Software Fault Tolerance in a Clustered Architecture: Techniques and Reliability Modeling

Michael R. Lyu
Computer Science and Engineering Department
The Chinese University of Hong Kong
Shatin, Hong Kong
+852-2609-8429
lyu@cse.cuhk.edu.hk

Veena B. Mendiratta
Bell Labs., Luecnt Technologies
2000 N. Naperville Road
Naperville, IL 60566, USA
+1-630-979-3872
veena@lucent.com

Abstract—System architectures based on a cluster of computers have gained substantial attention recently. In a clustered system, complex software-intensive applications can be built with commercial hardware, operating systems, and application software to achieve high system availability and data integrity, while performance and cost penalties are greatly reduced by the use of separate error detection hardware and dedicated software fault tolerance routines. Within such a system a watchdog provides mechanisms for error detection and switch-over to a spare or backup processor in the presence of processor failures. The application software is responsible for the extent of the error detection, subsequent recovery actions and data backup. The application can be made as reliable as the user requires, being constrained only by the upper bounds on reliability imposed by the clustered architecture under various implementation schemes.

We present reliability modeling and analysis of the clustered system by defining the hardware, operating system, and application software reliability techniques that need to be implemented to achieve different levels of reliability and comparable degrees of data consistency. We describe these reliability levels in terms of fault detection, fault recovery, volatile data consistency, and persistent data consistency, and develop a Markov reliability model to capture these fault detection and recovery activities. We also demonstrate how this cost-effective fault tolerant technique can provide quantitative reliability improvement within applications using clustered architectures.

TABLE OF CONTENTS

1. INTRODUCTION
2. RCC PRINCIPAL TECHNIQUES AND ARCHITECTURE ASSUMPTIONS
3. RELIABILITY TECHNIQUES
4. RELIABILITY MODELING AND ANALYSIS
5. CONCLUSIONS

[1] The first author was supported by a Direct Grant from the Chinese University of Hong Kong for this research work.

1. INTRODUCTION

Modern systems are required to be available upon user's request and their data should be consistent in the user's view. The requirements for *availability* and *data consistency* vary by the type of user. For example,

- users of telephone switching systems demand continuous availability,
- users of bank teller machines demand the highest degree of data consistency, and
- safety critical real-time systems need the highest levels of both availability and data consistency, and have an additional requirement to *fail safe*.

The events of system unavailability and data inconsistency are often caused by the existence and manifestation of faults in the system. To tolerate faults, most systems incorporate some form of redundancy scheme in their design and implementation. Tolerating faults in redundant systems involves detecting a component failure, gathering information about the failure and recovering from the failure. The more faults the system can tolerate, the higher the system reliability is. Faults that are not tolerated generally lead to a total system failure, whereas in some cases the impact may be confined to a partial system failure. In traditional fault tolerant and high reliability systems, these fault tolerance actions, including data backup, are provided by the hardware, operating system or database system, and to a much smaller extent by software in the application.

The *clustered architecture* approach to achieving system reliability is somewhat different. The system is built with commercial hardware, operating system and database system. A *watchdog* provides the means for error detection and switchover to a spare/backup processor in the event of a failure of the active processor. However, the application software is responsible for error detection, consequent recovery actions and data backup. The application, therefore, can be made as reliable as the user requires it to be, being constrained only by the upper bounds on reliability imposed by the underlying architecture, and other performance and cost considerations.

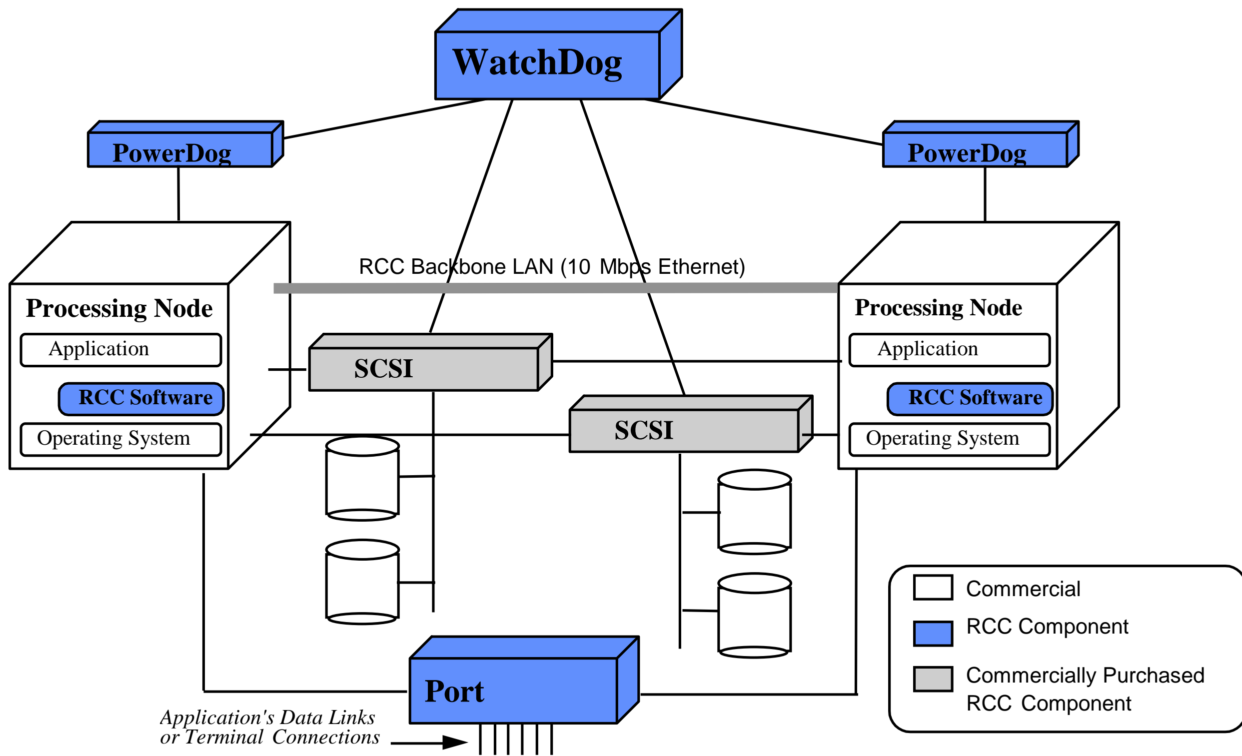


Figure 1 RCC Active/Standby Configuration

This paper presents a framework for reliability analysis of a clustered architecture called the Reliable Clustered Computing (RCC) system [1] and defines the hardware, operating system and application software reliability techniques that need to be implemented to achieve different levels of reliability — 0.9, 0.99, 0.999 and 0.9999 availability and comparable degrees of data consistency. To aid in the consistent use of terminology, failure response stages defined in [2] are reproduced in Appendix I.

2. RCC Principal Techniques and Architecture Assumptions

RCC was developed by Lucent Technologies. The purpose of RCC is to provide a low-cost computing platform for generic applications composed of standard commercial computing industry hardware and software, achieving high level of system availability for critical applications [1].

The RCC system architecture is organized as a loosely coupled collection of processing elements or nodes connected by a standard system interconnect, typically a local area network (LAN). Each node is a standard, commercially available computer (such as a workstation or PC), running a standard operating system. RCC components add the software and hardware “glue” that provides for increased reliability, availability, and maintainability of the overall configuration to enable a successful distributed processing platform. A cluster may have a redundant $n+k$ configuration, where k processing nodes serve as spares for the n active processing elements in the cluster. An assumption is that any member of the cluster is capable of supporting the processing functions of any other member. The simplest redundant configuration, shown in Figure 1, is Active-Standby, where there is one active node and one standby node (i.e., both n and k are 1). Other permissible cluster configurations are simplex (one active node, no spare) and active-active (two active nodes, no spares). An RCC system is composed of at least one cluster of processing elements, plus the Emergency Action Interface (EAI). A system may optionally have additional clusters.

At the heart of an RCC system is a dedicated recovery and maintenance processor known as the system WatchDog which, along with associated configuration management and fault recovery software running on the nodes, controls the cluster configurations. The RCC System Integrity software interfaces and works with the WatchDog to monitor and control system and cluster configurations, maintain node state and resource information, and direct fault recovery actions if required. Provisions for reliable resource monitoring are also included. Application programming interfaces (APIs) are provided to the System Integrity software to allow application software to exchange status information with the RCC components and direct configuration and fault recovery activities if desired, depending on the selected fault recovery strategy.

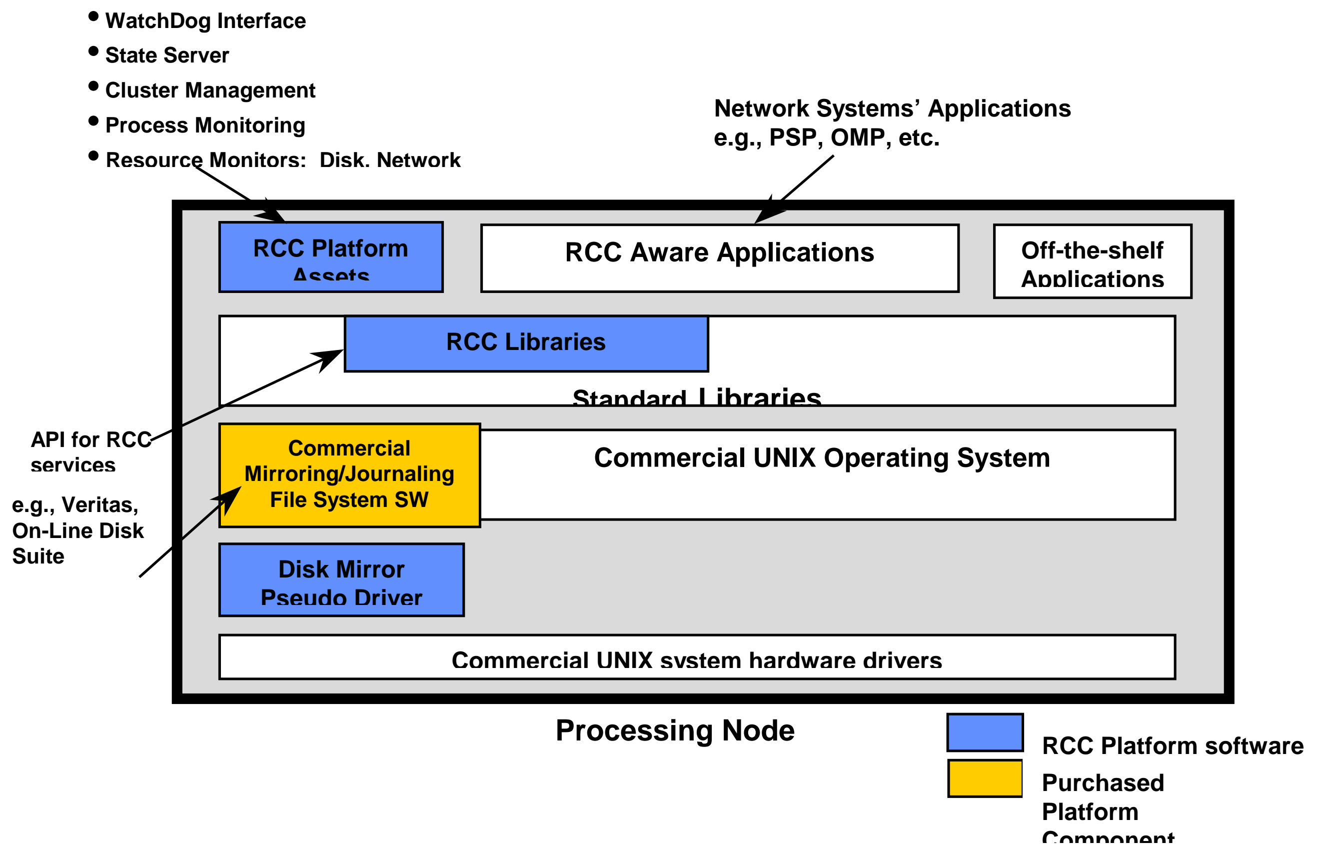


Figure 2 Reliable Clustered Computing Software Architecture

The RCC software architecture within each processing node is shown in Figure 2. Within RCC, a resource monitor is a process that monitors the availability of a particular resource and relays that information to other System Integrity software components, as the (un)availability of the resource may need to affect the usability of the node and/or the cluster configuration. The implementation of a resource monitor, by its nature, is unique to the type of resource it monitors. Resource monitors are included for the RCC add-on packages (e.g., Disk Mirroring). In addition, a template and interfaces are provided for applications to implement and add resource monitors for application-specific resources, such as special peripherals (e.g., voice circuits).

3. Reliability Techniques

3.1 Reliability Dimensions

From a user's point of view, reliability is related to two dimensions: *availability* and *data consistency*.

Availability

This is the proportion of time that the system is in working order or *up*, and, for repairable systems, it is defined as

Availability = MTBF/(MTBF+MTTR),

where

MTBF = Mean Time Between Failures (Reliability)

MTTR = Mean Time To Repair (Maintainability)

Data Consistency

There are two categories of data: *volatile data* and *persistent data*. Volatile data are data which reside in the memory for the duration of the application. When the application is complete the volatile data will no longer exist. Persistent data are data which have been permanently written in the secondary storage (disk, tape, etc.). Data consistency is achieved through the implementation of reliability techniques such as checkpointing, message logging, journaling and data backup on disk in the application software. These techniques provide the means to recover the application volatile and persistent data after the occurrence of a hardware or software failure. The extent of the data recovered after a failure is a function of the reliability techniques [2], described in Appendix II, that are implemented.

3.2 Reliability Models

The hardware and software models provide predictions of system availability. The data consistency model provides

predictions of the defect rate, that is, the units of load – calls, messages, transactions, etc. – that are lost due to hardware and software failures as a proportion of the total offered load.

Hardware model inputs

- hardware unit MTBF
- sparing scheme
- hardware fault recovery coverage factor
- hardware failure recovery duration
- MTTR

Software fault model inputs

- software failure rate
- software fault recovery coverage factor
- software failure recovery duration

Data consistency model inputs

- hardware and software failure rate
- hardware and software failure recovery duration
- transient and persistent data backup mechanisms implemented.

3.3 Levels of Reliability

The definition of the levels of reliability presented below is based partly on the definition of *levels of software fault tolerance* presented in Reference [3]. The reliability levels are in ascending order, that is, Level 1 is more reliable than Level 0, Level 2 is more reliable than Level 1, and so forth.

Level 0: Basic automatic fault detection by watchdog, no automatic fault recovery, no data consistency.

A small set of fault classes – hardware and software – is detected by the watchdog. On detection of a fault, the system halts and manual intervention is necessary. For a hardware fault, the system is manually reconfigured, and the faulty processor is taken out of service. For a software fault, the application process is restarted at the initial internal state which will require initialization of the faulty processor since the application may leave its data in an inconsistent or incorrect state.

Level 1: Basic automatic fault detection by watchdog, automatic fault recovery, no data consistency.

A small set of fault classes – hardware and software – is detected by the watchdog and recovery is automatic. When a fault is detected by the watchdog, the system is automatically recovered – reconfigured for hardware faults and initialized for software faults.

The internal state of the application is not saved and, hence, the process restarts at the initial internal state. Restart along with reinitialization is slow. The restarted internal state may not reflect all the messages that have been processed in the previous execution and thus, may not be consistent with the persistent data.

Difference between Levels 0 and 1 is the following: since detection and restart are automatic in Level 1, the Level 1 application availability is higher.

Level 2: Level 1 plus enhanced automatic fault detection by watchdog plus periodic checkpointing, logging and recovery of internal state.

The watchdog and application are enhanced to automatically detect a larger set of fault – hardware and software – classes. The internal state of the application process is periodically checkpointed, that is, the critical volatile data are saved, and the messages to the application are logged. After a hardware failure is detected, the system is reconfigured around the faulty unit.

Then, for both hardware and software failures, the application is restarted at the most recent checkpointed internal state and the logged messages are reprocessed to bring the application close to the state at which it crashed. Difference between Levels 1and 2 is the following: application availability and volatile data consistency are higher in Level 2.

Level 3: Level 2 plus persistent data recovery. (this is the highest level achievable with RCC)

In addition to the capabilities in Level 2, the persistent data of the application is replicated on a backup disk connected to a backup node, and is kept consistent with the data on the primary node throughout the normal operation of the application. In case of a fault – hardware and software – and resulting recovery of the application on the backup node, the backup disk brings the application's persistent data as close to the state at which the application crashed as possible. Difference between Levels 2 and 3 is the following: data consistency of the application in Level 3 is higher.

Level 4: Continuous operation without interruption.

This level of reliability is not achievable with the RCC and, therefore, will not be discussed further in this paper.

4. RELIABILITY MODELING AND ANALYSIS

4.1 Basic Model for Software Fault Tolerance

Figure 3 shows the reliability model for systems incorporating various levels of software fault tolerance. This model can be used to describe different levels of RCC engagement in a target system regarding its reliability. Different levels of RCC impact the system reliability through the change of parameters in these models, which will be discussed in the following sections.

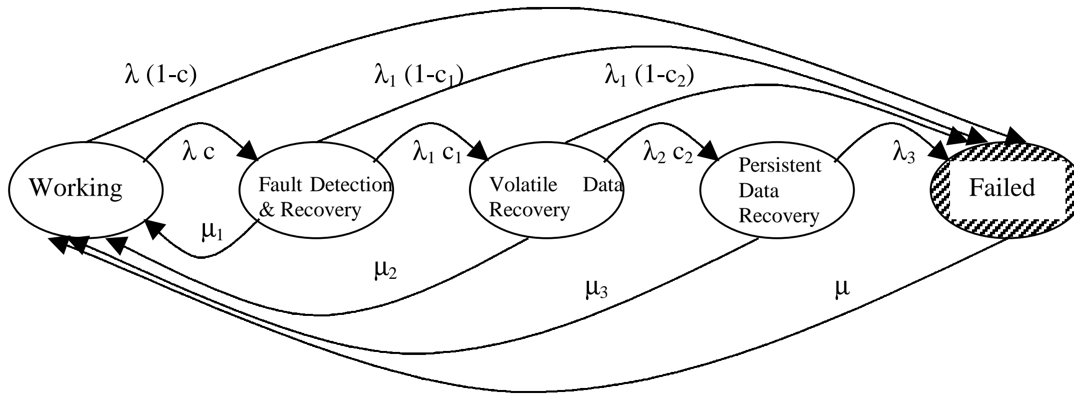


Figure 3 Conceptual Model for Software Fault Tolerance

There are five states in this basic model: Working, Fault Detection and Recovery, Volatile Data Recovery, Persistent Data Recovery, and Failed. Transitions from the Fault Detection and Recovery state to the Volatile Data Recovery state and so forth represent progressive recovery effort within the system. This can also be viewed as an escalating recovery policy. Details on the escalating recovery model can be found in [5] and [6].

In Figure 3 the state Working represents the normal execution state of the system. In the event of an error, the system will go into other states. λ denotes this exiting rate (i.e., the error rate), and c represents the fault recovery coverage factor for the error. If the error is recoverable, the system enters the Fault Detection & Recovery state where escalating recovery starts. If the error is recovered in this state, it goes back to Working; otherwise, the system either fails or another level of recovery is entered. μ_1 denotes the rate at which successful recovery is performed in this state, while λ_1 denotes the rate at which recovery cannot complete in this state. In the latter case, the system either fails (with conditional probability c_1), or enters the Volatile Data Recovery state (with conditional probability $1-c_1$). This recovery process goes on to Volatile Data Recovery state and Persistent Data Recovery state in a similar fashion, and the parameters associated with these two states are (μ_2, λ_2, c_2) and (μ_3, λ_3), respectively. Finally, μ is the manual repair rate.

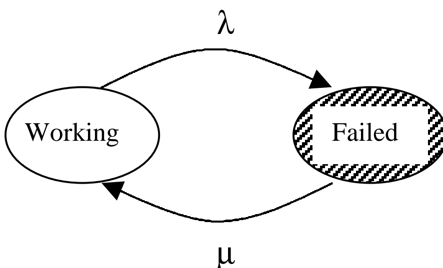


Figure 4 Model for Non Fault-Tolerant Systems

Figure 4 shows the reliability models for a system without fault tolerance, in which case the system goes from state Working to state Failed in the presence of an error (with error rate λ).

4.2 Sample Reliability Measures for the Conceptual Model

As an illustration, we give values for the c and μ parameters for different reliability levels for the conceptual model depicted in Figure 3. We list typical situations for reliability levels 0 through 3. We assume the failure rate to be 0.001 failures per hour and the manual repair rate to be 0.25 repairs per hour. Note that NA represents Not Applicable since the value does not impact the result.

Level 0 Reliability

Level 0 contains only basic automatic fault detection by watchdog. There is no automatic fault recovery and no data consistency. Typical parameters are

$\lambda = 0.001$, $c = 0$, $\mu = 1/4$
$\lambda_1 =$ NA, $c1 = 0$, $\mu_1 =$ NA
$\lambda_2 =$ NA, $c_2 = 0$, $\mu_2 =$ NA
$\lambda_3 =$ NA

That is, the "Active State" reliability model reduces to a 2-state model. This will be equivalent to the model for the "Standby State". The resulting unavailability in this case is 2094 minutes down time per year.

Level 1 Reliability

$\lambda = 0.001$, $c = 0.9$, $\mu = 1/4$
$\lambda_1 = 30$, $c_1 = 0$, $\mu_1 = 30$
$\lambda_2 =$ NA, $c_2 = 0$, $\mu_2 =$ NA
$\lambda_3 =$ NA

That is, the "Active State" reliability model reduces to a 3-state model, and the coverage factor c becomes non-zero. The resulting unavailability is 1162 minutes down time per year.

Level 2 Reliability

$\lambda = 0.001$, $c = 0.99$, $\mu = 1/4$
$\lambda_1 = 30$, $c_1 = 0.9$, $\mu_1 = 30$
$\lambda_2 = 1800$, $c_2 = 0$, $\mu_2 = 1800$
$\lambda_3 = NA$, $\mu_3 = NA$

That is, the "Active State" reliability model reduces to a 4-state model, c value increases, and c_1 becomes non-zero. The resulting unavailability is 593 minutes down time per year.

Level 3 Reliability

$\lambda = 0.001$, $c = 0.999$, $\mu = 1/4$
$\lambda_1 = 30$, $c_1 = 0.99$, $\mu_1 = 30$
$\lambda_2 = 1800$, $c_2 = 0.9$, $\mu_2 = 1800$
$\lambda_3 = 100$, $\mu_3 = 3600$

That is, the "Active State" reliability model is a full 5-state model, c value increases further, c_1 value increases, and c_2 becomes non-zero. The resulting unavailability is 98 minutes down time per year.

4.3 RCC Reliability Analysis

In this section we present the reliability analysis, using our escalation reliability model, to study a hypothetical RCC application. We use the recovery escalation model depicted in Figure 3 to predict the system reliability under RCC application. The modeling parameters are classified in three categories: failure rates, recovery rates, and coverage factors. The failure rates are estimated from similar projects. The recovery rates are estimated from the underlying recovery mechanism provided by RCC, and the coverage factors are assumed in different scenarios. The parameter values used in the reliability analysis are listed in Table 1.

Table 1 Sample Model Inputs

Id	Description	Value/Range
λ_h	hardware failure rate	0.00001 failures per hour (~0.1 failures per year)
λ_s	software failure rate	0.00114 failures per hour (~10 failures per year)
λ	total failure rate	(1) 0.00114 failures per hour (2) 0.00228 failures per (3) 0.00342 failures per hour
λ_1	level 1 to 2 escalation rate	30 exits per hour (2 minutes)
λ_2	level 2 to 3 escalation rate	1800 exits per hour (2 seconds)
λ_3	failure rate at level 3	100 exits per hour (0.66 minutes)
μ_1	repair rate at level 1	30 recoveries per hour (2 minutes 60 recoveries per hour (1 minute)
μ_2	repair rate at level 2	1800 recoveries per hour (2 seconds) 3600 recoveries per hour (1 second)
μ_3	repair rate at level 3	3600 recoveries per hour (1 second)
μ	manual repair rate	1/4 repairs per hour (i.e., MTTR = 4 hours)
c	fault detection coverage	0.9, 0.99
c_1	level 1 to 2 coverage	0.9, 0.99
c_2	level 2 to 3 coverage	0.9, 0.99

Table 2 shows the results in our reliability analysis. The numerical computation was obtained using the SHARP software package [7]. The expected downtime for the system, expressed in minutes per year, is listed in this table for various scenarios, including three types of failure rate (10,20,30 failures per year), two sets of repair rates, and three values of coverage for C (0.99, 0.9 and 0). "Total Down Time" includes all the time periods when the system is not available (which include both time for automatic repair and time for manual repair), and "Failed Time" represents the time when the system is under manual repair.

From Table 2 we can see that the reliability of the system is influenced by all the parameters in the model, particularly, the coverage factor c. When c is 0.99 and when the recovery rate is high, the total system down time can be as low as 48 minutes when 10 failures per year are encountered. This value would achieve the RCC availability goal of 0.9999 (less than 52.6 minutes per year down time), and is very compatible to leading edge availability achievement described in [4]. RCC, however, can achieve this figure in a much more cost-effective fashion since it does not require special-purpose high-reliability computer systems. On the other hand, if the coverage is as low as 0.9, the achievable availability is reduced to as high as 344 to 416 minutes down time per year in the presence of 10 failures. The last column in Table 2 (c=0) list the case where no fault tolerance is available in the system, where down time per year is usually in the order of thousands of minutes.

5. CONCLUSIONS

We propose a generic Markov chain based reliability model to describe software fault tolerance mechanism provided by the RCC architecture. We describe the error detection and recovery methods, including process recovery, volatile data recovery, and persistent data recovery, using several levels of recovery procedures. The resulting model is applied to a hypothetical yet typical RCC-monitored system. Our results show that if coverage can be well provided by the RCC fault-tolerant mechanism, the reliability and availability of the target system can be greatly improved over non fault-tolerant architecture, where manual recovery can be very timely and costly. We provide several scenarios and the resulting reliability measures to illustrate the importance and criticality of RCC in system reliability and availability improvement.

Table 2: Reliability Analysis Results for the Sample RCC Application

Coverage	c = 0.99		c = 0.9		c = 0
	Total Down Time	**"Failed" Time**	**Total Down Time**	**"Failed" Time**	**Down Time**
μ_1=30, μ_2=1800, μ_3=3600					
10 failures per year	67.4	57.4	416.6	407.5	2385
20 failures per year	134.8	114.8	832.6	814.4	4750
30 failures per year	202.2	172.2	1247.9	1220.7	7093
μ_1=60, μ_2=3600, μ_3=3600					
10 failures per year	48.1	41.5	344.2	338.2	2385
20 failures per year	96.2	82.9	688.0	676.6	4750
30 failures per year	144.3	124.4	1031.3	1013.2	7093

REFERENCES

[1] G. Hughes-Fenchel, "A Flexible Clustered Approach to High Availability," *Proceedings of the Twenty-Seventh Annual International Symposium on Falut-Tolerant Computing*, Seattle, WA, June, 1997.

[2] D. P. Siewiorek and R. S. Swarz, *Reliable Computer Systems: Design and Evaluation*, Digital Press, 1992.

[3] Y. Huang and C. Kintala, "Software Fault Tolerance in the Application Layer," In M. R. Lyu (Ed.), *Software Fault Tolerance*, John Wiley, 1995.

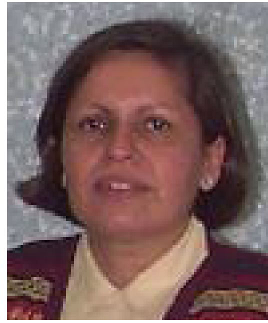
[4] "High Availability Trends: A Poll of Leading Edge Users," D.H.Brown Associates, Inc., September, 1995.

[5] D. A. Hoeflin and V. B. Mendiratta, "An Elementary Model for Predicting Switching System Outage Durations," *Proceedings of the XV International Switching Symposium*, Berlin, April 1995.

[6] V. B. Mendiratta, "Assessing the Reliability Impacts of Software Fault-Tolerance Mechanisms," *Proceedings of 1996 International Symposium on Software Reliability Engineering*, White Plains, NY, October 1996.

[7] R.A. Sahner, K.S. Trivedi, and A. Puliafito, *Performance and Reliability Analysis of Computer Systems: An Example-Based Approach Using the SHARPE Software Package,* Kluwer Academic Publishers, Boston, MA, 1996.

Michael R. Lyu *is currently an Associate Professor at the Computer Science and Engineering Department of the Chinese University of Hong Kong. He worked at the Jet Propulsion Laboratory as a Member of the Technical Staff from 1988 to 1990. From 1990 to 1992 he was with the Electrical and Computer Engineering Department at the University of Iowa as an Assistant Professor. From 1992 to 1995, he was a Member of the Technical Staff in the Applied Research Area of the Bell Communications Research (Bellcore). From 1995 to 1997, he was a Member of the Technical Staff at Bell Labs Research, which was originally part of AT&T and later became part of Lucent Technologies. Dr. Lyu's research interests include software reliability engineering, software process and metrics, distributed systems, and fault-tolerant computing. He has published over 80 refereed journal and conference papers in these areas. He initiated the first International Symposium on Software Reliability Engineering (ISSRE) in 1990. He was the program chair for ISSRE'96, and has served in program committees for many conferences. He is the editor for two book volumes: Software Fault Tolerance, published by Wiley in 1995 and the Handbook of Software Reliability Engineering, published by IEEE and McGraw-Hill in 1996. He is an associated editor of IEEE Transactions on Reliability and an editor for IEEE Transactions on Knowledge and Data Engineering. He is a senior member of IEEE.*



Veena B. Mendiratta has been at Bell Labs, Lucent Technologies (formerly AT&T) since 1984. She is currently a Distinguished Member of Technical Staff in the Switching Architecture, Performance and Engineering Department in the Switching and Access Solutions business unit. Dr Mendiratta's interests are in the areas of fault tolerant computing and software reliability engineering. Most of Dr Mendiratta's work has focused on the reliability and performance analysis of switching and access systems to guide system architecture solutions. She has presented papers at several refereed conferences and is a member of IEEE and INFORMS.

Appendix I: Failure Response Stages

1. Fault confinement. This stage limits the spread of fault effects to one area of the system, thus preventing contamination of other areas. Fault-confinement can be achieved through use of: fault-detection circuits, consistency checks and multiple requests/confirmations.
2. Fault detection. This stage recognizes that something unexpected has occurred in the system. Fault latency is the period of time between the occurrence of a fault and its detection. Techniques fall in 2 classes: off-line and on-line. With off-line techniques, such as diagnostic programs, the device is not able to perform useful work while under test. On-line techniques, such as parity and duplication, provide a real-time detection capability that is performed concurrently with useful work.
3. Diagnosis. This stage is necessary if the fault detection technique does not provide information about the failure location and/or properties.
4. Reconfiguration. This stage occurs when a fault is detected and a permanent failure is located. The system may reconfigure its components either to replace the failed component or to isolate it from the rest of the system.
5. Recovery. This stage utilizes techniques to eliminate the effects of faults. Two basic recovery approaches are based on: fault masking, retry and rollback. Fault-masking techniques hide the effects of failures by allowing redundant information to outweigh the incorrect information. Retry attempts a second attempt at an operation and is based on the premise that many faults are transient in nature. Rollback makes use of the fact that the system operation is backed up (checkpointed) to some point in its processing prior to fault detection and operation recommences from this point. Fault latency is important here because the rollback must go back far enough to avoid the effects of undetected errors that occurred before the detected error.
6. Restart. This stage occurs after the recovery of undamaged information.
   - Hot restart: resumption of all operations from the point of fault detection and is possible only if no damage has occurred.
   - Warm restart: only some of the processes can be resumed without loss.
   - Cold restart: complete reload of the system with no processes surviving.
7. Repair. In this stage, a failed component is replaced. Repair can be off-line or on-line. In off-line repair either the system will continue if the failed component is not necessary for operation or the system must be brought down to perform the repair. In on-line repair the component may be replaced immediately with a backup spare (procedure equivalent to reconfiguration) or operation may continue without the component (for example, masking redundancy or graceful degradation). With on-line repair system operation is not interrupted.
8. Reintegration. In this stage the repaired module must be reintegrated into the system. For on-line repair, reintegration must be performed without interrupting system operation.

Appendix II: Classes of Reliability Techniques

Reliability techniques can be classified in the following areas [2]:

Fault avoidance
Fault detection
Masking redundancy
Dynamic redundancy

Non-redundant systems are fault intolerant and, to achieve reliability, generally use fault avoidance techniques. Redundant systems generally use fault detection, masking redundancy, and dynamic redundancy to automate one or more of the stages of fault handling. Fault tolerance is achieved through the use of masking redundancy and dynamic redundancy techniques.

Table A3 summarizes the 4 classes of reliability techniques based on their mode of implementation – hardware and software – and a brief description of these techniques follows.

Table A3: Reliability Techniques - Hardware and Software

Class	Hardware Techniques	Software Techniques
Fault avoidance	Quality changes Component integration level	Software engineering - Modularity
Fault detection	Duplication Error detection codes Self-checking and fail-safe logic Watchdog timers and timeouts Consistency and capability checks Processor monitoring	Program monitoring Watchdog timers and timeouts
Masking redundancy	Error correcting codes Masking logic	Algorithm construction
Dynamic redundancy	Reconfigurable duplication Backup sparing Graceful degradation Reconfiguration Recovery	Forward error recovery Backward recovery - retry - checkpointing - journaling - recovery blocks

Error detection codes. Systematic applications of redundancy to information.

Self-checking circuits. Self-checking circuit design is based on the premise that the circuit inputs are already encoded in some code and the circuit outputs are also to be encoded. The inputs and outputs are not necessarily in the same code. The following definitions are based on this premise.

Self-Testing. If, for every fault from a prescribed set, the circuit produces a non-code output for at least one code input.

Totally Self-Checking. If, not only is the circuit self-testing but is also fault secure – that is, if, for every fault from a prescribed set, the circuit never produces an incorrect output for code inputs.

Fail-safe circuits. If, for every fault from a prescribed set, any input produces a *safe* output – that is, one of a preferred set of erroneous outputs.

Bus timeouts. Based on the principle that some operations should take no more than a certain maximum time to complete. time limits are set for certain responses required by the bus protocol. Thus, when one device (e.g., master) requires a response from another device (e.g., slave), a failure to respond in time indicates a possible failure. Timeouts are different from watch-dog timers in that they provide a finer check of control flow.

Consistency checking. Verifies that the intermediate or final results are reasonable, either on an absolute basis (fixed text) or as a simple function of the inputs used to derive the results. Hardware implementations include address checking, op-code checking and arithmetic operation checking. Software implementations include range checks. Memory implementations: utilize a memory in which the parity bit on any word can be arbitrarily set for either parity sense (odd or even). In practice, data words would use odd parity and instruction words even parity. In addition to parity errors, addressing errors and programming errors are likely to be discovered.

Capability checking. Usually part of the operating system, but may be implemented in hardware. Access to objects is limited to users with the proper authorization. Objects include memory segments and I/O devices. Users might be processes or independent physical processors in a system. The memory mapping mechanism of virtual address machines is a common means of checking access privileges. In addition to error detection, this technique provides some fault isolation by locking out isolated users.

Backup sparing. Some means of failure detection is used to trigger the replacement of a failed on-line unit with a spare. The detection means can be internal (self-test or self-checking) or external (timer, parity check, reasonability check) or some combination of internal and external checks. Switch complexity and effectiveness of the failure-detection techniques used are important factors. One widely used application of spares switching is in systems that are bit- or byte- sliced, for example, memories physically assembled from a set of bit planes and ALUs made from ALU byte slices.

Retry. These techniques are the fastest form of error recovery, and conceptually the simplest. They depend upon detection of an error as soon as it occurs, and immediately after detection the necessary repairs are effected. If the error is transient the repair action is to pause long enough for the transient to die away. If there is a hard failure, the system is reconfigured. The operation affected by the error is then retried, this necessitates knowing what the system state was immediately before the operation was first attempted. If the attempted operation had irrevocably modified some data, the retry will be unsuccessful, especially if the failure itself caused a spurious (and undiscovered) modification. These techniques are most commonly used for tolerating transient errors.

Checkpointing. In checkpointing, some subset of the system state is saved at specific points (the checkpoints) during process execution. The information to be stored is the subset of the system state (data, programs, machine state) that is necessary for the continued successful execution and completion of the process past the checkpoint, and that is not backed up by other means. Rollback is part of the actual recovery process and occurs after the repair (for example, by reconfiguration) of the physical damage that caused the detected error (or after the transient causing the error dies out). The rollback consists of resetting the system and process state to the state stored at the latest checkpoint. Hence, the only loss is the computation time between the checkpoint and the rollback, plus any data received during that interval that cannot be recreated.

Journaling. Simplest and least efficient of the software backward error recovery techniques; it requires the longest time to recover the state attained before an error.

A copy of the initial data (database, disk, file) is stored as the process begins. As the process executes, it makes a record of all transactions that affect the data. The, if the process fails, its effect can be recreated by running a copy of the backup data through the transactions a second time (after failures have been repaired). The recovery takes the same amount of time as the initial attempt. Journaling is better than a complete restart because it eliminates the loss of information involved in a restart.