for conducting injection campaigns [22].

Error Models

The error models used the injection experiments represent a combination of those employed in several past experimental studies and those proposed by JPL engineers.

- *SIGINT/SIGSTOP.* These signals were used to mimic "clean" crash and hang failures as described in the introduction.
- *Register and text-segment errors.* Fault analysis has predicted that the most prevalent faults in the targeted spaceborne environment will be single-bit memory and register faults, although shrinking feature sizes have raised the likelihood of clock errors and multiple-bit flips in future technologies. Several error injections were uniformly distributed within each run since each injection was unlikely to cause an immediate failure, and only the most frequently used registers and functions in the text segment were targeted for injection.
- *Heap errors*. Heap injections were used to study the effects of error propagation. One error was injected per run into non-pointer data values only, and the effects of the error were traced through the system.

Errors were not injected into the operating system since our experience has shown that kernel injections typically led to a crash, led to a hang, or had no impact. Maderia et al. [18] used the same REE testbed to examine the impact of transient errors on LynxOS.

Definitions and Measurements

System, experiment, and run. We use the term *system* to refer to the REE cluster and associated software (i.e., the SIFT environment and applications). The system does not include the radiation-hardened SCC or communication channel to the ground. An error injection *experiment* targeted a specific process (application process, FTM, Execution ARMOR, or Heartbeat ARMOR) using a particular error model. For each process/error model pair, a series of *runs* were executed in which one or more errors were injected into the target process.

Activated errors and failures. An injection causes an error to be introduced into the system (e.g., corruption at a selected memory location or corruption of the value in a register). An error is said to be *activated* if program execution accesses the erroneous value. A *failure* refers to a process deviating from its expected (correct) behavior as determined by a run without fault injection. The application can also fail by producing output that falls outside acceptable tolerance limits as defined by an external application-provided verification program.

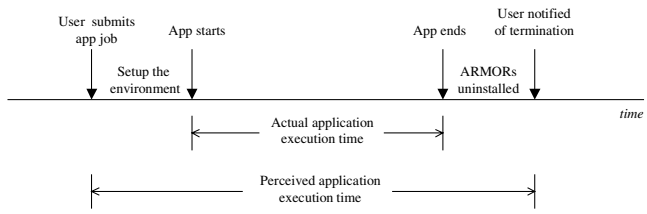

Fig. 2. Perceived vs. Actual Execution Time

A s*ystem failure* occurs when either (1) the application cannot complete within a predefined timeout or (2) the SIFT environment cannot recognize that the application has completed successfully. System failures require that the SCC reinitialize the SIFT environment before continuing, but they do not threaten the SCC or spacecraft integrity[1].

Recovery time. Recovery time is the interval between the time at which a failure is detected and the time at which the target process restarts. For ARMOR processes, this includes the time required to restore the ARMOR's state from checkpoint. In the case of an application failure, the time lost to rolling back to the most recent application checkpoint is accounted for in the application's total execution time, not in the recovery time for the application.

Perceived application execution time. The perceived execution time is the interval between the time at which the SCC submits an application for execution and the time at which the SIFT environment reports to the SCC that the application has completed.

Actual application execution time. The actual execution time is the interval between the start and the end of the application. The difference between perceived and actual execution time accounts for the time required to install the Execution ARMORs before running the application and the time required to uninstall the Execution ARMORs after the application completes (see Fig. 2). This is a fixed overhead independent of the actual application execution time.

Baseline application execution time. In the injection experiments, the perceived and actual application execution times are compared to a baseline measurement in order to determine the performance overhead added by the SIFT environment and recovery. Two measures of baseline application performance are used: (1) the application executing without the SIFT environment and without fault injection and (2) the application executing in the SIFT environment but without fault injection. The difference between these two measures provides the overhead that the SIFT processes impose on the application. Table 1 shows that the SIFT environment adds less than two seconds to the perceived application execution time. The mean application execution time and recovery time are calculated for each fault model. Ninety-five percent confidence intervals (t-distribution) are also calculated for all measurements.

Table 1. Baseline Application Execution Time

	Perceived	Actual
Without SIFT	75.71 ±0.65	75.71 ±0.65
With SIFT	77.97 ±0.48	75.74 ±0.48

2.3 Crash and Hang Failures

This section presents results from SIGINT and SIGSTOP injections into the application and SIFT processes, which were used to evaluate the SIFT environment's

[11] While the vast majority of failures in the SIFT environment will not affect the trusted SCC, in reality there exists a nonzero probability that the SCC can be impacted by SIFT failures. We discount this possibility in the paper because there is not a full-fledged SCC available for conducting such an analysis.

ability to handle crash and hang failures. We first summarize the major findings from over 700 crash and hang injections:

- All injected errors into both the application and SIFT processes were recovered.
- Recovering from errors in SIFT processes imposed a mean overhead of 5% to the application's actual execution time. This 5% overhead includes 25 cases out of roughly 700 runs in which the application was forced to block or restart because of the unavailability of a SIFT process. Neglecting those cases in which the application must redo lost computation, the overhead imposed by a recovering SIFT process was insignificant.
- Correlated failures involving a SIFT process and the application were observed. In 25 cases, crash and hang failures caused a SIFT process to become unavailable, prompting the application to fail when it did not receive a timely response from the failed SIFT process. All correlated failures were successfully recovered.

Results for 100 runs per target are summarized in Table 2. In some cases, the injection time (used to determine when to inject the error) occurred after the application completed. For these runs, no error was injected. The row "Baseline" reports the application execution time with no fault injection. One hundred runs were chosen in order to ensure that failures occurred throughout the various phases of an application's execution (including an idle SIFT environment before application execution, application submission and initialization, application execution, application termination, and subsequent cleanup of the SIFT environment).

Application Recovery

Hangs are the most expensive application failures in terms of lost processing time. Application hangs are detected using a polling technique in which the Execution ARMOR executes a thread that wakes up every 20 seconds to check the value of a counter incremented by progress indicator messages sent by the application. Because the counter is polled at fixed intervals, the error detection latency for hangs can be up to twice the checking period.

Table 2. SIGINT/SIGSTOP Injection Results

Target	Failures	Successful Recoveries	App. Exec. Time (s)		Recovery Time (s)
			Perceived	Actual	
SIGINT					
Baseline	-	-	74.78 ± 0.55	72.68 ± 0.49	-
Application	100	100	89.80 ± 1.50	87.88 ± 1.50	0.48 ± 0.05
FTM	81	81	79.60 ± 1.61	73.89 ± 0.25	0.64 ± 0.16
Execution ARMOR	100	100	77.91 ± 1.01	75.98 ± 1.00	0.61 ± 0.07
Heartbeat ARMOR	97	97	75.26 ± 0.92	74.39 ± 0.96	0.47 ± 0.12
SIGSTOP					
Baseline	-	-	71.96 ± 0.32	70.03 ± 0.27	-
Application	84	84	112.21 ± 1.87	110.21 ± 1.87	0.47 ± 0.05
FTM	97	97	76.20 ± 1.94	70.09 ± 0.88	0.79 ± 0.15
Execution ARMOR	98	98	85.01 ± 4.41	82.21 ± 4.28	0.63 ± 0.15
Heartbeat ARMOR	77	77	71.88 ± 0.24	70.24 ± 0.24	0.56 ± 0.21

SIFT Environment Recovery

FTM recovery. The perceived execution time for the application is extended if (1) the FTM fails while setting up the environment before the application execution begins or (2) the FTM fails while cleaning up the environment and notifying the Spacecraft Control Computer that the application terminated. The application is decoupled from the FTM's execution after starting, so failures in the FTM do not affect it. The only overhead in actual execution time originates from the network contention during the FTM's recovery, which lasts for only 0.6-0.7 s.

An FTM-application correlated failure. The error injections also revealed a correlated failure in which the FTM failure caused the application to restart in 2 of the 178 runs (see [32] for description of correlated failure scenarios).

The SIFT environment is able to recover from this correlated failure because the components performing the detection (Heartbeat ARMOR detecting FTM failures and Execution ARMOR detecting application failures) are not affected by the failures.

Execution ARMOR. Of the 198 crash/hang errors injected into the Execution ARMORs, 175 required recovery only in the Execution ARMOR. For these runs, the application execution overhead was negligible. The overhead reported in Table 2 (up to 10% for hang failures) resulted from the remaining 23 cases in which the application was forced to restart.

An Execution ARMOR-application correlated failure. If the application process attempted to contact the Execution ARMOR (e.g., to send progress indicator updates or to notify the Execution ARMOR that it is terminating normally) while the ARMOR was recovering, the application process blocked until the Execution ARMOR completely recovered. Because the MPI processes are tightly coupled, a correlated failure is possible if the Execution ARMOR overseeing the other MPI process diagnosed the blocking as an application hang and initiated recovery.

This correlated failure occurred most often when the Execution ARMOR hung (i.e., due to SIGSTOP injections): 22 correlated failures were due to SIGSTOP injections as opposed to 1 correlated failure resulting from an ARMOR crash (i.e., due to SIGINT injections). This is because an Execution ARMOR crash failure is detected immediately by the daemon through operating system calls, making the Execution ARMOR unavailable for only a short time. Hangs, however, are detected via a 10-second heartbeat.

2.4 Register and Text-Segment Injections

This section expands the scope of the injections to further stress the detection and recovery mechanisms by allowing for the possibility of checkpoint corruption and error propagation to another process. Results from approximately 9,000 single-bit errors into the register set and text segment of the application and SIFT processes show that:

- Most register and text-segment errors led to crash and hang failures that were recovered by the SIFT environment.
- Eleven of the approximately 700 observed failures led to system failures in which either the application did not complete or the SIFT environment did not detect that the application successfully completed. These 11 system failures resulted from

injected errors that corrupted an ARMOR's checkpoint or propagated outside the injected process.

- Text-segment errors were more likely than register errors to lead to system failures. This was because values in registers typically had a shorter lifetime (i.e., they were either never used or quickly overwritten) when compared to information stored in the text segment.

Table 3 summarizes the results of approximately 6,000 register injections and 3,000 text-segment injections into both the application and ARMOR processes. Failures are classified into four categories: segmentation faults, illegal instructions, hangs, and errors detected via assertions. The second column in Table 3 gives the number of successful recoveries vs. the number of failures for each set of experiments. Errors that were not successfully recovered led to system failures (4 due to FTM failures, 5 due to Execution ARMOR failures, and 2 due to Heartbeat ARMOR failures).

FTM recovery. Table 3 shows that the FTM successfully recovered from all register injections. Two text-segment injections were detected through assertions on the FTM's internal data structures, and both of these errors were recovered.

Table 3 also shows that the FTM could not recover from four text-segment errors. In each case, the error corrupted the FTM's checkpoint prior to crashing. Because the checkpoint was corrupted, the FTM crashed shortly after being recovered. This cycle of failure and recovery repeated until the run timed out.

There were seven cases of a correlated failure in which the FTM failed during the application's initialization: three from text-segment injections and four from register injections. Both the FTM and the application recovered from all seven correlated failures.

Table 3. Register and Text-Segment Injection Results

Target	Recoveries/ Failures	Failure Classification				App. Exec. Time (s)		Recoverry Time (s)
		Seg. fault	Illegal instr.	Hang	Assert-ion	Perceived	Actual	
Baseline	-	-	-	-	-	71.96 ±0.32	70.03 ±0.27	-
Register Injections								
Application	95 / 95	71	4	20	0	90.70 ±2.57	88.81 ±2.57	0.70 ±0.21
FTM	84 / 84	58	6	16	4	75.65 ±1.54	73.42 ±1.28	0.71 ±0.03
Execution ARMOR	77 / 80	56	6	15	3	76.19 ±1.82	73.56 ±1.83	0.45 ±0.08
Heartbeat ARMOR	77 / 77	62	6	8	1	73.00 ±0.22	70.66 ±0.21	0.31 ±0.04
Text-segment Injections								
Application	82 / 82	41	23	18	0	89.47 ±2.87	87.49 ±2.88	1.05 ±0.33
FTM	84 / 88	53	28	5	2	76.47 ±2.87	71.00 ±2.31	0.51 ±0.05
Execution ARMOR	93/ 95	45	31	11	8	77.48 ±1.93	74.83 ±1.86	0.43 ±0.04
Heartbeat ARMOR	95 / 97	53	33	11	0	73.23 ±0.37	71.21 ±0.36	0.30 ±0.01

Execution ARMOR recovery. Three register injections and two text-segment injections into the Execution ARMOR led to system failure. In each of these cases, the error propagated to other ARMOR processes or to the Execution ARMOR's checkpoint.

One text-segment injection and three register injections caused errors in the Execution ARMOR to propagate to the FTM (i.e., the error was not fail-silent). Although the Execution ARMOR did not crash, it sent corrupted data to the FTM when the application terminated, causing the FTM to crash. The FTM state in its checkpoint was not affected by the error, so the FTM was able to recover to a valid state. Because the FTM did not complete processing the Execution ARMOR's notification message, the FTM did not send an acknowledgment back to the Execution ARMOR. The missing acknowledgment prompted the Execution ARMOR to resend the faulty message, which again caused the FTM to crash. This cycle of recovery followed by the retransmission of faulty data continued until the run timed out.

One of the text-segment injections caused the Execution ARMOR to save a corrupted checkpoint before crashing. When the ARMOR recovered, it restored its state from the faulty checkpoint and crashed shortly thereafter. This cycle repeated until the run timed out.

In addition to the system failures described above, three text-segment injections into the Execution ARMOR resulted in the restarting of the texture analysis application. All three of these correlated failures were successfully recovered.

Heartbeat ARMOR recovery. The Heartbeat ARMOR recovered from all register errors, while text-segment injections brought about two system failures. Although no corrupted state escaped the Heartbeat ARMOR, the error prevented the Heartbeat ARMOR from receiving incoming messages. Thus, the Heartbeat ARMOR falsely detected that the FTM had failed, since it did not receive a heartbeat reply from the FTM. The ARMOR then began to initiate recovery of the FTM by (1) instructing the FTM's daemon to reinstall the FTM process, and (2) instructing the FTM to restore its state from checkpoint after receiving acknowledgment that the FTM has been successfully reinstalled.

Among the successful recoveries from text-segment errors shown in Table 3, four involved corrupted heartbeat messages that caused the FTM to fail. Although faulty data escaped the Heartbeat ARMOR, the corrupted message did not compromise the FTM's checkpoint. Thus, the FTM was able to recover from these four failures.

2.5 Heap Injections

Careful examination of the register injection experiments showed that crash failures were most often caused by segmentation faults raised from dereferencing a corrupted pointer. To maximize the chances for error propagation, only data (not pointers) were injected on the heap. Results from targeted injections into FTM heap memory were grouped by the element into which the error was injected. Table 4 shows the number of system failures observed from 100 error injections per element, classified as to the their effect on the system. One hundred targeted injections were sufficient to observe either escaped or detected errors given the amount of state in each element; overall, 500 heap injections were conducted on the FTM.

Table 4. System Failures Observed Through Heap Injections

Legend (Effect on system): (A) unable to register daemons, (B) unable to install Execution ARMORs, (C) unable to start applications, (D) unable to uninstall Execution ARMORs after application completes.

Legend (System failure/assertion check classification): (2) system failure without assertion firing, (3) system failure with assertion firing, (4) successful recoveries after assertion fired.

Element	Effect on System				System Failures			#4
	A	B	C	D	Total	#2	#3	
mgr_armor_info. Stores information about subordinate ARMORs such as location and element composition.	4	1	5	4	14	6	8	19
exec_armor_info. Stores information about each Execution ARMOR such as status of subordinate application.	0	0	5	4	9	4	5	9
app_param. Stores information about application such as executable name, command-line arguments, and number of times application restarted.	0	0	0	0	0	0	0	2
agr_app_detect. Used to detect that all processes for MPI application have terminated and to initiate recovery if necessary.	0	0	0	0	0	0	0	4
node_mgmt. Stores information about the nodes, including the resident daemon and hostname.	0	14	0	0	14	0	14	3
TOTAL	4	15	10	8	37	10	27	37

Many data errors were detectable through assertions within the FTM, but not all assertions were effective in preventing system failures. One of four scenarios resulted after a data error was injected (the last three columns in Table 4 are numbered to refer to scenarios 2-4):

1. The data error was not detected by an assertion and had no effect on the system. The application completed successfully as if there were no error.
2. The data error was not detected by an assertion but led to a system failure. None of the system failures impacted the application while it was executing.
3. The data error was detected by an assertion check, but only after the error had propagated to the FTM's checkpoint or to another process. Rolling back the FTM's state in these circumstances was ineffective, and system failures resulted from which the SIFT environment could not recover. These cases show that error latency is a factor when attempting to recover from errors in a distributed environment.
4. The data error was detected by an assertion check before propagating to the FTM's checkpoint or to another process. After an assertion fired, the FTM killed itself and recovered as if it had experienced an ordinary crash failure.

The injection results in Table 4 show that the least sensitive elements (*app_param* and *mgr_app_detect*) were those modules whose state was substantially read-only after being written early within the run. With assertions in place, none of the data errors led to system failures. At the other end of the sensitivity spectrum, 28 errors in two elements caused system failures. In contrast with the elements causing no system failures, the data in *mgr_armor_info* and *node_mgmt* were repeatedly written during the initialization phases of a run.

Table 4 also shows the efficiency of assertion checks in preventing system failures. The rightmost two columns in the table represent the total number of runs in which assertions detected errors. For example, assertions in the *mgr_armor_info* element detected 27 errors, and 19 of those errors were successfully recovered. The data also show that assertions coupled with the incremental microcheckpointing were able to prevent system failures in 58% of the cases (27 of 64 runs in which assertions fired).

On the other hand, assertions detected the error too late to prevent system failures in 27 cases. For example, 14 of the 17 runs in which assertions detected errors in the node_mgmt element resulted in system failures. This problem was rectified by adding checks to the translation results before sending the message.

2.6 Lessons Learned

SIFT overhead should be kept small. System designers must be aware that SIFT solutions have the potential to degrade the performance and even the dependability of the applications they are intended to protect. Our experiments show that the functionality in SIFT can be distributed among several processes throughout the network so that the overhead imposed by the SIFT processes is insignificant while the application is running.

SIFT recovery time should be kept small. Minimizing the SIFT process recovery time is desirable from two standpoints: (1) recovering SIFT processes have the potential to affect application performance by contending for processor and network resources, and (2) applications requiring support from the SIFT environment are affected when SIFT processes become unavailable. Our results indicate that fully recovering a SIFT process takes approximately 0.5 s. The mean overhead as seen by the application from SIFT recovery is less than 5%, which takes into account 10 out of roughly 800 failures from register, text-segment and heap injections that caused the application to block or restart because of the unavailability of a SIFT process. The overhead from recovery is insignificant when these 10 cases are neglected.

SIFT/application interface should be kept simple. In any multiprocess SIFT design, some SIFT processes must be coupled to the application in order to provide error detection and recovery. The Execution ARMORs play this role in our SIFT environment. Because of this dependency, it is important to make the Execution ARMORs as simple as possible. All recovery actions and those operations that affect the global system (e.g., job submission and detecting remote node failures) are delegated to a remote SIFT process that is decoupled from the application's execution. This strategy appears to work, as only 5 of 373 observed Execution ARMOR failures led to system failures.

SIFT availability impacts the application. Low recovery time and aggressive checkpointing of the SIFT processes help minimize the SIFT environment downtime, making the environment available for processing application requests and for recovering from application failures.

System failures are not necessarily fatal. Only 11 of the 10,000 injections resulted in a system failure in which the SIFT environment could not recover from the error. These system failures did not affect an executing application.

3 Error and Failure Analysis of a LAN of Windows NT-Based Servers

Direct monitoring, recording, and analysis of naturally occurring errors and failures in the system can provide valuable information on actual error/failure behavior, identify system bottlenecks, quantify dependability measures, and verify assumptions made in analytical models. In this section we provide an example of system dependability analysis using failure data collected from a Local Area Networks (LAN) of Windows NT servers.

In most commercial systems, information about failures can be obtained from the manual logs maintained by administrators or from the automated event-logging mechanisms in the underlying operating system. Manual logs are very subjective and often unavailable. Hence they are not typically suited for automated analysis of failures. In contrast, the event logs maintained by the system have predefined formats, provide contextual information in case of failures (e.g., a trace of significant events that precede a failure), and are thus conducive to automated analysis. Moreover, as failures are relatively rare events, it is necessary to meticulously collect and analyze error data for many machine-months for the results of the data analysis to be statistically valid. Such regular and prolonged data acquisition is possible only through automated event logging. Hence most studies of failures in single and networked computer systems are based on the error logs maintained by the operating system running on those machines.

This section presents methodology and results from an analysis of failures found in a network of about 70 Windows NT based mail servers (running Microsoft Exchange software). The data for the study is obtained from event logs (i.e., logs of machine events that are maintained and modified by the Windows NT operating system) collected over a six-month period from the mail routing network of a commercial organization. In this study we analyze only machine reboots because they constitute a significant portion of all logged failure data and are the most severe type of failure. As a starting point, a preliminary data analysis is conducted to classify the nature of observed failure events. This failure categorization is then used to examine the behavior of individual machines in detail and to derive a finite state model. The model depicts the behavior of a typical machine. Finally, a domain-wide analysis is performed to capture the behavior of the domain in a finite state model. The thorough failure data analysis, the reader can find in [12].

Related Work. Analysis of failures in computer systems has been the focus of active research for quite some time. Studies of failures occurring in commercial systems (e.g., VAX/VMS, Tandem/GUARDIAN) are based primarily on failure data collected from the field. The focus of such studies is on categorizing the nature of failures in the systems (e.g., software failures, hardware failures), identifying availability bottlenecks, and obtaining models to estimate the availability of the systems being analyzed. Lee [15], [16] analyzed failures in Tandem's GUARDIAN operating system. Tang [25] analyzed error logs pertaining to a multicomputer environment based on VAX/VMS cluster. Thakur [27] presented an analysis of failures in the Tandem Nonstop-UX operating system.

Hsueh [9] explored errors and recovery in IBM's MVS operating system. Based on the error logs collected from MVS systems, a semi-Markov model of multiple errors

(i.e. errors that manifest themselves in multiple ways) was constructed to analyze system failure behavior. Measurement-based software reliability models were also presented in [15], [16] (for the GUARDIAN system) and [25], [26] (for the VAX cluster).

The impact of workload on system failures was also extensively studied. Castillo [6] developed a software reliability prediction model that took into account the workload imposed on the system. Iyer [11] examined the effect of workload on the reliability of the IBM 3081 operating system. Mourad [21] performed a reliability study on the IBM MVS/XA operating system and found that the error distribution is heavily dependent on the type of system utilization. Meyer [20] presented an analysis of the influence of workload on the dependability of computer systems.

Lin [17] and Tsao [28] focused on trend analysis in error logs. Gray [8] presented results from a census of Tandem systems. Chillarege [7] presented a study of the impact of failures on customers and the fault lifetimes. Sullivan [23], [24] examined software defects occurring in operating systems and databases (based on field data). Velardi [29] examined failures and recovery in the MVS operating system.

3.1 Error Logging in Windows NT

Windows NT operating system offers capabilities for error logging. This software records information on errors occurring in the various subsystems, such as memory, disk, and network subsystems, as well as other system events, such as reboots and shutdowns. The reports usually include information on the location, time, type of the error, the system state at the time of the error, and sometimes error recovery (e.g., retry) information. The main advantage of on-line automatic logging is its ability to record a large amount of information about transient errors and to provide details of automatic error recovery processes, which cannot be done manually. Disadvantages are that an on-line log does not usually include information about the cause and propagation of the error or about off-line diagnosis. Also, under some crash scenarios, the system may fail too quickly for any error messages to be recorded.

An important question to be asked here is: How accurate are event logs in characterizing failure behavior of the system? While event logs provide valuable insight into understanding the nature and dynamics of typical problems observed in a network system, in many cases the information in event logs is not sufficient to precisely determine a nature of a problem (e.g., whether it was a software or hardware component failure). The only reliable way to improve accuracy of logs is (1) to perform more frequent, detailed logging by each component and (2) instrument the Windows NT code with new (more precise) logging mechanisms. However, there is always a trade-off between accuracy and intrusiveness of measurements. No commercial organization will permit someone to install an untested tool to monitor the network. Consequently, we use existing logs not only to characterize failure behavior of the network (presented in this paper), but also to determine how the logging system could be improved (e.g., by adding to the operating system a query mechanism to remotely probe system components about their status). It should be noted that in many commercial operating systems (e.g., MVS) event logs are accurate enough to document failures.

3.2 Classification of Data Collected from a LAN of Windows NT-Based Servers

The initial breakup of the data on a system reboot is primarily based on the events that preceded the current reboot by no more than an hour (and that occurred after the previous reboot). For each instance of a reboot, the most severe and frequently occurring events (hereafter referred to as prominent events) are identified. The corresponding reboot is then categorized based on the source and the id of these prominent events. In some cases, the prominent events are specific enough to identify the problem that caused the reboot. In other cases, only a high-level description of the problem can be obtained based on the knowledge of the prominent events. Table 5 shows the breakup of the reboots by category.

Hardware or firmware related problems: This category includes events that indicate a problem with hardware components (network adapter, disk, etc.), their associated drivers (typically drivers failing to load because of a problem with the device), or some firmware (e.g., some events indicated that the Power On Self Test had failed).

Connectivity problems: This category denotes events that indicated that either a system component (e.g., redirector, server) or a critical application (e.g., MS Exchange System Attendant) could not retrieve information from a remote machine. In these scenarios, it is not possible to pinpoint the actual cause of the connectivity problem. Some of the connectivity failures result from network adapter problems and hence are categorized as hardware related.

Table 5. Breakup of Reboots Based on Prominent Events

Category	Frequency	Percentage
Total reboots	1100	100
Hardware or firmware problems	105	9
Connectivity problems	241	22
Crucial application failures	152	14
Problems with a software component	42	4
Normal shutdowns	63	6
Normal reboots/power-off (no indication of any problems)	178	16
Unknown	319	29

Crucial application failure: This category encompasses reboots, which are preceded by severe problems with, and possibly shutdown of, critical application software (such as Message Transfer Agent). In such cases, it wasn't clear why the application reported problems. If an application shutdown occurs as a result of connectivity problem, then the corresponding reboot is categorized as connectivity-related.

Problems with a software component: Typically these reboots are characterized by startup problems (such as a critical system component not loading or a driver entry point not being found). Another significant type of problem in this category is the machine running out of virtual memory, possibly due to a memory leak in a software component. In many of these cases, the component causing the problem is not identifiable.

Normal shutdowns: This category covers reboots, which are not preceded by warnings or error messages. Additionally, there are events that indicate shutting down of critical application software and some system components (e.g., the BROWSER). These represent shutdowns for maintenance or for correcting problems not captured in the event logs.

Normal reboots/power-off: This category covers reboots which are typically not preceded by shutdown events, but do not appear to be caused by any problems either. No warnings or error messages appear in the event log before the reboot.

Based on data in Table 5, the following observations can be made about the failures:

1. 29% of the reboots cannot be categorized. Such reboots are indeed preceded by events of severity 2 or lesser, but there is not enough information available to decide (a) whether the events were severe enough to force a reboot of the machine or (b) the nature of the problem that the events reflect.
2. A significant percentage (22%) of the reboots have reported connectivity problems. Connectivity problems suggest that there could be propagated failures in the domain. Furthermore, it is possible that the machines functioning as the master browser and the Primary Domain Controller (PDC)[2], respectively are potential reliability bottlenecks of the domain.
3. Only a small percentage (10%) of the reboots can be traced to a system hardware component. Most of the identifiable problems are software related.
4. Nearly 50% of the reboots are *abnormal* reboots (i.e., the reboots were due to a problem with the machine rather than due to a normal shutdown).
5. In nearly 15% of the cases, severe problems with a crucial mail server application force a reboot of the machine.

3.3 Analysis of Failure Behavior of Individual Machines

After the preliminary investigation of the causes of failures, we probe failures from the perspective of an individual machine as well as the whole network. First we focus on the failure behavior of individual machines in the domain to obtain (1) estimates of machine up-times and down-times, (2) an estimate of the availability of each machine, and (3) a finite state model to describe the failure behavior of a typical machine in the domain. Machine up-times and down-times are estimated as follows:

- For every reboot event encountered, the timestamp of the reboot is recorded.
- The timestamp of the event immediately preceding the reboot is also recorded. (This would be the last event logged by the machine before it goes down.)
- A smoothing factor of one hour is applied to the reboots (i.e., for multiple reboots that occurred within an period of one hour, except the last one, are disregarded).
- Each up-time estimate is generated by calculating the time difference between a reboot timestamp and the timestamp of the event preceding the next reboot.

---

[2] In the analyzed network, the machines belonged to a common Windows NT domain. One of the machines was configured as the Primary Domain Controller (PDC). The rest of the machines functioned as Backup Domain Controllers (BDCs).

- Each down-time estimate is obtained by calculating the time difference between a reboot timestamp and the timestamp of the event preceding it.

Machine uptimes and machine downtimes are presented in Table 6. As the standard deviation suggests, there is a great degree of variation in the machine uptimes. The longest uptime was nearly three months. The average is skewed because of some of the longer uptimes. The median is more representative of the typical uptime.

Table 6. Machine Uptime & Downtime Statistics

Item	Machine Uptime Statistics	Machine Downtime Statistics
Number of entries	616	682
Maximum	85.2 days	15.76 days
Minimum	1 hour	1 second
Average	11.82 days	1.97 hours
Median	5.54 days	11.43 minutes
Standard Deviation	15.656 days	15.86 hours

As the table shows, 50% of the downtimes last about 12 minutes. This is probably too short a period to replace hardware components and reconfigure the machine. The implication is that majority of the problems are software related (memory leaks, misloaded drivers, application errors etc.). The maximum value is unrealistic and might have been due to the machine being temporarily taken off-line and put back in after a fortnight.

Since the machines under consideration are dedicated mail servers, bringing down one or more of them would potentially disrupt storage, forwarding, reception, and delivery of mail. The disruption can be prevented if explicit rerouting is per-formed to avoid the machines that are down. But it is not clear if such rerouting was done or can be done. In this context the following observations would be causes for concern: (1) average downtime measured was nearly 2 hours or (2) 50% of the measured uptime samples were about 5 days or less.

Availability

Having estimated machine uptime and downtime, we can estimate the availability of each machine. The availability is evaluated as the ratio:

[<average uptime> / (<average uptime> + <average downtime>)]*100

Table 7 summarizes the availability measurements. As the table depicts, the majority of the machines have an availability of 99.7% or higher. Also there is not a large variation among the individual values. This is surprising considering the rather large degree of variation in the average uptimes. It follows that machines with smaller average up-times also had correspondingly smaller average downtimes, so that the ratios are not very different. Hence, the domain has two types of machines: those that reboot often but recover quickly and those that stay up relatively longer but take longer to recover from a failure.

Table 7. Machine Availability

Item	Value
Number of machines	66
Maximum	99.99
Minimum	89.39
Median	99.76
Average	99.35
Standard Deviation	1.52

Fig. 3 shows the *unavailability distribution* across the machines (unavailability was evaluated as: *100 – Availability*). Less than 20% of the machines had an availability of 99.9% or higher. However, nearly 90% of the machines had an availability of 99% or higher. It should be noted that these numbers indicate the fraction of time the machine is alive. They do not necessarily indicate the ability of the machine to provide useful service because the machine could be alive but still unable to provide the service expected of it. To elaborate, each of the machines in the domain acts as a mail server for a set of user machines. Hence, if any of these mail servers has problems that prevent it from receiving, storing, forwarding, or delivering mail, then that server would effectively be unavailable to the user machines even though it is up and running. Hence, to obtain a better estimate of machine availability, it is necessary to examine how long the machine is actually able to provide service to user machines.

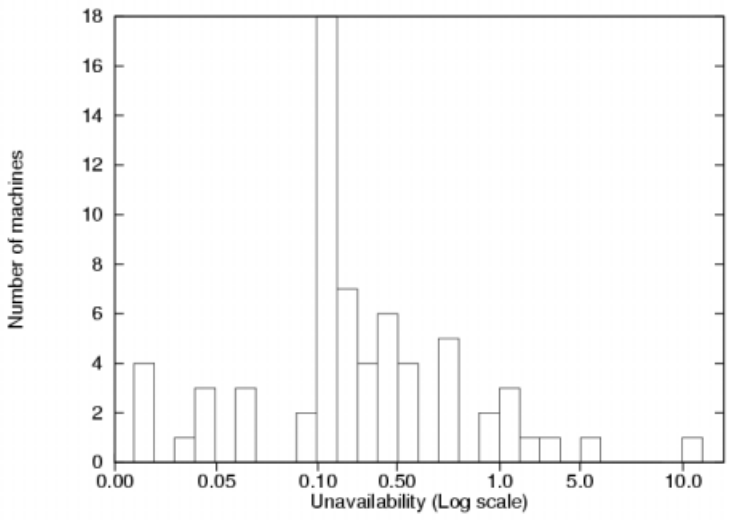

Fig. 3. Unavailability Distribution

Modeling Machine Behavior

To obtain more accurate estimates of machine availability, we modeled the behavior of a typical machine in terms of a *finite state model*. The model was based on the events that each machine logs. In the model, each state represents a level of functionality of the machine. A machine is either in a fully functional state, in which it logs events that indicate normal activity, or in a partially functional state, in which it logs events that indicate problems of a specific nature.

Selection and assignment of states to a machine was performed as follows. The logs were split into time-windows of one hour each. For each such window, the machine was assigned a state, which it occupied throughout the duration of the window. The assignment was based on the events that the machine logged in the window. Table 8 describes the states identified for the model.

Table 8. Machine States

State Name	Main Events (id/source/severity)	Explanation
Reboot	6005/EventLog/4	Machine logs reboot and other initialization events
Functional	5715/NETLOGON/4 1016/MSExchangeIS Private/8	Machine logs successful communication with PDC
Connectivity problems	3096/NETLOGON/1 5719/NETLOGON/1	Problems locating the PDC
Startup problems	7000/Service Control Manager/1 7001/Service Control Manager/1	Some system component or application failed to startup
MTA problems	2206/MSExchangeMTA/2 2207/MSExchangeMTA/2	Message Transfer Agent has problems with some internal databases
Adapter problems	4105/CpqNF3/1 4106/CpqNF3/1	The NetFlex Adapter driver reports problems
Temporary MTA problems	9322/MSExchangeMTA/4 9277/MSExchangeMTA/2 3175/MSExchangeMTA/2 1209/MSExchangeMTA/2	Message Transfer Agent reports problems of a temporary (or less severe) nature
Server problems	2006/Srv/1	Server component reports having received badly formatted requests
BROWSER problems	8021/BROWSER/2 8032/BROWSER/1	Browser reports inability to contact the master browser
Disk problems	11/Cpq32fs2/1 5/Cpq32fs2/1 9/Cpqarray/1 11/Cpqarray/1	Disk drivers report problems
Tape problems	15/dlttape/1	Tape driver reports problems
Snmpelea problems	3006/Snmpelea/1	Snmp event log agent reports error while reading an event log record
Shutdown	8033/BROWSER/4 1003/MSExchangeSA/4	Application/machine shutdown in progress

Each machine (except the Primary Domain Controller (PDC) whose transitions were different from the rest) in the domain was modeled in terms of the states mentioned in the table. A hypothetical machine was created by combining the transitions of all the individual machines and filtering out transitions that occurred less frequently. Fig. 4 describes this hypothetical machine. In the figure, the weight on each outgoing edge represents the fraction of all transitions from the originating state

(i.e., tail of the arrow) that end up in a given terminating state (i.e., head of the arrow). For example, if there is an edge from state A to state B with a weight of 0.5, then it would indicate that 50% of all transitions from state A are to state B. From Fig. 4 the following observations can be made:

- Only about 40 % of the transitions out of the *Reboot* states are to the *Functional* state. This indicates that in the majority of the cases, either the reboot is not able to solve the original problem, or it creates new ones.
- More than half of the transitions out of the *Startup problems* are to the *Connectivity problems* state. Thus, the majority of the startup problems are related to components that participate in network activity.
- Most of the problems that appear when the machine is functional are related to network activity. Problems with the disk and other components are less frequent.

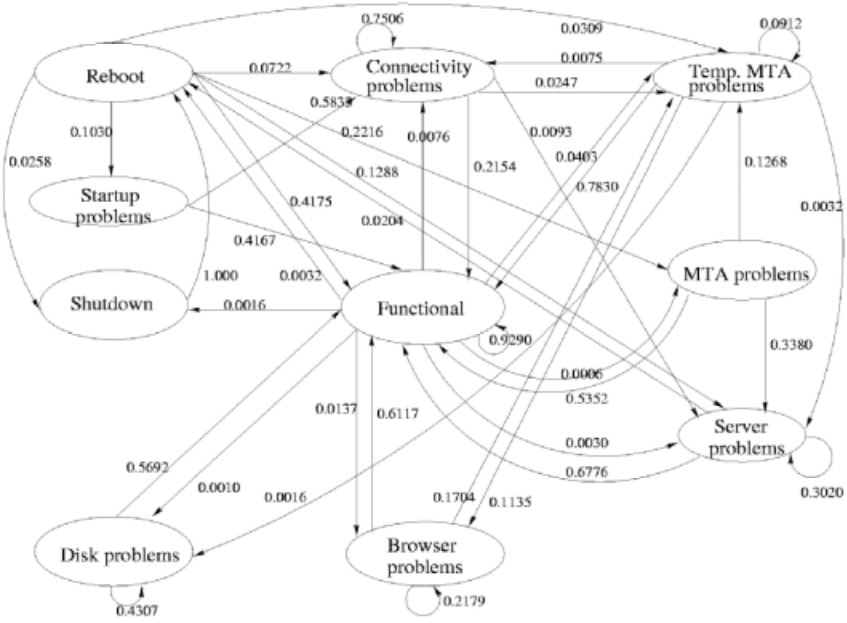

Fig. 4. State Transitions of a Typical Machine

- More than 50% of the transitions out of *Disk problems* state are to the *Functional* state. Also, we do not observe any significant transitions from the *Disk problems* state to other states. This could be due to one or more of the following:
  1. The machines are equipped with redundant disks so that even if one of them is down, the functionality is not disrupted in a major way.
  2. The disk problems, though persistent, are not severe enough to disrupt normal activity (maybe retries to access the disk succeed).
  3. The activities that are considered to be representative of the *Functional* state may not involve much disk activity.

- Over 11% of the transitions out of the *Temporary MTA problems* state are to the *Browser problems* state. We suspect that there was a local problem that caused RPCs to timeout or fail and caused problems for the MTA and BROWSER. Another possibility is that, in both cases, it was the same remote machine that could not be contacted. Based on the available data, it was not possible to determine the real cause of the problem.

To view the transitions from a different perspective, we computed the weight of each outgoing edge as a fraction of all the transitions in the finite state machine. Such a computation provided some interesting insights, which are enumerated below:

1. Nearly 10% of all the transitions are between the *Functional* and *Temporary MTA* problems states. These MTA problems are typically problems with some RPC calls (either failing or being canceled).
2. About 0.5% (1 in 200) of all transitions are to the *Reboot* state.
3. The majority of the transitions into the *MTA problems* state are from the *Reboot* state. Thus, MTA problems are primarily problems that occur at startup. In contrast, the majority of the transitions into the *Server problems* state and the *Browser problems* state (excluding the self loops) are from the *Functional* state. So, these problems (or at least a significant fraction of them) typically appear after the machine is functional.
4. About 92% of all transitions are into the *Functional* state. This figure is approximately a measure of the average time the hypothetical machine spends in the functional state. Hence it is a measure of the average availability of a typical machine. In this case, availability measures the ability of the machine to provide service, not just to stay alive.

3.4 Modeling Domain Behavior

Analyzing system behavior from the perspective of the whole domain (1) provides a macroscopic view of the system rather than a machine-specific view, (2) helps to characterize the nature of interactions in the network, and (3) aids in identifying potential reliability bottlenecks and suggests ways to improve resilience to operational faults.

Inter-reboot Times. An important characteristic of the domain is how often reboots occur within it. To examine this, the whole domain is treated as a black box, and every reboot of every machine in the domain is considered to be a reboot of the black box. Table 9 shows the statistics of such inter-reboot times measured across the whole domain.

Table 9. Inter-reboot Time Statistics for the Domain

Item	Value
Number of samples	882
Maximum	2.46 days
Minimum	Less than 1 second
Median	2402 seconds
Average	4.09 hours
Standard Deviation	7.52 hours

Finite State Model of the Domain

The proper functioning of the domain relies on the proper functioning of the PDC and its interactions with the Backup Domain Controllers (BDCs). Thus it would seem useful to represent the domain in terms of how many BDCs are alive at any given moment and also in terms of the PDC being functional or not. Accordingly, a finite state model was constructed as follows:

1. The data collection period was broken up into time windows of a fixed length,
2. For each such time window, the state of the domain was computed, and
3. A transition diagram was constructed based on the state information.

The state of the domain during a given time window was computed by evaluating the number of machines that rebooted during that time window. More specifically, the states were identified as shown in Table 10. Fig. 5 shows the transitions in the domain. Each time window was one hour long.

Table 10. Domain States and their Interpretation

State Name	Meaning
PDC	Primary Domain Controller (PDC) rebooted
BDC	1 Backup Domain Controller (BDC) rebooted
MBDC	Many BDCs rebooted
PDC+BDC	PDC and One BDC rebooted
PDC+MBDC	PDC and Many BDCs rebooted
F	Functional (no reboots observed)

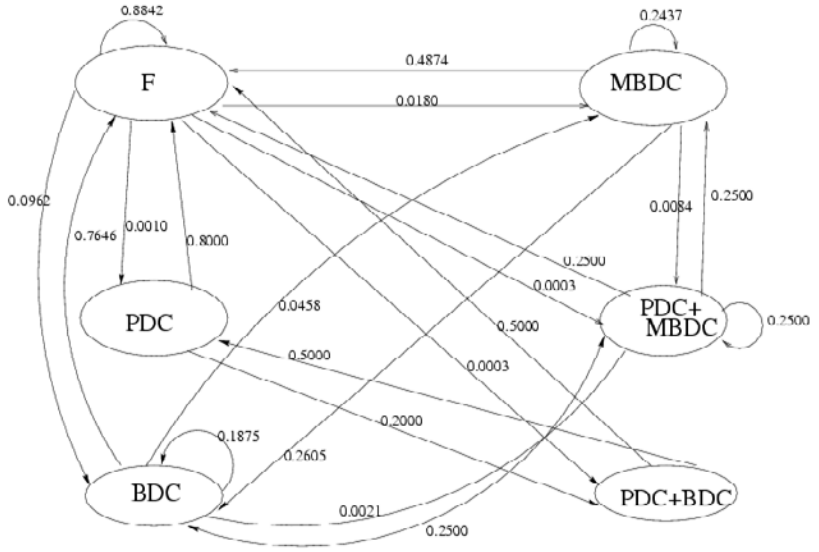

Fig. 5. Domain State Transitions

Fig. 5 reveals some interesting insights.

1. Nearly 77% of all transitions from the *F* state, excluding the self-loops, are to the *BDC* state. If these transitions do indeed result in disruption in service, then it is possible to improve the overall availability significantly just by tolerating single machine failures.
2. A non-negligible number of transitions are between the *F* state and the *MBDC* state and between states *BDC* and *MBDC*. This would indicate potentially correlated failures and recovery (see [12] for more details).
3. Majority of transitions from state *PDC* are to state *F*. This could be explained by one of the following:
   - Most of the problems with the *PDC* are not propagated to the BDCs,
   - The PDC typically recovers before any such propagation takes effect on the BDCs, or
   - The problems on the PDC are not severe enough to bring it down, but they might worsen as they propagate to the BDCs and force a reboot.

- However, 20% of the transitions from the *PDC* state are to the *PDC+BDC* state. So there is a possibility of the propagation of failures.

4 Conclusions

The discussion in this paper focused on the issues involved in analyzing the availability of networked systems using fault injection and the failure data collected by the logging mechanisms built into the system. To achieve accurate and comprehensive system dependability evaluation the analysis must span the three phases of system life: design phase, prototype phase, and operational phase.

For example the presented fault injection study of the ARMOR-based SIFT environment demonstrated that:

1. Structuring the fault injection experiments to progressively stress the error detection and recovery mechanisms is a useful approach to evaluating performance and error propagation.
2. Even though the probability for correlated failures is small, its potential impact on application availability is significant.
3. The SIFT environment successfully recovered from all correlated failures involving the application and a SIFT process because the processes performing error detection and recovery were decoupled from the failed processes.
4. Targeted injections into dynamic data on the heap were useful in further investigating system failures brought about by error propagation. Assertions within the SIFT processes were shown to reduce the number of system failures from data error propagation by up to 42%.

Similarly analysis of failure data collected in a network of Windows NT machines provides insights into network system failure behavior.

1. Most of the problems that lead to reboots are software related. Only 10% are attributable to specific hardware components.
2. Rebooting the machine does not appear to solve the problem in many cases. In about 60% of the reboots, the rebooted machine reported problems within a hour or two of the reboot.

3. Though the average availability evaluates to over 99%, a typical machine in the domain, on average, provides acceptable service only about 92% of the time.
4. About 1% of the reboots indicate memory leaks in the software.
5. There are indications of propagated or correlated failures. Typically, in such cases, multiple machines exhibit identical or similar problems at almost the same time.

Moreover, the failure data analysis also provides insights into the error logging mechanism. For example, event-logging features that are absent, but desirable, in Windows NT can be suggested:

1. The presence of a Windows NT shutdown event will improve the accuracy in identifying the causes of reboots. It will also lead to better estimates of machine availability.
2. Most of the events observed in the logs were either due to applications or to high-level system components, such as file-system drivers. It is not evident if this is due to a genuine absence of problems at the lower levels or it is just because the lower-level system components log events sparingly or resort to other means to report events. If the latter is true, then improved event logging by the lower-level system components (protocol drivers, memory managers) can enhance the value of event logs in diagnosis.

Acknowledgments. This manuscript is based on a research supported in part by NASA under grant NAG-1-613, in cooperation with the Illinois Computer Laboratory for Aerospace Systems and Software (ICLASS), by Tandem Computers, and in part by a NASA/JPL contract 961345, and by NSF grants CCR 00-86096 ITR and CCR 99-02026.

References

1. J.Arlat, et al., “Fault Injection for Dependability Validation – A Methodology and Some Applications,” *IEEE Trans. On Software Engineering*, Vol. 16, No. 2, pp. 166-182, Feb. 1990.
2. J.Arlat, et al., “Fault Injection and Dependability Evaluation of Fault-Tolerant Systems,” *IEEE Trans. On Computers*, Vol. 42, No. 8, pp.913-923, Aug. 1993.
3. D. Avresky, et al., “Fault Injection for the Formal Testing of Fault Tolerance,” *Proc. 22nd Int. Symp. Fault-Tolerant Computing*, pp. 345-354, June 1992.
4. S. Bagchi, “Hierarchical error detection in a software-implemented fault tolerance (SIFT) environment,” Ph.D. Thesis, University of Illinois, Urbana, IL, 2001.
5. J.H. Barton, E.W. Czeck, Z.Z. Segall, and D.P. Siewiorek, “Fault injection experiments using FIAT,” *IEEE Trans. Computers*, Vol.39, pp.575-582, Apr. 1990.
6. X. Castillo and D.P. Siewiorek, "A Workload Dependent Software Reliability Prediction Model," *Proc. 12th Int. Symp. Fault-Tolerant Computing*, pp.279-286, 1982.
7. R. Chillarege,S. Biyani, and J. Rosenthal, "Measurement Of Failure Rate in Widely Distributed Software," *Proc. 25th Int. Symp. Fault-Tolerant Computing,* pp. 424-433, 1995.
8. J. Gray, “A Census of Tandem System Availability between 1985 and 1990," *IEEE Trans. Reliability*,” Vol. 39, No. 4, pp. 409-418, 1990.

9. M.C. Hsueh, R.K. Iyer, and K.S. Trivedi, "Performability Modeling Based on Real Data: A Case Study," *IEEE Trans. Computers*, Vol. 37, No.4, pp. 478-484, April 1988.
10. R. Iyer, D. Tang, "Experimental Analysis of Computer System Dependability," *Chapter 5 in Fault Tolerant Computer Design*, D.K. Pradhan, Prentice Hall, pp.282-392, 1996.
11. R.K. Iyer and D.J. Rossetti, "Effect of System Workload on Operating System Reliability: A Study on IBM 3081," *IEEE Trans. Software Engineering*, Vol. SE-11, No. 12, pp. 1438-1448, 1985.
12. M. Kalyanakrishnam, "Failure Data Analysis of LAN of Windows NT Based Computers," *Proc. 18th Symp. on Reliable Distributed Systems*, pp.178-187, October 1999.
13. Z. Kalbarczyk, R. Iyer, S. Bagchi, K. Whisnant, "Chameleon: A software infrastructure for adaptive fault tolerance," *IEEE Trans. on Parallel and Distributed Systems*, vol. 10, no. 6, pp. 560-579, 1999.
14. G.A. Kanawati, N.A. Kanawati, and J.A. Abraham, "FERRARI: A flexible software-based fault and error injection system," *IEEE Trans. Computers*, Vol.44, pp.248-260, Feb. 1995.
15. I. Lee and R.K. Iyer, "Analysis of Software Halts in Tandem System," *Proc. 3rd Int. Symp. Software Reliability Engineering*, pp. 227-236, 1992.
16. I. Lee and R.K. Iyer, "Software Dependability in the Tandem GUARDIAN Operating System," *IEEE Trans. on Software Engineering*, Vol. 21, No. 5, pp. 455-467, 1995.
17. T.T. Lin, D.P. Siewiorek, "Error Log Analysis: Statistical Modeling and Heuristic Trend Analysis," *IEEE Trans. Reliability*, Vol. 39, No. 4, pp.419-432, 1990.
18. H. Maderia, R. Some, F. Moereira, D. Costa, D. Rennels, "Experimental evaluation of a COTS system for space applications," *Proc. Of Int. Conf. On Dependable Systems and Networks (DSN '02)*, Washington DC, pp. 325-330, June 2002.
19. Message Passing Interface Forum, "MPI-2: Extensions to the Message Passing Interface," *http://www.mpi-forum.org/docs/mpi-20.ps*.
20. J.F. Meyer and L. Wei, "Analysis of Workload Influence on Dependability" *Proc. 18th Int. Symp. Fault-Tolerant Computing*, pp.84-89, 1988.
21. S. Mourad and D. Andrews, "On the Reliability of the IBM MVS/XA Operating System," *IEEE Trans. on Software Engineering*, October 1987.
22. D. Stott, B. Floering, Z. Kalbarczyk, and R. Iyer, "Dependability assessment in distributed systems with lightweight fault injectors in NFTAPE," *Proc. Int. Performance and Dependability Symposium, IPDS-00*, pp. 91-100, 2000.
23. M.S. Sullivan, R. Chillarege,"Software Defects and Their Impact on System Availability — A Study of Field Failures in Operating Systems," *Proc. 21st Int. Symp. Fault-Tolerant Computing*, pp. 2-9, 1991.
24. M.S. Sullivan and R. Chillarege, "A Comparison of Software Defects in Database Management Systems and Operating Systems," *Proc. 22nd Int. Symp. Fault-Tolerant Computing*, pp.475-484, 1992.
25. D. Tang and R.K. Iyer, "Analysis of the VAX/VMS Error Logs in Multicomputer Environments — A Case Study of Software Dependability," *Proc. 3rd Int. Symp. Software Reliability Engineering*, Research Triangle Park, North Carolina, pp. 216-226, October 1992.
26. D. Tang and R.K. Iyer, "Dependability Measurement and Modeling of a Multicomputer Systems," *IEEE Trans. Computers*, Vol. 42, No. 1, pp.62-75, January 1993.
27. A.Thakur, R.K.Iyer, L. Young, I. Lee, "Analysis of Failures in the Tandem NonStop-UX Operating System," *Proc. Int'l Symp. Software Reliability Engineering*, pp. 40-49, 1995.
28. M.M. Tsao and D.P. Siewiorek, "Trend Analysis on System Error files," *Proc. 13th Int. Symp. Fault-Tolerant Computing*, pp. 116-119, June 1983.
29. P. Velardi and R.K. Iyer, "A Study of Software Failures and Recovery in the MVS Operating System"' *IEEE Trans. On Computers*, Vol. C-33, No. 6, pp.564-568, June 1984.
30. K. Whisnant, Z. Kalbarczyk, and R. Iyer, "Micro-checkpointing: Checkpointing for multithreaded applications," *in Proceedings of the 6th International On-Line Testing Workshop*, July 2000.

31. K. Whisnant, R. Iyer, Z. Kalbarczyk, P. Jones, "An Experimental Evaluation of the ARMOR-based REE Software-Implemented Fault Tolerance Environment," pending technical report, University of Illinois, Urbana, IL, 2001.
32. K. Whisnant, et al., "An Experimental Evaluation of the REE SIFTEnvironment for Spaceborne Applications," *Proc. Of Int. Conf. On Dependable Systems and Networks (DSN '02)*, Washington DC, pp. 585-594, June 2002.