11

Software Fault Tolerance in Computer Operating Systems

RAVISHANKAR K. IYER and INHWAN LEE
University of Illinois at Urbana-Champaign

ABSTRACT

This chapter provides data and analysis of the dependability and fault tolerance for three operating systems: the Tandem/GUARDIAN fault-tolerant system, the VAX/VMS distributed system, and the IBM/MVS system. Based on measurements from these systems, basic software error characteristics are investigated. Fault tolerance in operating systems resulting from the use of process pairs and recovery routines is evaluated. Two levels of models are developed to analyze error and recovery processes inside an operating system and interactions among multiple instances of an operating system running in a distributed environment.

The measurements show that the use of process pairs in Tandem systems, which was originally intended for tolerating hardware faults, allows the system to tolerate about 70% of defects in system software that result in processor failures. The loose coupling between processors which results in the backup execution (the processor state and the sequence of events occurring) being different from the original execution is a major reason for the measured software fault tolerance. The IBM/MVS system fault tolerance almost doubles when recovery routines are provided, in comparison to the case in which no recovery routines are available. However, even when recovery routines are provided, there is almost a 50% chance of system failure when critical system jobs are involved.

11.1 INTRODUCTION

The research presented in this chapter evolved from our previous studies on operating system dependability [Hsu87, Lee92, Lee93a, Lee93b, Tan92b, Vel84]. This chapter provides data and analysis of the dependability and fault tolerance of three operating systems: the Tandem/GUARDIAN fault-tolerant system, the VAX/VMS distributed system, and the

Software Fault Tolerance, Edited by Lyu

IBM/MVS system. A study of these three operating systems is interesting because they are widely used and represent the diversity in the field. The Tandem/GUARDIAN and VAX/VMS data provide high-level information on software fault tolerance. The MVS data provide detailed information on low-level error recovery. Our intuitive observation is that GUARDIAN and MVS have a variety of software fault tolerance features, while VMS has little explicit software fault tolerance.

Although an operating system is an indispensable software system, little work has been done on modeling and evaluation of the fault tolerance of operating systems. Major approaches for software fault tolerance rely on design diversity [Ran75, Avi84]. However, these approaches are usually inapplicable to large operating systems as a whole due to cost constraints. This chapter illustrates how a fault tolerance analysis of actual software systems, performing analogous functions but having different designs, can be performed based on actual measurements. The chapter provides the information of how software fault tolerance concepts are implemented in operating systems and how well current fault tolerance techniques work. It also brings out relevant design issues in improving the software fault tolerance in operating systems. The analysis performed illustrates how state-of-the-art mathematical methods can be applied for analyzing the fault tolerance of operating systems.

Ideally, we would like to measure different systems under identical conditions. The reality, however, is that differences in operating system architectures, instrumentation conditions, measurement periods, and operational environments make this ideal practically impossible. Hence, a direct and detailed comparison between the systems is inappropriate. It is, however, worthwhile to demonstrate the application of modeling and evaluation techniques using measurements on different systems. Also, these are mature operating systems that are slow-changing and have considerable common functionality. Thus, the major results can provide some high-level comparisons that point to the type and nature of relevant dependability issues.

Topics discussed include: 1) investigation of basic error characteristics such as software fault and error profile, time to error (TTE) and time to recovery (TTR) distributions, and error correlations; 2) evaluation of the fault tolerance of operating systems resulting from the use of process pairs and recovery routines; 3) low-level modeling of error detection and recovery in an operating system, illustrated using the IBM/MVS data, and 4) high-level modeling and evaluation of the loss of work in a distributed environment, illustrated using the Tandem/GUARDIAN and VAX/VMS data.

The next section introduces the related research. Section 11.3 explains the systems and measurements. Section 11.4 investigates software fault and error profile, TTE and TTR distributions, and correlated software failures. Section 11.5 evaluates the fault tolerance of operating systems. Section 11.6 builds two levels of models to describe software fault tolerance and performs reward analysis to evaluate software dependability. Section 11.7 concludes the chapter.

11.2 RELATED RESEARCH

Software errors in the development phase have been studied by researchers in the software engineering field [Mus87]. Software error data collected from the DOS/VS operating system during the testing phase was analyzed in [End75]. A wide-ranging analysis of software error data collected during the development phase was reported in [Tha78]. Relationships between the frequency and distribution of errors during software development, the maintenance of the

developed software, and a variety of environmental factors were analyzed in [Bas84]. An approach, called *orthogonal defect classification*, to use observed software defects to provide feedback on the development process was proposed in [Chi92]. These studies attempt to tune the software development process based on error analysis.

Software reliability modeling has been studied extensively, and a large number of models have been proposed (reviewed in [Goe85, Mus87]). However, modeling and evaluation of fault-tolerant software systems are not well understood, although several researchers have provided analytical models of fault-tolerant software. In [Lap84], an approximate model was derived to account for failures due to design faults; the model was also used to evaluate fault-tolerant software systems. In [Sco87], several reliability models were used to evaluate three software fault tolerance methods. Recently, more detailed dependability modeling and evaluation of two major software fault tolerance approaches—recovery blocks and N-version programming—were proposed in [Arl90].

Measurement-based analysis of the dependability of operational software has evolved over the past 15 years. An early study proposed a workload-dependent probabilistic model to predict software errors based on measurements from a DEC system [Cas81]. A study of failures and recovery of the MVS/SP operating system running on an IBM 3081 machine addressed the issue of hardware-related software errors [Iye85]. A recent analysis of data from the IBM/MVS system investigated software defects and their impact on system availability [Sul91]. A discussion of issues of software reliability in the system context, including the effect of hardware and management activities on software reliability and failure models, was presented in [Hec86]. Methodologies and advances in experimental analysis of computer system dependability over the past 15 years are reviewed in [Iye93].

11.3 MEASUREMENTS

For this study, measurements were made on three operating systems: the Tandem/GUARDIAN system, the VAX/VMS system, and the IBM/MVS system. Table 11.1 summarizes the measured systems. These systems are representative of the field in that they have varying degrees of fault tolerance embedded in the operating system. The following subsections introduce the three systems and measurements. Details of the measurements and data processing can be found in [Hsu87, Lee92, Lee93b, Tan92b, Vel84].

Table 11.1 Measured systems

HW/SW System	Architecture	Fault-Tolerance	Workload
Tandem/GUARDIAN	Distributed	Single-Failure Tolerance	1) Software Development 2) Customer Applications
IBM 3081/MVS	Single	Recovery Management	System Design/Development
VAXcluster/VMS	Distributed	Quorum Algorithm	1) Scientific Applications 2) Research Applications

11.3.1 Tandem/GUARDIAN

The Tandem/GUARDIAN system is a message-based multiprocessor system built for on-line transaction processing. High availability is achieved via single-failure tolerance. A critical system function or user application is replicated on two processors as the primary and backup processes, i.e., as process pairs. Normally, only the primary process provides service. The primary sends checkpoints to the backup so that the backup can take over the function on a failure of the primary. A software failure occurs when the GUARDIAN system software detects nonrecoverable errors and asserts a processor halt. The "I'm alive" message protocol allows the other processors to detect the halt and take over the primaries which were executing on the halted processor.

A class of faults and errors that cause software failures was collected. Two types of data were used: human-generated software failure reports (used in Section 11.4.1 and Section 11.5.1) and on-line processor halt logs (used in Section 11.4.2, Section 11.4.4, and Section 11.6.1). Human-generated software failure reports provide detailed information about the underlying faults, failure symptoms, and fixes. Processor halt logs provide near-100% of reporting and accurate timing information on software failures and recovery.

The source of human-generated software failure reports is the Tandem Product Report (TPR) database. A TPR is used to report all problems, questions, and requests for enhancements by customers or Tandem employees concerning any Tandem product. A TPR consists of a header and a body. The header provides fixed fields for information such as the date, customer and system identifications, and brief problem description. The body of a TPR is a textual description of all actions taken by Tandem analysts in diagnosing a problem. If a TPR reports a software failure, the body also includes the log of memory dump analyses performed by Tandem analysts. Two-hundred TPRs consisting of all reported software failures in all customer sites during a time period in 1991 were used.

The processor halt log is a subset of the Tandem Maintenance and Diagnostic System (TMDS) event log maintained by the GUARDIAN operating system. Measurements were made on five systems—one field system and four in-house systems—for a total of five system-years. Software failures are rare in the Tandem system, and only one of the in-house systems had enough software failures for a meaningful analysis. This system was a Tandem Cyclone system used by Tandem software developers for a wide range of design and development experiments. It was operating as a beta site and was configured with old hardware. As such, it is not representative of the Tandem system in the field. The measured period was 23 months.

11.3.2 IBM/MVS

The MVS is a widely used IBM operating system. Primary features of the system are reported to be efficient storage management and automatic software error recovery. The MVS system attempts to correct software errors using recovery routines. The philosophy in the MVS is that for each major system function, the programmer envisions possible failure scenarios and writes a recovery routine for each. It is, however, the responsibility of the installation (or the user) to write recovery routines for applications. The detection of an error is recorded by an operating system module.

Measurements were made on an IBM 3081 mainframe running the IBM/MVS operating system. The system consisted of dual processors with two multiplexed channel sets. Time-stamped, low-level error and recovery data on errors affecting the operating system functions

were collected. During the measurement period, the system was used primarily to provide a time-sharing environment to a group of engineering communities for their daily work on system design and development. Two measurements were made. The measurement periods were 14 months and 12 months. The source of the data was the on-line error log file produced by the IBM/MVS operating system.

11.3.3 VAX/VMS

A VAXcluster is a distributed computer system consisting of several VAX machines and mass storage controllers connected by the Computer Interconnect (CI) bus organized as a star topology [Kro86]. One of the VAXcluster design goals is to achieve high availability by integrating multiple machines in a single system. The operating system provides the cluster-wide sharing of resources (devices, files, and records) among users. It also coordinates the cluster members and handles recoverable failures in remote nodes via the Quorum algorithm.

Each operating system running in the VAXcluster has a parameter called VOTES and a parameter called QUORUM. If there are n machines in the system, each operating system usually sets its QUORUM to $\lfloor n/2 + 1 \rfloor$. The parameter VOTES is dynamically set to the number of machines currently alive in the VAXcluster. The processing of the VAXcluster proceeds only if VOTES is greater than or equal to QUORUM. Thus, the VAXcluster functions like an $\lfloor n/2 + 1 \rfloor$-out-of-n system.

The two measured VAXclusters had different configurations. The first system, VAX1, was located at the NASA Ames Research Center, a typical scientific application environment. It consisted of seven machines (four 11/785's, one 11/780, one 11/750, and one 8600) and four controllers. The data collection periods for the different machines in VAX1 varied from 8 to 10 months (from October 1987 through August 1988). The second system, VAX2, was located at the University of Illinois, an academic research and student application environment. It consisted of four machines (two 6410's, one 6310, and one 11/750) and one controller. The data collection period was 27 months (from January 1989 through March 1991). The source of the data was the on-line error log file produced by the VAX/VMS operating system.

11.4 BASIC ERROR CHARACTERISTICS

In this section, we investigate basic error characteristics using the measured data. These include fault and error profile, time to error (TTE) and time to recovery (TTR) distributions, and correlated software failures.

11.4.1 Fault and Error Classification

Collection of software faults and errors identified naturally reflect the characteristics of the software development environment. Many studies attempted to tune the software development process by analyzing the faults identified during the development phase [Tha78, End75, Bas84]. However, fault and error profiles of operational software can be quite different from those of the software during the development phase, due to the differences in the operational environment and software maturity. Therefore, it is necessary to investigate the fault and error profiles of operational software. Also, software fault and error categorization for the three measured operating systems is important because they are widely used operating systems. In

order to be of value to the community at large, such a knowledge should be accumulated in a public domain database that is regularly updated. Results of such categorization can then be used for testing and for designing efficient on-line error detection and recovery strategies as well as for fault avoidance.

11.4.1.1 GUARDIAN

We studied the underlying causes of 200 Tandem Product Reports (TPRs) consisting of all software failures reported by users for a time period in 1991 [Lee93b]. Twenty-one of the 200 TPRs were due to nonsoftware causes. Underlying causes of these failures indicate that hardware and operational faults sometimes cause failures that look as though they are due to software faults. Our experience shows that determining whether a failure is due to software faults is not always straightforward. This is partly because of the complexity of the system and partly because of close interactions between software and hardware platforms in the system. In 26 out of the remaining 179 TPRs, analysts believed that the underlying problems were software faults but had not yet located the faults. These are referred to as *unidentified* problems.

Table 11.2 shows the results of a fault classification using 153 TPRs whose software causes were identified. The table shows both the number of TPRs and the number of unique faults. Differences between the two represent multiple failures due to the same fault. The numbers inside parentheses show a further subdivision of a fault class.

Table 11.2 Software fault classification in GUARDIAN

Fault Class	#Faults	#TPRs
Incorrect computation	3	3
Data fault	12	21
Data definition fault	3	7
Missing operation:	20	27
– Uninitialized pointer	(6)	(7)
– Uninitialized nonpointer variable	(4)	(6)
– Not updating data structure on the occurrence of event	(6)	(9)
– Not telling other processes about the occurrence of event	(4)	(5)
Side effect of code update	4	5
Unexpected situation:	29	46
– Race/timing problem	(14)	(18)
– Errors with no defined error handling procedures	(4)	(8)
– Incorrect parameter or invalid call from user process	(3)	(7)
– Not providing routines to handle legitimate but rare operational scenarios	(8)	(13)
Microcode defect	4	8
Other (cause does not fit any of the above class)	10	12
Unable to classify due to insufficient information	15	24
All	100	153

Table 11.2 shows what kinds of faults the developers introduced. In the table, the faults

were ordered by the difficulty in testing and identifying them. "Incorrect computation" means an arithmetic overflow or the use of an incorrect arithmetic function (e.g., use of a signed arithmetic function instead of an unsigned one). "Data fault" means the use of an incorrect constant or variable. "Data definition fault" means a fault in declaring data or in defining a data structure. "Missing operation" means that a few lines of source code were omitted. A "side effect" occurs when not all dependencies between software components are considered when updating software. "Unexpected situation" refers to cases in which software designers did not anticipate a potential operational situation and the software does not handle the situation correctly. Table 11.2 shows that "missing operation" and "unexpected situation" are the most common causes of TPRs. Additional code inspection and testing efforts can be directed to such faults.

A high proportion of simple faults, such as incorrect computations or missing operations, is usually observed in new software, while a high proportion of complex causes, such as unexpected situations, is usually observed in mature software. The coexistence of a significant number of simple and complex faults is not surprising, because the measured system is a large software system consisting of both new and mature components. Further, some customer sites run earlier versions of software, while other sites run later versions. Yet one would like to see fewer simple faults. The existence of a significant proportion of simple faults indicates that there is room for improving the code inspection and testing process.

A software failure due to a newly found fault is referred to as a *first occurrence*, and a software failure due to a previously-reported fault is referred to as a *recurrence*. Out of the 153 TPRs whose underlying software faults were identified, 100 were due to unique faults. Out of the 100 unique faults, 57 were diagnosed before our measurement period. Therefore, 43 new software faults were identified during the measurement period. That is, about 72% (110 out of 153) of the software failures were recurrences of previously-reported faults. Considering that a quick succession of failures at a site, failures likely to be due to the same fault, is typically reported in a single TPR, the actual percentage of recurrences can be higher. This shows that, in environments where a large number of users run the same software, software development is not the only factor that determines the quality of software. Recurrences can seriously degrade software dependability in the field. Clearly, the impact of recurrences on system dependability needs to be modeled and evaluated.

11.4.1.2 MVS

In MVS, software error data, such as the type of error detection (hardware and software), error symptom, severity, and the results of hardware and software attempts to recover from the problem, are logged by the system. The error symptoms provided by the system were grouped into classes of similar errors. The error classes were chosen to reflect commonly encountered problems. Six classes of errors were defined [Vel84]:

1. Control: indicates the invalid use of control statements and invalid supervisor calls.
2. I/O and data management: indicates a problem occurred during I/O management or during the creation and processing of data sets.
3. Storage management: indicates an error in the storage allocation/deallocation process or in virtual memory mapping.
4. Storage exceptions: indicates addressing of nonexistent or inaccessible memory locations.
5. Programming exceptions: indicates a program error other than a storage exception.

6. Timing: indicates a system or operator-detected endless loop, endless wait state, or violation of system or user-defined time limits.

Table 11.3 shows the percentage distribution of the errors during the measured period. On the average, the three major error classes are storage management (40%), storage exceptions (21%), and I/O and data management (19%). This result is probably related to the fact that a major feature of MVS is the multiple virtual storage organization. Storage management and I/O and data management are high-volume activities critical to the proper operation of the system. Therefore, one might expect their contributions to errors to be significant.

Table 11.3 Software error classification in MVS (measurement period: 14 months)

Error Type	Frequency	Fraction (%)
Control	22	5.5
Timing	29	7.3
I/O and Data Management	74	18.5
Storage Management	161	40.4
Storage Exceptions	82	20.6
Programming Exceptions	31	7.8
All	399	100.0

11.4.1.3 VMS

Software errors in a VAXcluster system are identified from "bugcheck" reports in the error log files. All software detected errors were extracted from bugcheck reports and divided into four types in [Tan92c]:

1. Control: problems involving program flow control or synchronization, for example, "Unexpected system service exception," "Exception while above ASTDEL (Asynchronous System Traps DELivery) or on interrupt stack," and "Spinlock(s) of higher rank already owned by CPU."
2. Memory: problems referring to memory management or usage, for example, "Bad memory deallocation request size or address," "Double deallocation of memory block," "Pagefault with IPL (Interrupt Priority Level) too high," and "Kernel stack not valid."
3. I/O: inconsistent conditions detected by I/O management routines, for example, "Inconsistent I/O data base," "RMS (Record Management Service) has detected an invalid condition," "Fatal error detected by VAX port driver," "Invalid lock identification," and "Insufficient nonpaged pool to remaster locks on this system."
4. Others: other software-detected problems, for example, "Machine check while in kernel mode," "Asynchronous write memory failure," and "Software state not saved during powerfail."

Table 11.4 shows the frequency for each type of software-detected error for the two VAXcluster systems. Nearly 13% of software-detected errors are type "Others," and almost all of them belong to VAX2. The VAX2 data showed that most of these errors were "machine check" (i.e., CPU errors). It seemed that the VAX1 error logs did not include CPU errors in

the bugcheck category. A careful study of the VAX error logs and discussions with field engineers indicate that different VAX machine models may report the same type of error (in this case, CPU error) to different classes. Thus, it is necessary to distinguish these errors in the error classification. Most "Others" errors were judged to be nonsoftware problems.

Table 11.4 Software error classification in VMS (Measurement period: 10 months for VAX1 and 27 months for VAX2)

Error Type	Frequency (VAX1)	Frequency (VAX2)	Fraction (%), Combined
Control	71	26	50.0
Memory	8	4	6.2
I/O	16	44	30.9
Others	1	24	12.9
All	96	98	100.0

11.4.2 Error Distributions

Time to error (TTE) and time to failure (TTF) distributions provide the information on error and failure arrivals. Figure 11.1 shows the empirical TTE or TTF distributions fitted to analytic functions for the three measured systems. Here, a failure means a processor failure, not a system failure. An error is defined as a nonstandard condition detected by the system software. Due to the differences in semantics and logging mechanisms between the measured systems, a direct comparison of the distributions is not possible. But we can make high level observations that point to relevant dependability issues.

None of the distributions in Figure 11.1 fit simple exponential functions. The fitting was tested using the Kolmogorov-Smirnov or Chi-square test at the 5% significance level. This result conforms to the previous measurements on IBM [Iye85] and DEC [Cas81] machines. Several reasons for this nonexponential behavior, including the impact of workload, were documented in [Cas81].

The two-phase hyperexponential distribution provided satisfactory fits for the VAXcluster software TTE and Tandem software TTF distributions. An attempt to fit the MVS TTE distribution to a phase-type exponential distribution led to a large number of stages. As a result, the following multistage gamma distribution was used:

$$f(t) = \sum_{i=1}^{n} a_i g(t; \alpha_i, s_i) \tag{11.1}$$

where $a_i \geq 0$, $\sum_{i=1}^{n} a_i = 1$, and

$$g(t; \alpha, s) = \begin{cases} 0 & t < s, \\ \frac{1}{\Gamma(\alpha)}(t-s)^{\alpha-1} e^{-(t-s)} & t \geq s. \end{cases} \tag{11.2}$$

It was found that a 5-stage gamma distribution provided a satisfactory fit.

Figure 11.1b and Figure 11.1c show that the measured software TTE and TTF distributions can be modeled as a probabilistic combination of two exponential random variables, indicating

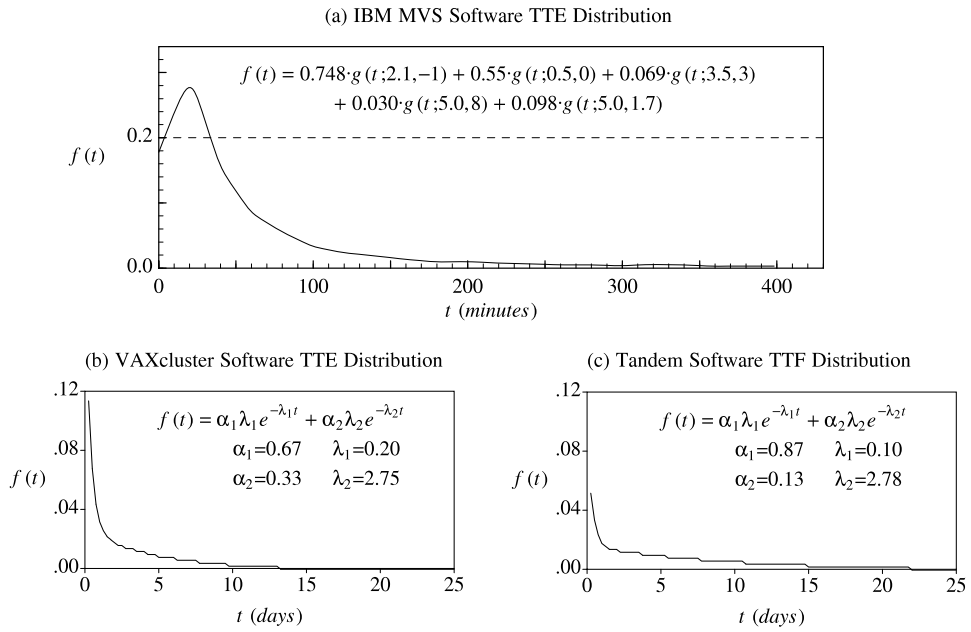


Figure 11.1 Empirical software TTE/TTF distributions

that there are two dominant error modes. The higher error rate, λ_2, with occurrence probability α_2, captures both the error bursts (multiple errors occurring on the same operating system within a short period of time) and concurrent errors (multiple errors on different instances of an operating system within a short period of time) on these systems. The lower error rate, λ_1, with occurrence probability α_1, captures regular errors and provides an interburst error rate. Error bursts are also significant in MVS. They are not clearly shown in Figure 11.1a because each error burst was treated as a single situation, called a *multiple error*. (The characteristics of multiple errors and their significance are discussed in Section 11.6.2.)

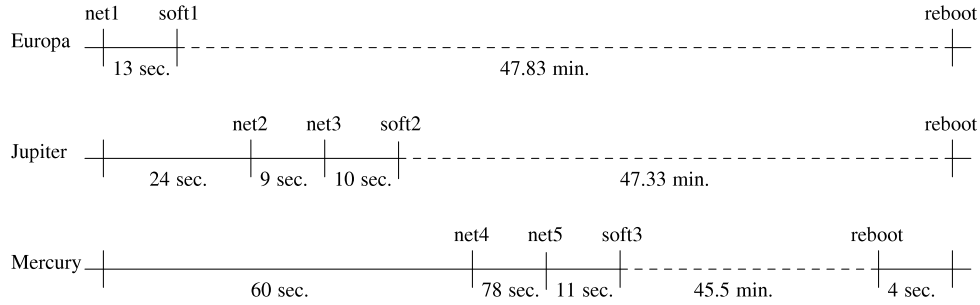
The above results show that error bursts need to be taken into account in the system design and modeling. The inclusion of error bursts in a model can cause a stiffness problem which may require improved solution methods. Error bursts, which are near-coincident problems, can affect recovery/retry techniques because additional errors can hit the system while it is recovering from the first error. Hence design tradeoffs between performing a rapid recovery and a full-scale power-on-self-test (POST) need to be investigated.

Error bursts can also be repeated occurrences of the same software problem or multiple effects of an intermittent hardware fault on the software. Software error bursts have been observed in laboratory experiments reported in [Bis88]. This study showed that, if the input sequences of the software under investigation are correlated (rather than being independent), one can expect more "bunching" of failures than those predicted using a constant failure rate assumption. In an operating system, input sequences (user requests) are highly likely to be correlated. Hence, a defect area can be triggered repeatedly.

11.4.3 Correlated Software Failures

When multiple instances of an operating system interact in a multicomputer environment, the issue of correlated failures should be addressed. The data showed that about 10% of software failures in the VAXcluster and 20% of software failures in the Tandem system occurred on multiple machines concurrently. To understand these concurrent software failures on different machines, it is instructive to examine a real case of correlated failures in detail.

Figure 11.2 shows a scenario of correlated software failures. In the figure, Europa, Jupiter, and Mercury are machine names in the VAXcluster. A dashed line represents that the corresponding machine is in a failure state. At one time, a network error (net1) was reported from the CI (Computer Interconnect) port on Europa. This resulted in a software failure (soft1) 13 seconds later. Twenty-four seconds after the first network error (net1), additional network errors (net2,net3) were reported on the second machine (Jupiter), which was followed by a software failure (soft2). The error sequence on Jupiter was repeated (net4,net5,soft3) on the third machine (Mercury). The three machines experienced software failures concurrently for 45.5 minutes. All three software failures occurred shortly after network errors occurred, so they are network error related. Further analysis of the data revealed that the network-related software of the VAX/VMS is a potential software bottleneck in terms of correlated failures.



Note: soft1, soft2, soft3 — Exception while above asynchronous system traps delivery or on interrupt stack.
net1, net3, net5 — Port will be re-started. net2, net4 — Virtual circuit timeout.

Figure 11.2 A scenario of correlated software failures

The higher percentage of correlated software failures in the Tandem system is attributed to the architectural characteristics of the Tandem system. In the Tandem system, it is possible that a single software fault causes halts of two processors on which the primary and backup processes of the faulty software are executing. If the two halted processors control a disk that includes files needed by other processors on the system, additional software halts can occur on these processors. (In the Tandem system, a disk can typically be accessed by two processors via dual-port disk controllers.) This explains why there is a higher percentage of correlated software failures in the Tandem system.

Note that the above scenario is a multiple component failure situation. A substantial amount of efforts has been directed at developing general system design principles against correlated failures. Still, correlated failures exist due to design holes and unmodeled faults. Generally, correlated failures can stress recovery and break the protection provided by the fault tolerance.

It has been shown that even a low percentage of correlated failures can have a big impact on system dependability [Dug92, Tan92a]. Thus, correlated failures cannot be neglected.

11.4.4 Recovery Distributions

In MVS, time to recovery (TTR) is defined as the time difference between the operating system's departure from the normal state due to the detection of an error and its subsequent return to the normal state. The normal state means that no error recovery is pending, i.e., all previously-detected errors are resolved by the operating system. In GUARDIAN, TTR is defined as the time difference between a processor's going down due to software and its coming back on-line. VMS shares the definition of TTR with MVS, but the TTR data from VMS look closer to the GUARDIAN TTR data than the MVS TTR data. This is probably because VMS logs only serious software errors. About 80% of software errors logged resulted in a node failure in the measured VAXcluster systems.

Since each system has different recovery procedure and maintenance environment, it is inappropriate to compare the measured systems in terms of TTR distribution. Our intention is to understand and discuss the different recovery mechanisms and resulting recovery time characteristics in the three operating systems.

Figure 11.3a plots a spline-fit for the TTR distribution of multiple software errors in the MVS system. A multiple software error is an error burst consisting of different types of software errors. The TTR distribution for multiple software errors was studied because these errors have longer recovery times than other software errors and are more typical in terms of recovery process (see Table 11.13 in Section 11.6). Our analysis found that a three-phase hyperexponential function can be used to approximate the distribution, suggesting a multimode recovery process. Because most MVS software errors do not lead to system failures, the TTR for multiple errors is short, although these errors take the longest time to recover of all software errors.

Figure 11.3b and Figure 11.3c plot the empirical software TTR distributions for the VAXcluster and Tandem systems. Because of their peculiar shapes, the raw distributions are provided. In the VAXcluster (Figure 11.3b), most of the TTR instances (85%) are less than 15 minutes. This is attributed to those errors recovered by on-line recovery or automatic reboot without shutdown repair. However, some TTR instances last as long as several hours (the maximum is about 6.6 hours). These failures are, in our experience, probably due to a combination of software and hardware problems. Since the Tandem system does not allow an automatic recovery from a halt, its TTR distribution (Figure 11.3c) reflects the time to reload and restart by the operator.

Typically, analytical models assume exponential or constant recovery times. Our results show that this does not apply universally. None of the three TTR distributions is a simple exponential. For the MVS system, since the recovery is usually quick, a constant recovery time assumption may be suitable. For the VAXcluster and Tandem systems, neither exponential nor constant recovery time can be assumed. More complex multimode functions are needed to model these TTR distributions.

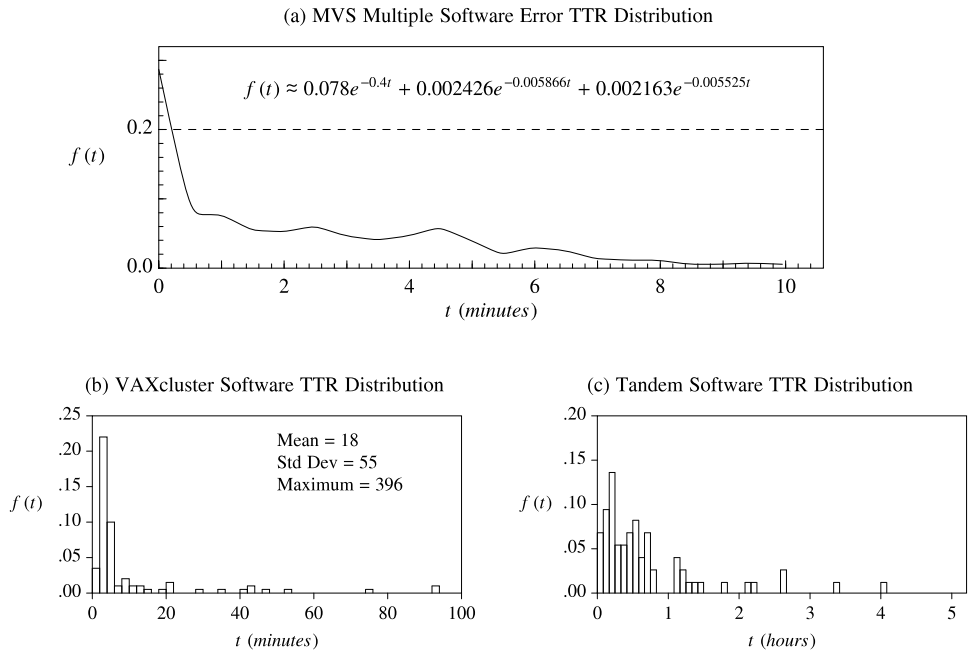


Figure 11.3 Empirical software TTR distributions

11.5 EVALUATION OF FAULT TOLERANCE

This section discusses the evaluation of the software fault tolerance achieved by the use of 1) process pairs in the Tandem/GUARDIAN operating system [Lee93b] and 2) recovery routines in the IBM/MVS operating system [Vel84]. Process pairs are an implementation of checkpointing and restart, which is a general approach developed in the context of distributed data management. Recovery routines are an implementation of exception handling.

The evaluation of process pairs focuses on the faults in the system software that cause processor failures. The evaluation of recovery routines focuses on software errors occurred in the system software. Clearly, the two evaluations cover different software fault spaces. As such, the results in this section should be regarded as what we can achieve by using process pairs and recovery routines, not as a comparison between the two techniques. Recovery routines and process pairs are techniques that can be used together.

While the Tandem/GUARDIAN and VAX/VMS operating systems rely on recovery routines to a certain degree, the data from these systems did not allow us to evaluate their effectiveness against software faults and errors. Also, the IBM/MVS and VAX/VMS operating systems do not have constructs for checkpointing and restart.

11.5.1 Evaluation of Process Pairs

It has been observed that process pairs allow the system to tolerate certain software faults [Gra85, Gra90]. That is, in many processor halts due to software faults, the backup of a failed primary can continue the execution. This is rather counter-intuitive because the primary and backup run the same copy of the software. The phenomenon was explained by the existence

of subtle faults, often referred to as "transient" software faults, that are not exercised again on a restart of the failed software. Field software faults were not detected during the testing phase, and many of them could be transient in nature. Since the technique is not explicitly intended for tolerating software faults, study of field data is essential for understanding the phenomenon and for measuring the effectiveness of the technique against software faults.

Using human-generated field software failure reports in the Tandem system, [Lee93b] measured 1) the user-perceived ability of the Tandem system to tolerate faults in its system software due to the use of process pairs and 2) the detailed reasons for software fault tolerance. Recently, attempts have been made to use the transient nature of some software faults for tolerating such faults in user applications using checkpoint and restart [Hua93, Wan93].

11.5.1.1 MEASURE OF SOFTWARE FAULT TOLERANCE

There were 179 TPRs generated due to software faults during the measured period (see Section 11.4.1). Since each TPR reports just one problem, sometimes two TPRs are generated as a result of a multiple processor halt. There were five such cases, making a total of 174 software failures during the measured period. Table 11.5 shows the severity of the 174 software failures. A single-processor halt implies that the built-in single-failure tolerance of the system masked the software fault that caused the halt. We aggregated all multiple processor halts into a single group because, in the Tandem system, a double processor halt can potentially cause additional processor halts due to the system architecture (see Section 11.4.3). There was one case in which a software failure occurred in the middle of a system coldload.

Table 11.5 Severity of software failures

Severity	# Failures
Single processor halt	138
Multiple processor halt	31
During system coldload	1
Unclear	4
All	174

Here the term software fault tolerance (SFT) refers to the ability to tolerate software faults. Quantitatively, it was defined as

$$SFT = \frac{\text{number of software failures in which a single processor is halted}}{\text{total number of software failures}}. \quad (11.3)$$

Thus, it represents the user-perceived ability of the system to tolerate faults in its system software specifically due to the use of process pairs.

Table 11.5 shows that process pairs can provide a significant level of software fault tolerance in distributed transaction processing environments. The measure of the software fault tolerance is 82% (138 out of 169). This measure is based on the reported software failures. The issue of underreporting was discussed in [Gra90]. The consensus among experienced Tandem engineers seems to be that about 80% of software failures do not get reported as TPRs and that most of them are single-processor halts. If that is true, then the software fault tolerance may be as high as 96%.

11.5.1.2 OUTAGES DUE TO SOFTWARE

We first focused on the multiple processor halts. For each multiple processor halt, we investigated the first two processor halts to determine whether the second halt occurred on the processor executing the backup of the failed primary process. In such cases, we also investigated whether the two processors halted due to the same software fault.

Table 11.6 shows that in 86% (24 out of 28, excluding "unclear" cases) of the multiple processor halts, the backup of the failed primary process was unable to continue the execution. In 81% (17 out of 21, excluding "unclear" cases) of these halts, the backup failed due to the same fault that caused the failure of the primary. In the remaining 19% of the halts, the processor executing the backup of the failed primary halted due to another fault during job takeover. While the level of software fault tolerance achieved with process pairs is high, it is not perfect. As a result, there is a chance that a single software fault in the system software can manifest itself as a multiple processor halt that the system is not designed against.

Table 11.6 Reasons for multiple processor halts

Reasons for Multiple Processor Halts	# Failures
The second halt occurs on the processor executing the backup of the failed primary	24
– The second halt occurs due to the same fault that halted the primary	(17)
– The second halt occurs due to another fault during job takeover	(4)
– Unclear	(3)
Not related to process pairs	4
– System hang	(1)
– Execution of faulty parallel software	(1)
– Random coincidence of two independent faults	(1)
– Single-processor halt, but system coldload was necessary for recovery	(1)
Unclear (insufficient information in TPR)	3

11.5.1.3 CHARACTERIZATION OF SOFTWARE FAULT TOLERANCE

The information in Table 11.5 poses the question of why the Tandem system loses only one processor in 82% of software failures and, as a result, tolerates the software faults that cause these failures. We identified the reasons for software fault tolerance in all single-processor halts and classified them into several groups. Table 11.7 shows that, in 29% of single-processor halts, the fault that causes a failure of a primary process is not exercised again when the backup reexecutes the same task after a takeover. This happens because some software faults are exposed on rare situations such as 1) a specific memory state (e.g., running out of buffer), 2) the occurrence of a single event or a sequence of asynchronous events during a vulnerable time window (timing), 3) race conditions or concurrent operations among multiple

processes, or 4) the occurrence of an error. These situations are usually not repeated on the backup.

Table 11.7 Reasons for software fault tolerance

Reasons for Software Fault Tolerance	Fraction (%)
Backup reexecutes the failed task after takeover, but the fault that caused a failure of primary is not exercised by backup	29
– Memory state	(4)
– Timing	(7)
– Race or concurrency	(6)
– Error	(4)
– Others	(7)
Backup, after takeover, does not automatically reexecute the failed task	20
Effect of error latency	5
Fault affects only backup	16
Unidentified problem	19
Unable to classify due to insufficient information	12

The following is a real example of a fault that is exercised only in a specific memory state. The primary of an I/O process pair requested a buffer to serve a request. Due to the high activity in the processor executing the primary, the buffer was not available. But, due to a fault, the buffer management routine returned a "successful" flag, instead of an "unsuccessful" flag. The primary used the uninitialized buffer pointer, thinking that it was a valid one, and a halt occurred in the processor running the primary. The backup took over and served the same request, but the fault was not exercised again because buffer was available in the processor running the backup.

Table 11.7 also shows that, in 20% of single-processor halts, the backup of a failed primary process does not serve the failed request after a successful takeover. This is because some faults are exposed while the primary is handling requests that are important but are not automatically resubmitted to the backup on a failure of the primary. Examples of such requests are operator commands to reconfigure I/O lines. These requests are not automatically resubmitted to the backup because they are interactive tasks that can easily be resubmitted by the operator if a failure occurs. Also, some software faults are exposed while executing system functions that are important but do not run as process pairs. An example of such system functions is a utility program for on-line processor performance monitoring. Consider that the response time on a processor increases. Then the operator would run such a utility program. In this situation, it is not imperative to run the utility as process pairs, because there is no need to monitor a processor any longer if the processor fails. If the monitoring processor fails, then the operator can run the utility on another processor. In this case, the failed task does not survive the failure. But process pairs allow the other applications on the halted processor to continue to run. This is not a strict software fault tolerance but a side benefit of using process pairs. If these failures are excluded, the measure of software fault tolerance becomes 77%.

Another reason for the software fault tolerance is that some software faults cause errors

that are detected after the service that caused the errors has finished successfully (see "effect of error latency" in Table 11.7). For example, a process overwrote words in a system data structure due to an uninitialized pointer when it served a request. The underlying fault was a missing operation, i.e., not initializing a pointer. The process that caused the error finished serving the request successfully, which was checkpointed successfully to its backup. (The error did not affect the service and was not a part of the checkpoint information.) After a while, another process in the same processor detected the error and asserted a halt. The backups of these processes continued the executions, but the service that caused the error was not repeated because the service was already provided. Differences between this case and the first group listed in Table 11.7 is that the software function that caused the failure of the primary did not have to be executed again in the backup.

Table 11.7 also shows that 16% of single-processor halts occur due to a failure of a backup process. This indicates that the software fault tolerance does not come free: the added complexity due to the implementation of process pairs introduces additional software faults in the system software. The measure of software fault tolerance (77%) estimated above can be adjusted again to 72% by excluding these failures. All unidentified failures were single-processor halts. This is understandable because these are due to subtle faults that are very hard to observe and diagnose. The reason why an unidentified problem caused a single-processor halt is unknown. Based on their symptoms, we speculate that a significant number of these were single-processor halts due to the effect of error latency.

11.5.1.4 DISCUSSION

The results in this section have several implications. First, process pairs are not explicitly designed for tolerating software faults. Our results show that a major reason for the measured software fault tolerance is the loose coupling between processors which results in the backup execution (the processor state and the sequence of events occurring) being different from the original execution. This confirms that there is another dimension for achieving software fault tolerance in distributed environments. The actual level of software fault tolerance achieved by the use of process pairs will depend on the level of differences between the original and backup executions. Each processor on a Tandem system has an independent processing environment, so the system naturally provides such differences. ([Gra85] discussed the advantages of using checkpointing, as opposed to lock-step operation, in terms of the ability to tolerate software faults.) The level of software fault tolerance achieved by the use of process pairs will also depend on the proportion of subtle faults in the software that are exercised only in rare processor states. While process pairs may not provide perfect software fault tolerance, the implementation of process pairs is not as prohibitively expensive as developing and maintaining multiple versions of large software programs.

Second, the results indicate that process pairs can also allow the system to tolerate nontransient software faults. This is because software failures can occur while executing important tasks that are not automatically resubmitted to the backup on a failure of the primary. In this case, the failed task does not survive, but process pairs allow the other applications on the failed processor to survive.

Third, it has been observed that short error latency with error confinement within a transaction is desirable [Cri82]. In actual designs, such a strict error confinement might be rather hard to achieve. In Tandem systems, the unit of error confinement is a processor, not a transaction [Gra85]. Errors generated by a transaction may be detected by another transaction. Interest-

ingly, the data show that long error latency, when combined with error propagation across transactions, sometimes helps the system to tolerate software faults. This result should not be interpreted as long error latency or error propagation is a desirable characteristic. Rather, it should be interpreted as a side effect of the system software containing subtle faults.

Finally, an interesting question is: if process pairs are good, are process triples better? Our results show that process triples may not necessarily be better, since faults that cause double processor halts with process pairs may cause triple processor halts with process triples.

11.5.2 Evaluation of Recovery Routines

[Vel84] evaluated the effectiveness of recovery routines using error logs collected from an IBM/MVS operating system. Using job names (at error occurrence) supplied by the system, three groups of job functions were defined: *critical* (for system survival), *essential* (would degrade but not crash the system), and *nonessential* (to system survival). Table 11.8 evaluates the effectiveness of recovery routines in dealing with these jobs. The table shows that retries occurred on 43% of errors involving critical jobs and for 68% on essential jobs. Importantly, in over 50% of the cases where system-critical jobs are involved, task termination results. The task is a module of the critical jobs, and usually system termination (recall that this is defined as a failure) results. Similar, although slightly improved, figures are found for essential jobs. This points toward an inadequacy in recovery management, since one would like better recovery and far fewer task terminations when critical and essential jobs are involved.

Table 11.8 Recovery management

Job Criticality and Type of Recovery				
	Retry %	Task Termination %	Job Termination %	Frequency
Critical	43.3	53.0	3.7	402
Essential	68.6	23.5	7.8	51
Nonessential	24.8	51.9	23.3	592

(a)

Effectiveness of Recovery Routines			
	Recovery Routines Provided %	Failures (Recovery Routines Provided) %	Failures (Recovery routines Not Provided) %
Critical	65.7	44.3	80.4
Essential	78.4	20.0	72.7

(b)

Table 11.8 also shows that recovery routines were specified in about 65% of the errors where critical jobs were involved. In interpreting this table, recall that recovery is possible even when no recovery routine is provided through the recovery termination manager. The percentage of failures in cases where recovery routines were specified is 44%, compared to 80% when no recovery routine was specified. This appears to show that recovery routines

have an effect in improving the system fault tolerance, but there is still considerable room for improvement. For essential jobs (where we expect degradation in service but not necessarily a system failure), the percentage of failures where recovery routines are specified drops to nearly 20%, compared to 72% when no recovery routines are specified. Thus, the recovery routines are doing a much better job of dealing with essential jobs than with critical jobs. In fact, one would like to see these figures reversed.

Table 11.9 relates the provision of recovery routines to the specified error classes when critical jobs are involved. (See Section 11.4.1 for the definition of error classes in MVS.) It is found that the recovery routines are most effective in dealing with storage management problems (an important feature of MVS). When no recovery routines are provided, the probability of a storage management failure is high (81%). The recovery routines are weakest in dealing with timing errors, I/O and data management errors, and programming exceptions. Thus, it appears that these are the particularly vulnerable areas of the system where further attention could be directed. To quantify the above figures, measures of fault tolerance were defined and evaluated.

Table 11.9 Effectiveness of recovery routines for critical jobs

Error Type	Frequency	Recovery Routine Provided %	Failures** (Recovery Routine Provided) %	Failures** (Recovery Routine Not Provided) %
Control	22	63.6	21.4	100.0*
Timing	29	82.8	100.0	100.0
I/O and Data Management	74	82.4	90.2	7.7
Storage Management	161	79.5	7.8	81.8
Storage Exceptions	82	18.3	46.7	92.5
Programming Exceptions	31	64.5	80.0	63.6
All	399	65.7	44.3	80.4

* The number of failures due to control errors were statistically insignificant.

** A failure means a job or task termination in critical jobs.

Here, the software fault tolerance (FT) (i.e., the probability of recovery given that a software error has occurred) is

$$FT = 1 - \frac{\text{number of failures}}{\text{total number of errors}} \tag{11.4}$$

where the number of failures is the number of job/task terminations of critical jobs. This measure was evaluated for all errors and each error category defined in the operating system.

Table 11.10 presents the system fault tolerance under two conditions. It shows how well the system handles all problems, i.e., regardless of the type of job in control at the time of error. It also shows the fault-tolerance measure (FT) when a critical job was in control at the time of error, to quantify how well the system recovery management handles serious system problems. The overall system fault tolerance to a software error is found to be 0.88. Table 11.10 shows that the system is weak in dealing with errors occurring on critical jobs. It is seen that the

system deals best with storage management and control problems. It is at its weakest in dealing with timing and exception errors. The figure for I/O and data management errors is rather low.

Table 11.10 Fault tolerance

Error Type	All Jobs	Critical Jobs
Control	0.80	0.50
Timing	0.90	0.00
I/O and Data Management	0.42	0.24
Storage Management	0.89	0.77
Storage Exceptions	0.63	0.16
Programming Exceptions	0.69	0.26
All	0.88	0.43

11.6 MODELING AND ANALYSIS

The previous sections discussed the fault tolerance of operating systems resulting from the use of process pairs and recovery routines, and basic error characteristics such as TTE and TTR distributions. This section investigates how all these factors affect system availability and completion of jobs, using Markov modeling and reward analysis. We build two levels of models using the data and conduct analyses to quantify the effects of errors and recovery on service loss and job completion. The low-level modeling focuses on error detection and recovery inside an operating system, while the high-level modeling deals with distributed systems in which multiple instances of operating systems interact. The IBM/MVS data are suited for illustrating the lower-level modeling, while the Tandem/GUARDIAN and VAX/VMS data are suited for illustrating the higher-level modeling and reward analysis. The two-level modeling and reward analysis not only allows us to evaluate software fault tolerance and software dependability, it also provides a framework for modeling complex software systems in a hierarchical fashion.

11.6.1 High-Level Modeling

In distributed environments such as the Tandem and VAXcluster systems, multiple instances of an operating system are running, and these instances form a single overall software system. In this subsection, each operating system instance is treated as a software element of the overall software system, and software fault tolerance is discussed at a high level. The modeling is illustrated using the Tandem/GUARDIAN and VAX/VMS data.

11.6.1.1 MODEL CONSTRUCTION

We constructed two-dimensional, continuous-time Markov models using the software error logs from the Tandem and VAXcluster systems. Figure 11.4 shows the model structure. In the model, the state $S_{i,j}$ represents that, out of a total of n operating systems, i copies are in an error state, j copies are in a failure state, and $(n - i - j)$ copies are running error-free. The

state transition probabilities were estimated from the measured data. For example, the state transition probability from state $S_{i,j}$ to state $S_{i,j+1}$ was obtained from

$$p_{(i,j),(i,j+1)} = \frac{\text{observed number of transitions from state } S_{i,j} \text{ to state } S_{i,j+1}}{\text{observed number of transitions out of state } S_{i,j}} . \quad (11.5)$$

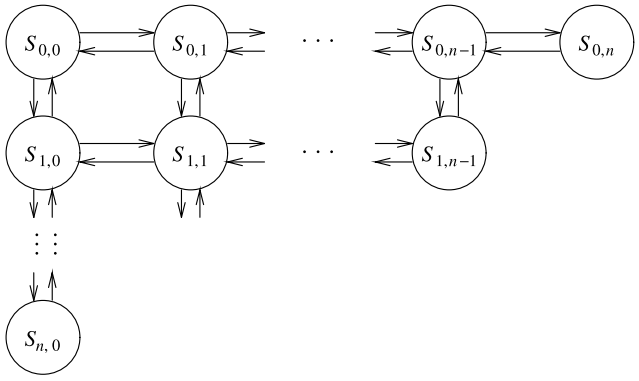


Figure 11.4 GUARDIAN and VMS software error/recovery model

11.6.1.2 REWARD FUNCTIONS

Performability models [Mey92] and reward models [Tri92] have been widely used to evaluate performance-related dependability measures in recent years. To evaluate the loss of service due to software errors, we define two reward functions for the Markov models. The first applies to a non-single-failure tolerant system, such as the VAXcluster, and the second applies to a single-failure tolerant system, such as the Tandem system.

A. NSFT (No Single-Failure Tolerance) Reward Function:

In a non-single-failure tolerant system, any recovery time spent on errors or failures results in degradation. Given a time interval ΔT, a reward rate for a single operating system is defined as

$$r(\Delta T) \; = \; W(\Delta T) \; / \; \Delta T \, , \quad (11.6)$$

where $W(\Delta T)$ denotes the useful service time provided by the operating system during the ΔT and is calculated by

$$W(\Delta T) \; = \; \begin{cases} \Delta T & \text{in normal state} \\ \Delta T - c\tau & \text{in error state} \\ 0 & \text{in failure state} \end{cases} , \quad (11.7)$$

where c is the number of raw errors occurring in the operating system during ΔT and τ is the mean recovery time for a single error. Thus, one unit of reward is given for each unit of time when the operating system is in the normal state. In an error state, the reward loss depends

on the amount of time the operating system spends on error recovery. (If ΔT is less than $c\tau$, $W(\Delta T)$ is set to 0.) In a failure state, the reward is zero.

With the above definition, the reward rate for state $S_{i,j}$ in the model (Figure 11.4) is obtained from

$$r_{i,j} = 1 \; - \; \frac{i \cdot \bar{c}\tau + j}{n} \tag{11.8}$$

where $\bar{c}$ is the average number of errors occurring in an operating system per unit time, when the operating system is in an error state. Here each operating system failure causes degradation.

B. SFT (Single-Failure Tolerance) Reward Function:

The Tandem system allows recovery from minor errors and can also tolerate a single operating system failure without noticeable performance degradation. (During job takeover, application programs would experience a short delay, which is typically less than 10 seconds.) To describe the built-in single-failure tolerance, we modify the reward rate (Equation 11.8) as follows:

$$r_{i,j} = \begin{cases} 1 \; - \; \frac{i \cdot \bar{c}\tau + j}{n} & if \; j = 0 \; or \; j = n \\ 1 \; - \; \frac{i \cdot \bar{c}\tau + (j-1)}{n} & if \; 1 \leq j \leq (n-1) \; . \end{cases} \tag{11.9}$$

Thus the first operating system failure causes no reward loss. For the second and subsequent failures, the reward loss is proportional to the number of these failures.

11.6.1.3 REWARD ANALYSIS

Given the Markov reward model described above, the expected steady-state reward rate, Y, can be estimated from [Tri92]

$$Y = \sum_{S_{i,j} \in S} r_{i,j} \cdot \Phi_{i,j} \tag{11.10}$$

where S is the set of valid states in the model and $\Phi_{i,j}$ is the steady-state occupancy probability for state $S_{i,j}$. The steady-state reward rate represents the relative amount of useful service the system can provide per unit time in the long run, and is a measure of service-capacity-oriented software availability. The steady-state reward loss rate, $(1 - Y)$, represents the relative amount of service lost per unit time due to software errors. If we consider a specific group of errors in the analysis, the steady-state reward loss quantifies the service loss due to this group of errors.

Table 11.11 shows the estimated steady-state reward loss due to software, nonsoftware, and all problems for the Tandem and VAXcluster systems. It is seen that software problems account for 30% of the service loss due to all problems in the Tandem system, while they account for 12% of the service loss due to all problems in the VAXcluster system. This indicates that software is not a dominant source of service loss in the measured VAXcluster system, while software is a significant source of service loss in the measured Tandem system.

A census of Tandem system availability [Gra90] has shown that, as the reliability of hardware and maintenance improves significantly, software is the major source (62%) of outages in the Tandem system. It is inappropriate, however, to directly compare our number with

Table 11.11 Estimated steady-state reward loss

System	Measure	Software	Nonsoftware	Total
Tandem	$1 - Y$	.00007	.00016	.00023
	Fraction (%)	30.4	69.6	100
VAXcluster	$1 - Y$	.00077	.00565	.00642
	Fraction (%)	12.0	88.0	100

Gray's because Gray's is an aggregate of many systems and ours is a measurement on a single system. Besides, the sources of the data and analysis procedures are different. Since our analysis is based on automatically generated event logs, some nonsoftware problems which require the replacement of faulty hardware can result in long recoveries and more reward loss. Also, because of the experimental nature of the measured Tandem system, nonsoftware problems due to operational or environmental faults may have been exaggerated. An operational or environmental fault can potentially affect all processors on the system.

11.6.1.4 WHAT DOES SINGLE-FAILURE TOLERANCE BUY?

The Tandem/GUARDIAN data allows us to evaluate the impact of built-in software fault tolerance on system dependability and to relate loss of service to different software components. We performed reward analysis using the two reward functions defined above (SFT and NSFT). The reward function defined in Equation 11.9 measures the reward loss under SFT. The reward function defined in Equation 11.8 allows us to determine the reward loss assuming no SFT. Difference between the two functions provides evaluation of the improvement in service due to the built-in single-failure tolerance. We evaluated the impact of six groups of halts on overall system dependability: all software halts, four mutually exclusive subsets of software halts, and all nonsoftware halts. The four subsets of software halts were formed based on the processes that were executing prior to the occurrences of software halts.

The first and the second columns of Table 11.12 show the estimated steady-state reward loss with and without SFT, respectively. The third column of the table shows what the fault tolerance buys, i.e., the decrease in reward loss due to the fault tolerance. It is seen that the single-failure tolerance in the measured system reduced the service loss due to software halts by approximately 90%. This clearly demonstrates the effectiveness of this fault tolerance mechanism against software failures and corroborates the results obtained in Section 11.5.1. The last row also shows that the single-failure tolerance reduced the service loss due to all nonsoftware halts by 92%.

The first column of Table 11.12 shows that nearly 100% of the service loss due to software halts with SFT was caused by the halts that occurred while the memory manager or an interrupt handler was executing. This indicates that some of these halts affected more than one operating system at the same time, while the rest of the software halts affected a single operating system. The software halts under interrupt handlers can occur for three reasons. First, they can occur due to bugs in interrupt handlers. Secondly, some problems are always detected during interrupt handling. For example, the failure of the "I'm alive" message protocol, which

Table 11.12 Estimated steady-state reward loss (Tandem)

Halt Group	With SFT		With NSFT		What fault tolerance buys: $\left(1-\frac{Reward\ Loss_{SFT}}{Reward\ Loss_{NSFT}}\right)$
	Reward Loss	(%)	Reward Loss	(%)	
SW, all	0.00007	100	0.00062	100	89%
SW, interrupt handlers	0.00003	43	0.00023	37	87%
SW, memory manager	0.00004	57	0.00035	56	89%
SW, all others	0	0	0.00003	5	100%
SW, unknown	0	0	0.00001	2	100%
Non-SW, all	0.00016		0.00205		92%

SW = Software halts

is a timeout in sending or receiving the "I'm alive" message, is detected at clock interrupt. Thirdly, sometimes an interrupt handler is called after a problem is detected by other routines. The software halts under the memory manager can occur due to bugs in memory management software or due to illegal requests from other processes.

11.6.2 Low-Level Modeling

This subsection discusses a low-level model that describes error detection and recovery inside an operating system. The model is illustrated using the IBM/MVS data. In the MVS operating system, when a program is abnormally interrupted due to an error, the supervisor routine gets control. If the problem is such that further processing can degrade the system or destroy data, the supervisor routine gives control to Recovery Termination Manager (RTM), an operating system module responsible for error and recovery management. If a recovery routine is available for the interrupted program, the RTM gives control to this routine before it terminates the program.

More than one recovery routine can be specified for the same program. If the current recovery routine is unable to restore a valid state, the RTM can give control to another recovery routine, if available. This process is called *percolation*. The percolation process ends if either a routine issues a valid retry request or no more recovery routines are available. In the latter case, the executing program and its related subtasks are terminated. An error recovery can result in the following four situations:

- Resume Operation (Resume Op): The system successfully recovers from the error and returns control to the interrupted program.
- Task Termination (Task Term): The program and its related subtasks are terminated, but the system does not fail.
- Job Termination (Job Term): The job in control at the time of the error is aborted.
- System Damage (System Failure): The job or task that was terminated is critical for system operation. As a result of the termination, a system failure occurs.

11.6.2.1 MODEL CONSTRUCTION

Using the collected error and recovery data, we constructed a semi-Markov model that provides a complete view of the measured MVS operating system from error detection to recovery. The states of the model consists of eight types of error states (see Table 11.13) and

four states resulting from error recoveries. Figure 11.5 shows the model. The Normal state represents that the operating system is running error-free. The transition probabilities were estimated from the measured data using Equation 11.5. Note that the System Failure state is not shown in the figure. This is because the occurrence of system failure was rare, and the number of observed system failures was statistically insignificant.

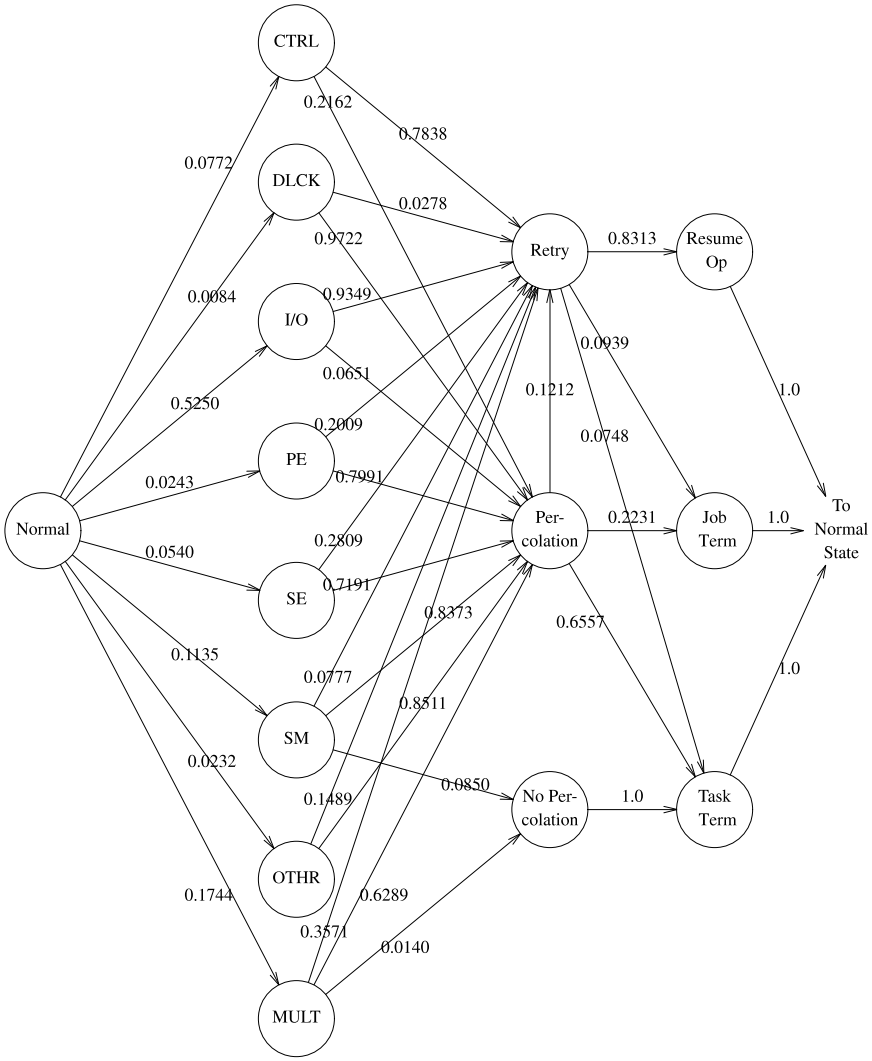


Figure 11.5 MVS Software error/recovery model

Table 11.13 shows the waiting time characteristics of the normal and error states in the model. The waiting time for a state is the time the system spends in that state before making a transition. In the table, a multiple software error is defined as an error burst consisting of more than one type of software error. Table 11.13 shows that the duration of a single error is typically in the range of 20 to 40 seconds, except for DLCK (deadlock), OTHR (others), and MULT (multiple error). The average recovery time from a program exception is twice as

long as that from a control error (21 seconds versus 42 seconds). This is probably due to the extensive software involvement in recovering from program exceptions. Table 11.13 clearly highlights the importance of incorporating multiple errors into a system model. The average duration of a multiple error is at least four times longer than that of any type of single error.

Table 11.13 Waiting time

State	# Observations	Mean Waiting Time (Sec.)	Standard Deviation
Normal (Error-Free)	2757	10461.33	32735.04
CTRL (Control Error)	213	21.92	84.21
DLCK (Deadlock)	23	4.72	22.61
I/O (I/O & Data Management Error)	1448	25.05	77.62
PE (Program Exception)	65	42.23	92.98
SE (Storage or Address Exception)	149	36.82	79.59
SM (Storage Management Error)	313	33.40	95.01
OTHR (Other Type)	66	1.86	12.98
MULT (Multiple Software Error)	481	175.59	252.79

An error recovery can be as simple as a retry or as complex as requiring several percolations before a successful retry. The problem can also be such that no retry or percolation is possible. Figure 11.5 shows that about 83.1% of all retries are successful. The figure also shows that the operating system attempts to recover from 93.5% of I/O and data management errors and 78.4% of control related errors by retries. These observations indicate that most I/O and control related errors are relatively easy to recover from, compared to the other types of errors such as deadlock and storage errors. Also note that "No Percolation" occurs only in recovering from storage management errors. This indicates that storage management errors are more complicated than the other types of errors. The problem can also be such that no retry or percolation is possible.

11.6.2.2 MODEL EVALUATION

The dynamic behavior of the modeled operating system can be described by various probabilities. Given the irreducible semi-Markov model of Figure 11.5, the following steady-state probabilities were evaluated. The derivations of these measures are given in [How71].

- Transition probability (π_j): given that the system is now making a transition, the probability that the transition is to state j.
- Occupancy probability (Φ_j): at any point in time the probability that the system occupies state j.
- Mean recurrence time ($\bar{\Theta}_j$): mean recurrence time of state j.

The occupancy probability of the normal state can be viewed as the operating system availability without degradation. The state transition probability, on the other hand, characterizes error detection and recovery processes in the operating system. Table 11.14(a) lists the state transition probabilities and occupancy probabilities for the normal and error states. Table 11.14(b) lists the state transition probabilities and the mean recurrence times of the

recovery and result states. A dashed line in the table indicates a negligible value (less than 0.00001).

Table 11.14 Error/recovery model characteristics

Measure	Normal State	Error State							
		CTRL	DLCK	I/O	PE	SE	SM	OTHR	MULT
π (%)	24.74	1.91	0.20	12.99	0.60	1.34	2.81	0.57	4.31
Φ (%)	99.50	0.016	—	0.125	0.0098	0.0189	0.036	—	0.291

(a)

Measure	Recovery State			Resultant State		
	Retry	Percolation	No Percolation	Resume Op	Task Term	Job Term
π (%)	17.04	8.45	0.30	14.14	7.12	3.48
$\overline{\Theta}(hr.)$	4.25	8.55	241.43	5.11	10.16	20.74

(b)

Table 11.14(a) shows that the occupancy probability of the normal state in the model is 0.995. This indicates that in 99.5% of the time the operating system is running error-free. In the other 0.5% of time the operating system is in the error or recovery states. In more than half of the error and recovery time (i.e., 0.29% out of 0.5%), the operating system is in the multiple error state. An early study of the MVS error/recovery estimated that the average reward rate for the software error/recovery state is 0.2736 [Hsu88]. Based on this reward rate and the occupancy probability for the error/recovery state shown in the table (0.005), it can be estimated that the steady-state reward loss in the modeled MVS is 0.00363.

By solving the model (Figure 11.5), it is found that the operating system makes a transition every 43.37 minutes. Table 11.14 shows that 24.74% of all transitions made in the model are to the normal state, 24.73% of them are to error states (obtained by summing all the π's for all error states), 25.79% of them are to recovery states, and 24.74% of them are to result states. Since a transition occurs every 43 minutes, it can be estimated that, on the average, a software error is detected every three hours and a successful recovery (i.e., reaching the Resume Op state) occurs every five hours. This indicates that nearly 43% of all software errors result in user task/job terminations. Although these terminations do affect the user perceived reliability and availability, only a few (statistically insignificant number) of them lead to system failures. This result indicates that recovery routines in MVS are effective in avoiding system failures, but are not so effective in avoiding user job terminations.

11.7 CONCLUSIONS

In this chapter, we provided data and analysis of the dependability and fault tolerance for three operating systems: the Tandem/GUARDIAN system, the VAX/VMS system, and the IBM/MVS system. Measurements were made on these systems for substantial periods to collect software error and recovery data. Basic software error characteristics were investigated via fault and error classification and via the analysis of error distributions and correlations. Fault tolerance in operating systems resulting from the use of process pairs and recovery routines was evaluated. A two-level modeling and reward analysis were used to analyze and evalu-

ate error and recovery processes inside an operating system and interactions among multiple instances of an operating system running in a distributed environment.

Process pairs in Tandem systems tolerate about 70% of defects in system software that result in processor failures. The loose coupling between processors which results in the backup execution (the processor state and the sequence of events occurring) being different from the original execution is a major reason for the measured software fault tolerance. The results indicate that the level of software fault tolerance achieved by the use of process pairs depend on differences between the original and backup executions and the proportion of subtle faults in the software.

The measurements in IBM/MVS showed that the system fault tolerance almost doubles when recovery routines are provided, in comparison to the case where no recovery routines are available. However, even when recovery routines are provided, there is almost a 50% chance of system failure when critical system jobs are involved. The system recovery routines are most effective in handling storage management problems (an important feature of MVS). Timing, I/O and data management, and exceptions are the main problem areas. The overall system availability is very high (0.995). From the user perspective, however, this is not quite so, since more than 40% of all software errors lead to user job/task termination. The above results show that a combination of recovery routines and process pairs—first try recovery routines to see if the failed process can be recovered locally and then switch over to the backup when the recovery routine fails— can be a viable approach for tolerating software faults.

Software errors tend to occur in bursts on all measured systems. Software TTE distributions obtained from the data are not simple exponentials. Both the VAXcluster and Tandem data demonstrated that software TTE distributions can be modeled by a two-phase hyperexponential random variable: a lower rate error pattern that characterizes regular errors, and a higher rate error pattern that characterizes error bursts and concurrent errors on multiple machines. The investigation of error correlations found that about 10% of software failures in the VAXcluster and 20% of software halts in the Tandem system occurs concurrently on multiple machines. It is suspected that the network-related software in the VAXcluster is a software reliability bottleneck, in terms of correlated failures.

It should be emphasized that the results of this study should not be interpreted as a direct comparison among the three measured operating systems, but rather an illustration of the proposed methodology. Differences in operating system architectures, instrumentation conditions, measurement periods, and operational environments make a direct comparison impossible. It is suggested that more measurements and analyses be conducted in a manner proposed here so that a wide range of information on software fault tolerance in computer operating systems is available.

ACKNOWLEDGMENTS

We thank Tandem Computers Incorporated, NASA AMES Research Center, and IBM Poughkeepsie for their assistance in collecting data from the Tandem, VAXcluster, and IBM machines, respectively. This research was supported by NASA under Grant NAG-1-613, in cooperation with the Illinois Computer Laboratory for Aerospace Systems and Software (ICLASS), by Tandem Computers, and by the Office of Naval Research under Grant N00014-91-J-1116.