Dependable Computing

A Multilevel Approach

Behrooz Parhami

University of California, Santa Barbara

parhami@ece.ucsb.edu
http://www.ece.ucsb.edu/~parhami

This is a draft of the forthcoming book
Dependable Computing: A Multilevel Approach,
by Behrooz Parhami, publisher TBD
ISBN TBD; Call number TBD

Dedication

To my academic mentors of long ago:

Professor Robert Allen Short (1927-2003),
who taught me digital systems theory
and encouraged me to publish my first research paper on
stochastic automata and reliable sequential machines,

and

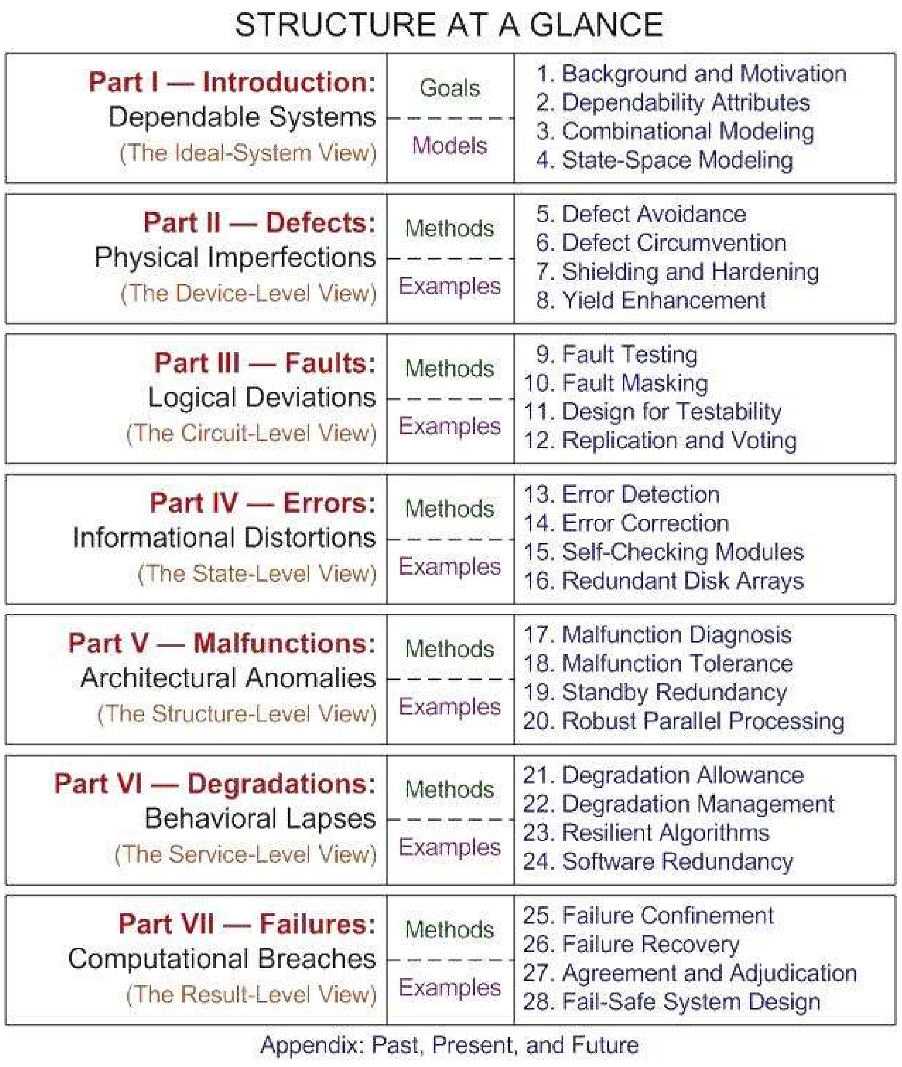
Professor Algirdas Antanas Avižienis (1932-)
who provided me with a comprehensive overview
of the dependable computing discipline
and oversaw my maturation as a researcher.

About the Cover

The cover design shown is a placeholder. It will be replaced by the actual cover image once the design becomes available. The two elements in this image convey the ideas that computer system dependability is a weakest-link phenomenon and that modularization & redundancy can be employed, in a manner not unlike the practice in structural engineering, to prevent failures or to limit their impact.

Structure at a Glance

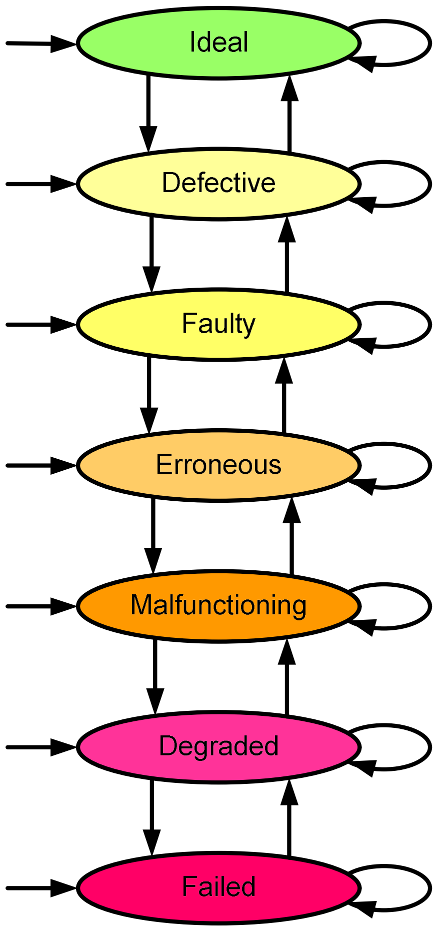
The multilevel model on the right of the following table is shown to emphasize its influence on the structure of this book; the model is explained in Chapter 1 (Section 1.4).



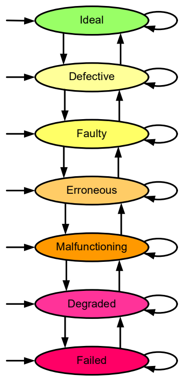
STRUCTURE AT A GLANCE

Part I — Introduction: Dependable Systems (The Ideal-System View)	Goals / Models	1. Background and Motivation 2. Dependability Attributes 3. Combinational Modeling 4. State-Space Modeling
Part II — Defects: Physical Imperfections (The Device-Level View)	Methods / Examples	5. Defect Avoidance 6. Defect Circumvention 7. Shielding and Hardening 8. Yield Enhancement
Part III — Faults: Logical Deviations (The Circuit-Level View)	Methods / Examples	9. Fault Testing 10. Fault Masking 11. Design for Testability 12. Replication and Voting
Part IV — Errors: Informational Distortions (The State-Level View)	Methods / Examples	13. Error Detection 14. Error Correction 15. Self-Checking Modules 16. Redundant Disk Arrays
Part V — Malfunctions: Architectural Anomalies (The Structure-Level View)	Methods / Examples	17. Malfunction Diagnosis 18. Malfunction Tolerance 19. Standby Redundancy 20. Robust Parallel Processing
Part VI — Degradations: Behavioral Lapses (The Service-Level View)	Methods / Examples	21. Degradation Allowance 22. Degradation Management 23. Resilient Algorithms 24. Software Redundancy
Part VII — Failures: Computational Breaches (The Result-Level View)	Methods / Examples	25. Failure Confinement 26. Failure Recovery 27. Agreement and Adjudication 28. Fail-Safe System Design

Appendix: Past, Present, and Future



VI Degradations: Behavioral Lapses



"Junk is the ultimate merchandise. The junk merchant does not sell his product to the consumer, he sells the consumer to the product. He does not improve and simplify his merchandise, he degrades and simplifies the client."

William S. Burroughs

"I will permit no man to narrow and degrade my soul by making me hate him."

Booker T. Washington

Chapters in This Part

We have defined a dependable computer system as one that produces trustworthy and timely results. Neither trustworthiness nor timeliness, however, is a binary (all or none) attribute: results may be incomplete or inaccurate, rather than missing or completely wrong, and they may be tardy enough to cause some inconvenience, without being totally useless or obsolete. Thus, various levels of inaccuracy, incompleteness, and tardiness may be distinguished and those that fall within certain margins might be viewed as degradations rather than failures. The first challenge in designing gracefully degrading systems is in mechanisms that allow degradations to occur without violating performance or safety requirements. The next challenge is to manage module switch-outs and switch-ins, as malfunctions are diagnosed and as the affected modules return to service following repair or recovery. We conclude this part by considering two specific examples of degradation allowance via resilient algorithms and degradation management by means of software redundancy.

21 Degradation Allowance

"A hurtful act is the transference to others of the degradation which we bear in ourselves."

Simone Weil

"My voice had a long, nonstop career. It deserves to be put to bed with quiet and dignity, not yanked out every once in a while to see if it can still do what it used to do. It can't."

Beverly Sills

Topics in This Chapter

The quotation "eighty percent of success is showing up," from humorist Woody Allen, can be rephrased for fail-soft systems as "eighty percent of not failing is degradation allowance." This is because malfunctions do not automatically lead to degradation: they may engender an abrupt transition to failure. In other words, providing mechanisms to allow operation in degraded mode is the primary challenge in implementing fail-soft computer systems. For a malfunction to be noncatastrophic, its identification must be quick and the module's internal state and associated data must be fully recoverable. Stable storage, checkpointing , and rollback are some of the techniques at our disposal for this purpose.

21.1 Graceful Degradation

A dependable computer system produces trustworthy and timely results. In reality, neither trustworthiness nor timeliness is a binary, all-or-none, attribute. For example, results may be incomplete or inaccurate, rather than totally missing or completely wrong, and they may be tardy enough to cause some inconvenience, without being totally useless or obsolete. Thus, various levels of inaccuracy, incompleteness, and tardiness can be distinguished and those that fall below particular thresholds might be viewed as degradations rather than failures. A system that is capable of operating in such intermediate states between fully operational and totally failed is characterized as *gracefully degradable*, *gracefully degrading*, or *fail-soft*. The noun form referring to the pertinent system attribute is *graceful degradation*.

Degradations occur in many different ways. A byte-sliced arithmetic/logic unit might lose precision if a malfunctioning slice is bypassed through reconfiguration (inaccuracy). A dual-processor real-time system with one malfunctioning unit might choose to ignore less critical computations in order to keep up with the demands of time-critical events (incompleteness). A malfunctioning disk drive within a transaction processing system can effectively slow down the system's response (tardiness). These are all instances of degraded performance. In this broader sense, performance is quite difficult to define, but Meyer [Meye80] does an admirable job:

> "Evaluations of computer performance and computer reliability are each concerned, in part, with the important question of computer system 'effectiveness'. [Therefore,] performance evaluations (of the fault-free system) will generally not suffice since structural changes, due to faults, may be the cause of degraded performance. By the same token, traditional views of reliability (probability of success, mean time to failure, etc.) no longer suffice since 'success' can take on various meanings and, in particular, it need not be identified with 'absence of system failure'."

Remember that in the quoted text above, faults/failures correspond to malfunctions in our terminology. It was the concerns cited above that led to the definition of performability (see Section 2.4) as a composite measure that encompasses performance and reliability and that constitutes a proper generalization of both notions.

Graceful degradation isn't a foregone conclusion when a system has resource redundancy in the form of multiple processors, extra memory banks, parallel interconnecting buses, and the like. Rather, active provisions are required to ensure that degradation, rather than total interruption, of service will result upon module malfunctions. The title “Degradation Allowance” for this chapter is intended to drive home the point that degradation must be explicitly provided for in the design of a system.

Example 21.1: Degradation allowance is not automatic Describe a system that has more resources of a particular kind than absolutely needed but that cannot gracefully degrade when even one of those resources become unavailable.

Solution: Most automobiles have 4 wheels. In theory, a vehicle can operate with 3 wheels; in fact, a variety of 3-wheeled autos exist. However, an ordinary 4-wheeled vehicle cannot operate if one of the wheels becomes unavailable, because the design of 3-wheeled vehicles is quite different from 4-wheeled ones.

Among the prerequisites for graceful degradation are quick diagnosis of isolated malfunctions, effective removal and quarantine of malfunctioning elements, on-line repair (preferably via hot-pluggable modules), and avoidance of catastrophic malfunctions. The issues surrounding degradation management, that is, adaptation to resource loss via task prioritization and load redistribution, monitoring of system operation in degraded mode, returning the system to intact or less degraded state at the earliest opportunity, and resuming normal operation when possible, will be discussed in Chapter 22.

On-line and off-line repairs, and their impacts on system operation and performance are depicted in Fig. 21-fsoft. *On-line repair* is accomplished via the removal/replacement of affected modules in a way that does not disrupt the operation of the remaining system parts. *Off-line repair* involves shutting down the entire system while affected modules are removed and their replacements are plugged in. Note that with on-line repair, it may be possible to avoid system shut-down altogether, thus improving both availability and performability of the system.

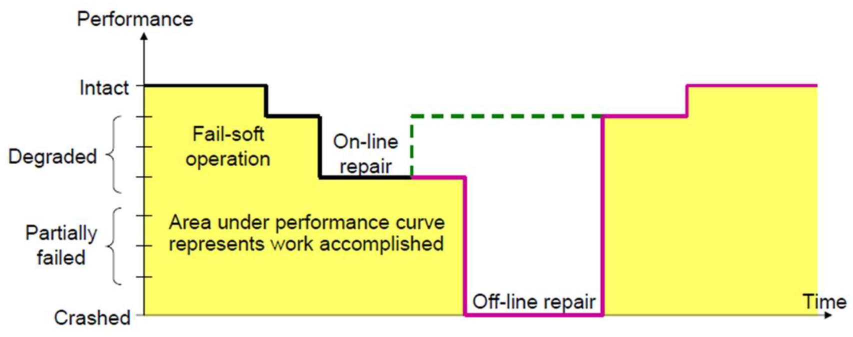


Fig. 21.fsoft A fail-soft system with possible on-line repair.

21.2 Diagnosis, Isolation, and Repair

The first step in allowing degradations is to correctly diagnose a malfunction. Removing a malfunctioning unit is done by updating the system resource tables within the operating system and, perhaps, via physical isolation (see Fig. 18.isol, for example) to ensure that the rest of the system is not affected by improper or random behavior on the part of the logically removed unit.

Next, a working configuration must be created that includes only properly functioning units. Such a working configuration would exclude processors, channels, controllers, and I/O elements (such as sensors) that have been identified as malfunctioning. Other examples of resources that might be excluded are bad tracks on disk, garbled files, and noisy communication channels. Additionally, virtual address remapping can be used to remove parts of memory from use. In the case of a malfunctioning cache memory, one might bypass the cache altogether or use a more restricted mapping scheme that exposes only the properly functioning part.

The final considerations before resuming disrupted processes include the recovery of state information from removed units, if possible, initializing any new resource that has been brought on-line, and reactivating processes via rollback or restart.

When, at some future time, the removed units are to be returned to service (say, after completion of repair or upon verification that the malfunction resulted from transient rather than permanent conditions), the steps outlined above may have to be repeated.

21.3 Stable Storage

A storage device or system is stable if it can never lose its contents under normal operating conditions. This kind of permanence is lacking in certain storage devices, such as register files and SRAM/DRAM chips, unless the system is provided with battery backup for a time duration long enough to save the contents of volatile memories, such as a disk cache, on a more permanent medium. Until recently, use of disk memory was the main way of realizing stable storage in computing devices, but now there are other options such as flash memory and magnetoresistive RAM. Combined stability and reliability can be provided for data via RAID-like methods.

Malfunction tolerance would become much easier if affected moduled simply stopped functioning, rather than engage in arbitrary behavior that may be disruptive to the rest of the system. Unpredicatable or Byzantine malfunctions are notoriously difficult to handle. Thus, we are motivated to seek methods of designing modules that behave in a malfunction-stop manner.

Given access to a reliable stable storage, along with its controlling s-process and (approximately) synchronized clocks, we can implement a k-malfunction-stop module from $k + 1$ units that can perform the same function. These units do not have to be identical. Here is how the s-process decides whether the high-level modules has stopped:

Algorithm 21.mstop Behavior of s-process for a k-malfunction-stop module

```
Input: Bag $R$ of received requests with appropriate timestamps
Output: possible setting of the variable $stop$ to TRUE
if $(|R| = k + 1) \wedge (\neg stop) \wedge$ (all requests are identical and from different sources)
then    if the request is a write
        then perform the write operation in stable storage
        else if the request is a read, send the value to all processors
        endif
else    set the variable $stop$ in stable storage to TRUE
endif
```

21.4 Process and Data Recovery

The simplest recovery scheme is restart, in which case the partially completed actions during the unsuccessful execution of a transaction must be undone. One way to achieve this end is by using logs, which form the main subject of this secion. Another way is through the use of a method known as shadow paging. Note, however, that recovery with restart may be impossible in systems that operate in real time, performing actions that cannot be undone. Examples abound in the process control domain and in space applications. Such actions must be compensated for as part of the degradation management strategy.

The use of recovery logs has been studied primarily in connection with database management systems. Logs contain sufficient information to allow the restoration of our system to a recent consistent state. They maintain information about the changes made to the data by various transactions. A previously backed up copy of the data is typically restored, followed by reapplying the operations of committed transactions, up to the time of failure, found in the recovery log.

A common protocol for recovery logs is write-ahead logging (WAL). Log entries can be of two kinds, undo-type entries and redo-type entries, with some logs containing both kinds of entries. Undo-type log entries hold old data values, so that the values can be restored if needed. Redo-type entries hold new data values to be used in case of operation retry. The main idea of write-ahead logging is that no changes should be made before the necessary log entries are created and saved. In this way, we are guaranteed to have the proper record for all changes should recovery be required. [More elaboration needed.]

Logs are sequential append-only files. The relative inefficiency of a sequential structure isn't a major concern, given that logs are rarely used to effect recovery.

An efficient scheme for using recovery logs is via deferred updates. In this method, any changes to the data are not written directly to stable storage. Rather, the data affected by updates is cached in main memory; only after all changes associated with a transaction have been applied will the data be written to stable storage (preceded, of course, by saving the requisite log entries). In this way, access to the typically slow stable storage is minimized and performance is improved.

An example of deferred updates is shadow paging. In order to avoide in-place updates, which may create inconsistencies, any page to be modified is copied into a shadow page, which is then freely updated, given that there are no external references to it [Hitz95]. Once the page becomes ready for assuming durable status, all pages that refer to the original are updated to point to the new page. The idea is similar to the method used in old batch-processing systems in which two copies of all daily updates were maintained on separate disks, with one disk kept as back-up and the other one used as the starting point for next day's operation.

21.5 Checkpointing and Rollback

Long-running computations, whose execution times are comparable to or exceed the hardware's MTTF are not likely to complete before hardware crash necessitates a restart. This situation was a routine occurrence in early digital computers whose MTTF was measured in hours, leading to many attempts at program execution before a successful run to termination. Thus, programmers of early computers devised method for recording intermediate results and state of a computation so that after recovery from a hardware failure, they did not have to restart the computation from the very beginning. This technique came to be known as *checkpointing*. Modern digital systems have a much longer MTTF but they also execute more complex programs, some of which may run for days or even weeks. So, checkpointing is still a useful technique.

Example 21.chkpt1: Effect of checkpointing on task completion probability Suppose a computation's running time T is twice the MTTF of the machine used to execute it. Ignoring the checkpointing time overhead, compare the probability of completing the computation in $2T$ time:
a. Assuming no checkpointing.
b. Assuming checkpointing at regular intervals of $T/2$

Solution: We assume an exponential reliability formula $R = e^{-\lambda t}$, with MTTF = $1/\lambda$.
a**.** The system reliability over the time $t = T = 2/\lambda$ is $e^{-\lambda t} = e^{-2} = 0.135\,335$, which is the probability that the task completes in time T. The no-checkpointing case can be modeled as a 2-state discrete Markov chain, with time step T and states S and C, corresponding to computation start and completion. There is a single transition from S to C, with an associated probability 0.135 335, leading to the transition matrix having rows (0.864 665 0.135 335) and (0 1). Beginning in state (1 0), the state after two units T of time will be (0.747 646 0.252 354), with 0.252 354, or about 25%, being the completion probability after $2T$ time.
b. Under checkpointing at regular intervals $T/2$, the Markov chain will also have an intermediate state H, where the task is half-completed, with transition probability from S to H and from H to C being $e^{-1} = 0.367\,879$. The unit time in this case is $T/2 = 1/\lambda$. Beginning with state (1 0 0), the system will go through states (0.632 121 0.367 879 0), (0.399 577 0.465 088 0.135 335), and (0.252 581 0.440 988 0.306 431) at times $T/2$, T, and $3T/2$, before ending up in state (0.159 662 0.371 677 0.468 661) at time $2T$, with 0.468 661, or about 47%, being the completion probability after $2T$ time.

Checkpoints are placed at convenient locations along the course of a computation, not necessarily at equal intervals, but we often assume equal checkpointing intervals for the sake of analytical tractability. Checkpointing entails some overhead consisting of the program instructions needed to save the state and partial results and those needed to recover from failure by reloading a previously computation state.

We see from Example 21.chkpt1 that not using checkpoints may lead to a small probability of task completion within a specified time period. On the other hand, using too many checkpoints may be counterproductive, given the associated overhead. Thus, there may be an optimal configuration that leads to the best expected completion time. We will discuss optimal checkpointing in Section 21.6.

Once a system malfunction disrupts an in-progress computation, the comutation must be rolled back to its latest checkpoint. Thus, checkpointing and *rollback* go hand in hand in recovering from system malfunctions. Process rollback or restart creates no problem for tasks that perform I/O only at their start and termination. Referring to Fig. 21.chkpt1, recovery from the detected malfunction, which affects processes 2 and 6, is readily accomplished by rolling back process 2 to its checkpoint 2 and restarting process 6. Interacting processes require more care during rollback, as we will see shortly.

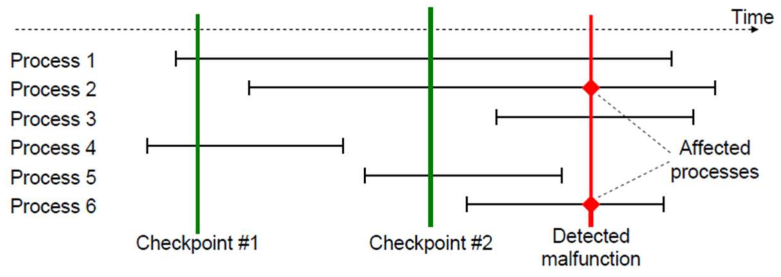


Fig. 21.chkpt1 Checkpointing for multiple independent, noncommunicating processes.

Example 21.chkpt2: Data checkpointing Consider data objects stored on a primary site and k backup sites. With appropriate design, such a scheme will be k-malfunction-tolerant. Each data access request is sent to the primary site, where a read request is honored immediately. A write request triggers a chain of events consisting of the primary site sending update transactions to the backup sites and acknowledging the original write request only after acknowledgements have been received from all backup sites. Argue that increasing the number k of backup sites beyond a certain point may be counterproductive in that it may lead to lower data availability.

Solution: When the primary site is up and running, data is available. Data becomes unavailable in three system state: (1) Recovery state, in which the primary site has malfunctioned and the system is in the process of "electing" a new primary site from among the backup sites. (2) Checkpoint state, in which the primary site is performing data backup. (3) Idle state, in which all sites, primary and backup, are unavailable. As the number of backups increases, the system will spend more time in states 1 and 2 and less time in state 3, so there may be an optimal choice for the number k of backup sites. Analysis by Huang and Jalote [Huan89], with a number of specific assumptions, indicates that data availability goes up from the nonredundant figure of 0.922 to 0.987 for $k = 1$, 0.996 for $k = 2$, 0.997 for $k = 4$, beyond which there is no improvement.

The checkpointing scheme depicted in Fig. chkpt1 is synchronous in the sense of all running processes doing their checkpointing at the same time, perhaps dictated by a central system authority. In large, or loosely coupled systems, it is more likely for processes to schedule their checkpoints independently based on their own state of computataion and when checkpointing is most convenient. These asynchronous checkpoints, depicted in Fig. 21.chkpt2, do not create any difficulty if the processes are independent and non-interacting. Upon a detected malfunction, all affected processes are notified, with each process independently rolling back to its latest checkpoint.

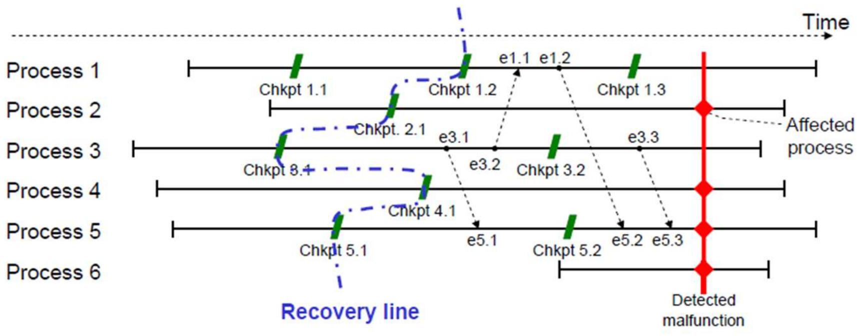


Fig. 21.chkpt2 Asynchronous checkpointing for multiple independent communicating processes.

If the independent processes interact, however, as shown by the dashed arrows representing message exchanges in Fig. 21.chkpt2, complications might arise. For example, if a malfunction is detected at the instant shown, Process 2 is rolled back to its latest checkpoint 2.1, with no other action necessary, given that the process has had no interaction with other processes since that checkpoint. On the other hand, rolling back Process 5 to its latest checkpoint 5.2 creates the problem that the process will be missing events 5.2 and 5.3, corresponding to messages arriving from Processes 1 and 3, respectively, upon its repeated execution. This is because Processes 1 and 3 have progressed in their execution (they were not affected by the malfunction) and will thus not resend those two messages.

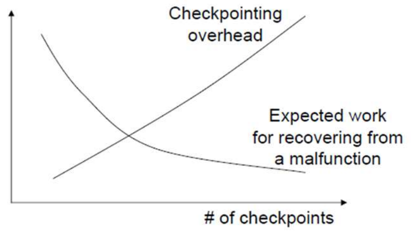
One way of dealing with such dependencies is to also roll back certain interacting processes when a given process is rolled back. There is a possibility of a chain reaction that could lead to all processes having to restart from their very beginning. In general, we need to identify a recovery line, or a consistent set of checkpoint, whose selection would lead to correct re-execution of all processes. This is a nontrivial problem. An alternative approach is to create stable logs of all interprocess communications, so that a process can consult them upon re-execution.

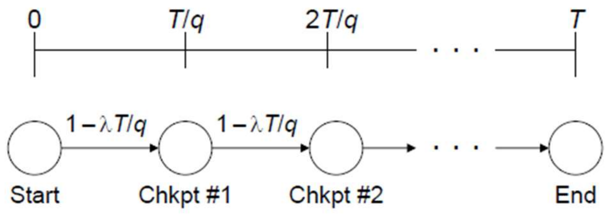
21.6 Optimal Checkpoint Insertion

There is a clear tradeoff in checkpoint insertion. Too few checkpoints lead to long rollback and waste of computational resources in the event of a malfunction. Too many checkpoints lead to excessive time overhead. These two opposing trends are depicted in Fig. 21.optcp-a. As in many other engineering problems, there is often a happy medium that can be found analytically or experimentally.

Example 21.optcp1: Optimal number of checkpoints Consider a computation of running time T divided into q segments, so that there are $q - 1$ checkpoints. Let λ denote the malfunction rate and T_{cp} be the time needed to capture a checkpoint snapshot. Determine an optimal value for q that minimizes the total running time.

Solution: The computation can be viewed as having $q + 1$ states corresponding to the fraction i/q of it completed, for $i = 0$ to q. From each state to the next one in sequence, the transition probability over the time step T/q is $1 - \lambda T/q$, as depicted in the discrete-time Markov chain of Fig. 21.optcp-b. By using the latter linear approximation, we have implicitly assumed that $T/q << 1/\lambda$. We can easily derive $T_{total} = T/(1 - \lambda T/q) + (q - 1)T_{cp} = T + \lambda T^2/(q - \lambda T) + (q - 1)T_{cp}$. Differentiating T_{total} with respect to q and equating with 0 yields $q^{opt} = T(\lambda + \sqrt{\lambda/T_{cp}})$. For example, if we have $T = 200$ hr, $\lambda = 0.01$ / hr, and $T_{cp} = 1/8$ hr, we get $q^{opt} = 59$ and $T_{total} \approx 211$ hr.





(a) Opposing trends (b) Determination of optimal checkpoint frequency

Fig. 21.optcp Tradeoffs in checkpoint insertion.

Example 21.optcp2: Optimal checkpointing in transaction processing Thus far, we have incorporated rollback time into the checkpointing overhead and assumed the composite overhead to be a constant. In some application contexts, such as transaction processing, it is possible that the rollback time increases (say, linearly) with the time interval over which the computation is to be rolled back. Representing the checkpointing period by P_{cp} and the rollback overhead by $a + bx$, where x $(0 < x < P_{\text{cp}})$ is the malfunction time within a checkpointing period and b is a relatively small constant of proportionality that accounts for the time needed for certain actions such as updates, find the optimal value of P_{cp}.

Solution: The expected rollback time due to a malfunction in the time interval $[0, P_{\text{cp}}]$ is found by integrating $(a + bx)\lambda dx$ over $[0, P_{\text{cp}}]$, yielding $T_{\text{rb}} = \lambda P_{\text{cp}}(a + bP_{\text{cp}}/2)$. We can choose P_{cp} to minimize the relative checkpointing overhead $O = (T_{\text{cp}} + T_{\text{rb}})/P_{\text{cp}} = T_{\text{cp}}/P_{\text{cp}} + \lambda(a + bP_{\text{cp}}/2)$ by equating dO/dP_{cp} with 0. The result is $P_{\text{cp}}^{\text{opt}} = \sqrt{2T_{\text{cp}}/(\lambda b)}$. Let us assume, for example, that $T_{\text{cp}} =$ 16 min and $\lambda = 0.0005/\text{min}$ (corresponding to an MTTF of 33.3 hr). Then, the optimal checkpointing period is $P_{\text{cp}}^{\text{opt}} = 800$ min = 13.3 hr. If by using a faster memory for saving the checkpoint snapshots we can reduce T_{cp} to 1 min (a factor of 16 reduction), the optimal checkpointing period goes down by a factor of 4 to become $P_{\text{cp}}^{\text{opt}} = 200$ min = 3.3 hr.

Problems

21.1 Checkpointing for long computations

In the optimal checkpointing example for long computations in Section 21.6, q = 59 (insertion of 58 checkpoints) was determined to be optimal. Because each checkpoint entails 1/8 hr of overhead, this accounts for a tad over 7 hr of the running time extension from 200 hr to 211 hr. What do you think is the source of the additional 4 hr in estimated additional running time?

21.2 Optimal checkpointing

We discussed optimal checkpointing under the assumption that time overhead per checkpoint is a constant. Suppose that checkpointing overhead is a linear function of checkpointing period, that is, the longer the time interval between checkpoints, the more information there is to store and the longer the time overhead for each checkpoint. Present an analysis of optimal checkpointing in this case, stating all your assumptions.

21.3 Effect of checkpointing on task completion probability

Continue Example 21.chkpt1 with checkpointing at regular intervals of $T/3$ and $T/4$. Discuss.

21.4 Discrete optimal checkpointing

In discussing optimal checkpointing, we assumed that we can insert a checkpoint at any desired point along the course of the computation with the same overhead T_{cp}. In reality, a computation may have a finite set of feasible times $t_1 < t_2 < . . . < t_m$ where checkpoints can realistically be placed and they have corresponding checkpointing overheads T_1, T_2, … , T_m, with each $T_i > 0$ being a known constant. Outline a procedure for finding an optimal subset of checkpoints from among the m choices. Is it possible for the optimal number of checkpoints to be 0? What about m checkpoints being optimal?

21.x Title

Intro

a. xxx

b. xxx

c. xxx

d. xxx

References and Further Readings

[Hitz95] Hitz, D., J. Lau, and M. Malcolm, "File System Design for an NFS File Server Appliance," Network Appliance Corp. Technical Report 3002, Rev. C, March 1995. https://atg.netapp.com/wp-content/uploads/2000/01/file-system-design.pdf

[Huan89] Huang, Y. and P. Jalote, "Analytic Models for the Primary Site Approach for Fault Tolerance," *Acta Informatica*, Vol. 26, pp. 543-557, 1989.

[Jalo94] Jalote, P., *Fault Tolerance in Distributed Systems*, Prentice Hall, 1994.

[Kris93] Krishna, C. M. and A. D. Singh, "Reliability of Checkpointed Real-Time Systems Using Time Redundancy," *IEEE Trans. Reliability*, Vol. 42, No. 3, pp. 427-435, September 1993.

[Prad94] Pradhan, D. K. and N. H. Vaidya, "Roll-Forward Checkpointing Scheme: A Novel Fault-Tolerant Architecture," *IEEE Trans. Computers*, Vol. 43, No. 10, pp. 1163-1174, October 1994.

[Schl83] Schlichting, R. D. and F. B. Schneider, "Fail Stop Processors: An Approach to Designing Fault-Tolerant Computing Systems," *ACM Trans. Computer Systems*, Vol. 1, No. 3, pp. 222-238, August 1983.

[Sour19] Souravlas, S. and A. Sifaleras, "Trends in Data Replication Strategies: A Survey," *Int'l J. Parallel, Emergent and Distributed Systems*, Vol. 34, No. 2, pp. 222-239, 2019.

22 Degradation Management

“Most of us don't think, we just occasionally rearrange our prejudices.”

Frank Knox

“The communications links were constantly tested by means of sending filler messages. At the time of the false alerts, these filler messages had the same form as attack messages, but with a zero filled in for the number of missiles detected. The system did not use any of the standard error correction or detection schemes for these messages. When the chip failed, the system started filling in the ‘missiles detected’ field with random digits.”

A. Borning, Computer System Reliability and Nuclear War

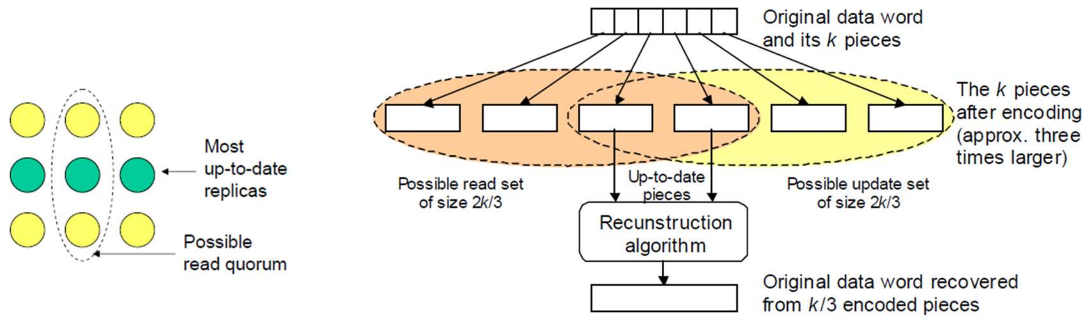
Topics in This Chapter

Assuming that malfunctioning units are correctly identified, offending subsystems are isolated, reconfiguration is appropriately performed, and recovery processes are successfully executed, several other steps are still necessary for the system to be able to function in a degraded mode. Tasks must be prioritized and those that cannot be executed with the limited resources disabled or removed. Similarly, adaptation in the opposite direction is required when previously malfunctioning resources are returned to service following checkout or repair. Thus, degradation management aims to ensure the smooth functioning of the system under resource fluctuations (losses and reactivations).

22.1 Data Distribution Methods

Reliable data storage requires that the availability and integrity of data not be dependent on the health of any one site. To ensure this property, data may be replicated at different sites, or it may be dispersed so that losing pieces of the data does not preclude its accurate reconstruction.

As discussed earlier, data replication can place a large burden on the system, perhaps even leading in the extreme to the nullification of its advantages. The need for updating all replicas before proceeding with further actions is one such burden. One way around this problem is the establishment of read and write quorums. Consider the example in Fig. 22.integ-a, where the 9 data replicas are logically viewed as forming a 2D array. If a read operation is defined as accessing the 3 replicas in any one column (the *read quorum*) and selecting the replica with the latest time-stamp, then the system can safely proceed after updating any 3 replicas in one row (the *write quorum*). Because the read and write quorums intersect in all cases, there is never a danger of using stale data and the system is never bogged down if one or two replicas are out of date or unavailable.


Most up-to-date replicas
Possible read quorum
Original data word and its k pieces
The k pieces after encoding (approx. three times larger)
Possible read set of size $2k/3$
Up-to-date pieces
Possible update set of size $2k/3$
Recunstruction algorithm
Original data word recovered from $k/3$ encoded pieces

(a) Replication with quorums (b) Data dispersion example

Fig. 22.integ Ensuring data integrity and availability via replication and dispersion.

A similar result can be achieved via the data dispersion scheme of Fig. 22.integ-b, where a piece of data is divided into 6 pieces, which are then encoded in such a way that any two of the encoded pieces suffice for reconstructing the original data. Such an encoding requires 3-fold redundancy and is thus comparable to 3-way replication in terms of storage overhead. Now, if we define read and write quorums to comprise any 4 of the 6 encoded pieces, gaining access to any 4 pieces would be sufficient for reconstructing the data, because the 4 pieces are bound to have at least 2 pieces that are up to date (have the latest time stamp). This scheme too eases the burden of updating the data copies by not requiring that every single piece be kept up to date at all times.

22.2 Multiphase Commit Protocols

Consider the following puzzle known as “the two generals problem.” The setting is as follows. Two generals lead the two divisions of an army camped on the mountains on each side of an enemy-occupied valley. The two army divisions can communicate only via messengers. Messengers, who are loyal and highly reliable, may need an arbitrary amount of time to cross the valley and in fact may never arrive due to being captured by the enemy forces.

We need a scheme for the two generals G_1 and G_2 to agree on a common attack time, given that attack by only one division would be disastrous. Here is a possible scenario. General G_1 decides on time T and sends a messenger to inform G_2. Because G_1 will not attack unless he is certain that G_2 has received the message about his proposed attack time, G_2 sends an acknowledgment to G_1. Now, G_2 will have to make sure that G_1 has received his acknowledgment, because he knows that G_1 will not attack without it. So, G_1 must acknowledge G_2’s acknowledgment. This can go on forever, without either general ever being sure that the other general will attack at time T.

The situation above is akin to what happens at a bank’s computer system when you try to make cash withdrawal from an ATM. The ATM should not dispense cash unless it is certain that the bank’s central computer checks the account balance and adjusts it after the withdrawal. On the other hand, you will not like it is your account balance is reduced without any cash being dispensed. So, the two sides, the ATM and the bank’s database, must act in concert, either both completing the transaction or both abandoning it. Thus, the withdrawal transaction, or electronic funds transfer between two accounts, must be an atomic, all-or-none action.

A key challenge is maintaining the atomicity of such actions in the occurrence of malfunctions in various system components. In centralized systems, atomicity can be ensured via locking mechanisms. Each operation is performed in three phases:

Acquire (read or write) lock for a desired data object and operation
Perform operation while holding the lock
Release lock

One must take care to avoid deadlocks arising from circular waiting for locks.

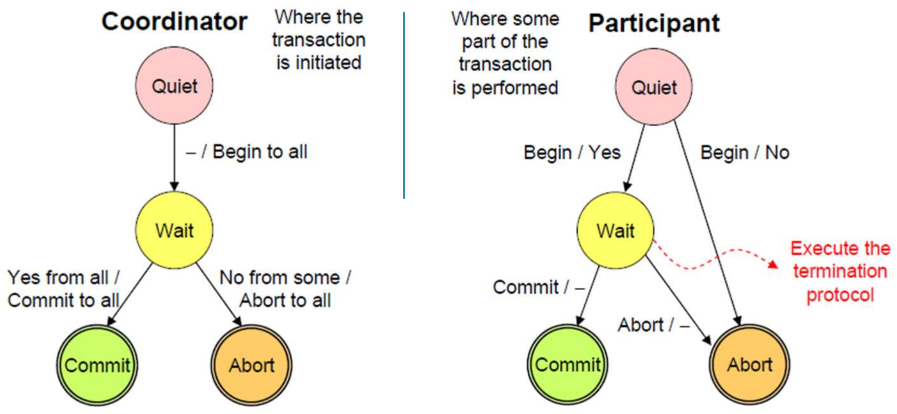

Coordinator
Where the transaction is initiated
Quiet
– / Begin to all
Wait
Yes from all / Commit to all
No from some / Abort to all
Commit
Abort
Where some part of the transaction is performed
Participant
Quiet
Begin / Yes
Begin / No
Wait
Execute the termination protocol
Commit / –
Abort / –
Commit
Abort

Fig. 22.2pcp Coordinator and participant states in the two-phase commit protocol.

An alternative to the use of locks is devising a protocol that requires cross-checking before commiting to changes arising from a transaction. The simplest such protocol is known as "the two-phase commit protocol." The protocol is executed between a coordinator and a number of participants, which have the states depicted in Fig. 22.2pcp. [Details to be supplied.]

To avoid participants being stranded in the "wait" state (e.g., when the coordinator malfunctions), a time-out scheme may be implemented.

To deal with the shortcomings of the two-phase commit, a three-phase commit protocol may be devised. As shown in Fig. 22.3pcp, an extra "prepare" state is inserted between the "wait" and "commit" states of two-phase commit. This protocol is safe from blocking, given the absence of a local state that is adjacent to both a "commit" state and an "abort" state.

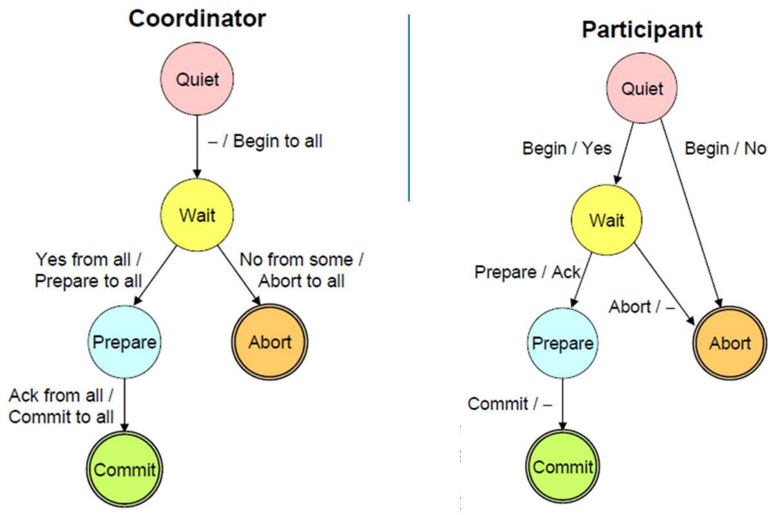


Fig. 22.3pcp Coordinator and participant states in the two-phase commit protocol.

22.3 Dependable Communication

Point-to-point messages can be protected against communication errors due to malfunctioning links or intermediate nodes through encoding, requiring receipt acknowledgments, and implementation of a time-out mechanism.

It is sometimes required that a message be reliably broadcast or multicast to a set of nodes, so that it is guaranteed to be received by all the intended nodes. One way of accomplishing reliable broadcast is to send the message along the branches of a broadcast tree, with possible repetition. In this scheme, duplicate messages can be recognized from their sequence numbers.

In order to cut down on the amount of communication during reliable broadcasting, acknowledgment messages may be piggybacked on subsequent broadcast messages. Suppose node P broadcasts a message m_1. Upon receiving m_1, node Q may tack on an acknowledgment for m_1 to its own broadcast message m_2. If a third node R did not receive m_1, it will find out about it from Q's acknowledgment and will take steps to acquire the missed message, perhaps by asking P for a retransmission.

Atomic broadcasting entails not only reliable message delivery but also requires that multiple broadcasts be received in the same order by all nodes. If we use the scheme outlined in the preceding paragraph, in-order delivery of messages will not be guaranteed, so atomic broadcasting is much more complicated.

Another form of reliable broadcasting is causal broadcast, which requires that if m_2 is sent after m_1, any message triggered by m_2 must not cause actions before those of m_1 have been completed.

22.4 Dependable Collaboration

Many distributed systems, built from COTS nodes (processors plus memory) and standard interconnects, contain sufficient resource redundancy to allow the implementation of software-based malfunction tolerance schemes. Interconnect malfunctions are dealt with by synthesizing reliable primitives for point-to-point and collective communication (broadcast, multicast, and so on), as discussed in Section 22.3. Node malfunctions are modeled differently, as illustrated in Fig. 22.malfns, with possible models reanging from benign crash malfunctions, that are fairly easy to deal with, to the arbitrary or Byzantine malfunctions, that require greater care in protocol development and much higher redundancy.

A potentially helpful resource in managing a group of cooperating nodes, that are subject to malfunctions, is a reliable group membership service. The group's membership may expand and contract owing to changing processing requirements or because of malfunctions and repairs. A reliable group membership service maintains up-to-date status information and thus supports a reliable multicast, via which a message sent by one group member is guaranteed to be received by all other members.

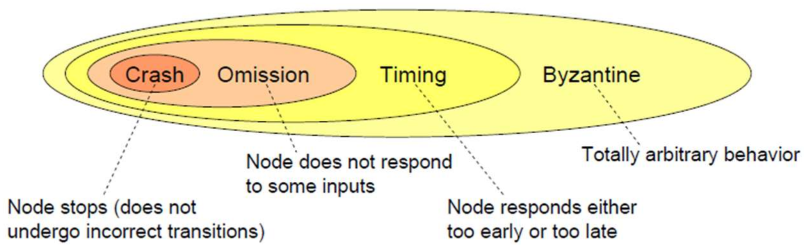

Crash
Omission
Timing
Byzantine
Node does not respond to some inputs
Totally arbitrary behavior
Node stops (does not undergo incorrect transitions)
Node responds either too early or too late

Fig. 22.malfns Node malfunctions range from benign to Byzantine.

Another recent development in the theory of distributed systems is the notion of malfunction detector, a distributed oracle tasked with monitoring system resources for signs of malfunctions. As part of its operation, a malfunction detector creates and maintains a list of suspected processes characterized by two properties: completeness (having all malfunctioning processes in the list) and accuracy (having no healthy processes). Using specialized malfunction detectors decouples the effort to detect malfunctions from that of the actual computation, leading to greater modularity. It also improves portability, because the same application can be used on a different platform is suitable malfunction detectors are available for it.

A perfect malfunction detector, having strong completeness and strong accuracy is the minimum required for interactive consistency. Strong completeness, along with eventual weak consistency are the minimum requirements for consensus [Rayn05]. [Elaboration to be added.]

22.5 Remapping and Load Balancing

When the configuration of a system changes due to the detection and removal of malfunctioning units, division of labor in ongoing computations must be reconsidered. This can be done via a remapping scheme that determines the new division of responsibilities among participating modules, or via load balancing (basic computational assignments do not change, but the loads of the removed modules are distributed among other modules). Load balancing is also used not just to accommodate lost/recovered resources due to malfunctions and repairs, but also to optimize system performance in the face of changing computational requirements.

Even in the absence of a detected malfunction and the attendant system reconfiguration, remapping of a computation to have its various pieces executed by different modules may be useful for exposing hidden malfunctions. This is because the effects of a malfunctioning module will likely be different on diverse computations, making it highly unlikely to get the same final results for the original and remapped computation.

Let us consider a remapping example for a computation that runs on a 5-cell linear array. By adding a redundant 6th cell at the end of the array (Fig. 22.remap), we can arrange for the computation to be performed in two different ways: one starting in cell 1 and another starting in cell 2 [Kwai97]. Each cell *j* + 1 can be instructed to compare the result of step *j* in the computation that it received from the cell to its left to the result of step *j* that it obtains within the second computation. A natural extension of this scheme is to provide 2 extra cells in the array and to perform three versions of the computation, with cell *j* + 2 voting on the three results obtained by cell *j* in the first computation, cell *j* + 1 in the second version, and cell *j* itself in the third version.

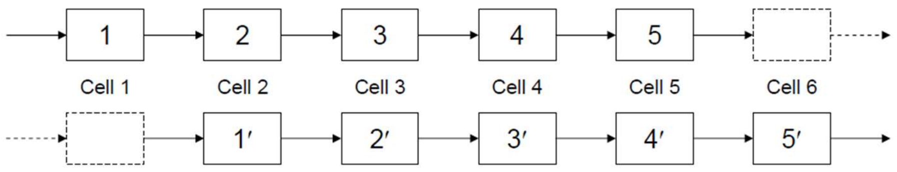


Fig. 22.remap Recomputation with shift in space.

22.6 Modeling of Degradable Systems

A gracefully degradable system typically has one ideal or intact state, multiple degraded states, and at least one failure state, as depicted in Fig. 22.degsys. To ensure that the system degrades rather than fail or come to a halt, we have to reduce the probability of malfunctions going undetected, increase the accuracy of malfunction diagnosis, make repair rates much greater than malfunction rates (typically, but keeping hot-pluggable spares), and provide sufficient safety factor in computational capacity.

In practice, besides the indirect paths to failure corresponding to resource exhaustion, as depicted in Fig. 22.gdsys, there may be direct or semidirect paths that lead to failure in a shorter amount of time (Fig. 22.paths). These faster paths to failure arise from imperfect coverage in malfunction detection or in the attendant reconfiguration to tolerate a detected malfunction. While the provision of additional spare capacity lengthens the indirect paths, it does nothing to avoid the direct paths; on the contrary, it may increase the probability of taking a direct path, given the growth in the complexity of system-level reconfiguration mechanisms. For a given coverage factor, addition of resources beyone a certain point would not be cost-effective with regard to the resulting reliability gain. This is quite similar to the effect we saw previously in standby sparing.

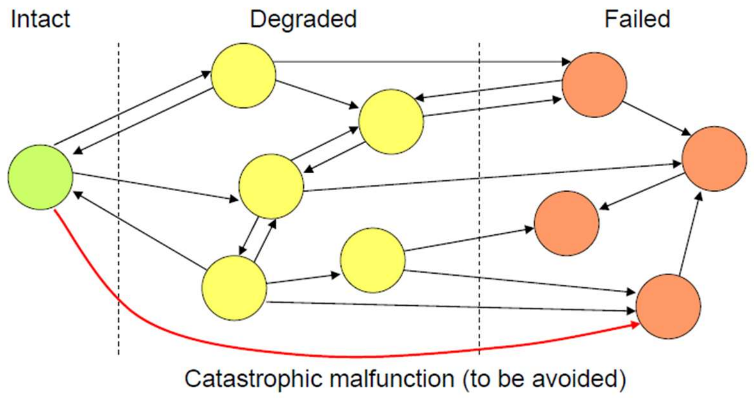


Fig. 22.gdsys State-space model for a gracefully degradable system, with multiple degradation levels and multiple failure states.

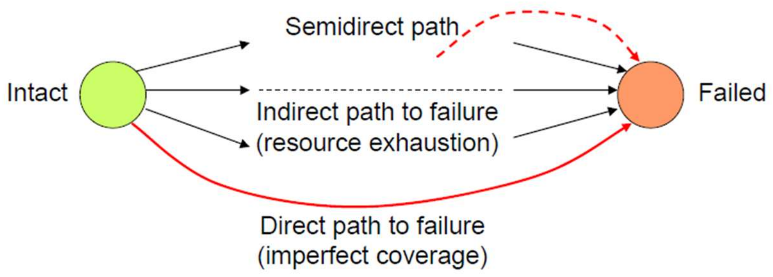


Fig. 22.paths Direct, semidirect, and indirect paths to failure in a degradable system.

Problems

22.1 Title

Intro

a. xxx

b. xxx

c. xxx

d. xxx

22.2 Soft failures in data centers

Read the paper [Sank14] and prepare a one-page, single-spaced report covering the nature of the failures considered, their frequency (e.g., single event or recurrent), their effects on down time, and possible countermeasures. Begin your report with a one-sentence summary of key findings.

22.x Title

Intro

a. xxx

b. xxx

c. xxx

d. xxx

References and Further Readings

[Jalo94] Jalote, P., *Fault Tolerance in Distributed Systems*, Prentice Hall, 1994.

[Kwai97] Kwai, D.-M. and B. Parhami, "An On-Line Fault Diagnosis Scheme for Linear Processor Arrays," *Microprocessors and Microsystems*, Vol. 20, No. 7, pp. 423-428, March 1997.

[Rabi89] Rabin, M., "Efficient Dispersal of Information for Security, Load Balancing, and Fault Tolerance, *J. ACM*, Vol. 36, No. 2, pp. 335-348, April 1989.

[Rayn05] Raynal, M., "A Short Introduction to Failure Detectors for Asynchronous Distributed Systems," *ACM SIGACT News*, Vol. 36, No. 1, pp. 53-70, March 2005.

[Sank14] Sankar, S. and S. Gurumurthi, "Soft Failures in Large Data Centers," *IEEE Computer Architecture Letters*, Vol. 13, No. 2, pp. 105-108, July-December 2014.

23 Resilient Algorithms

"Perfection is achieved, not when there is nothing more to add, but when there is nothing left to take away."

Antoine de Saint Exupery

"The catastrophic nature of some program failures, in which the program collapses suddenly and utterly, has its analogue in the metallurgical phenomenon of brittle fracture, in which the crack propagates at nearly the speed of sound."

P.W. Abrahams, The Role of Failure in Software Design

Topics in This Chapter

One approach degradation allowance is through the design of resilient algorithms that are by design insensitive to resource malfunctions. Resilient algorithms have built-in redundancy in their computations and data structures, so that when a limited number of malfunctions are experienced, the resulting errors are detected or even corrected. This approach is very attractive because it may allow the use of commercial off-the-shelf system components instead of specially designed, and rather expensive, hardware. Beginning from primitive, or building-block, algorithms for communication and other forms of collaboration, a multilevel algorithmic structure is built that can function correctly under adverse conditions.

23.1 COTS-Based Paradigms

Many of the hardware and software redundancy methods assume that we are building the entire system (or a significant part of it) from scratch. Many users of highly reliable systems, however, do not have the capability to develop such systems and thus have one of two options.

The first option is to buy dependable systems from vendors that specialize in such systems and associated support services. Here is a partial list of companies which have offered, or are now offering, fault-tolerant systems and related services:

ARM: Fault-tolerant ARM (launched in late 2006), automotive applications
Nth Generation Computing: High-availability and enterprise storage systems
Resilience Corp.: Emphasis on data security
Stratus Technologies: "The Availability Company"
Sun Microsystems: Fault-tolerant SPARC (ft-SPARC™)
Tandem Computers: An early leader, part of HP/Compaq since 1997

An alternative is to build upon commercial off-the-shelf (COTS) components and systems some protective layers that ensure dependable operation. A number of algorithm and data-structure design methods that are resilient to imperfect hardware are available.

An early experiment with the latter approach was performed in the 1970s, when Stanford University built one of two "concept systems" for fly-by-wire aircraft control, using mostly COTS components. The resulting multiprocessor, named SIFT (software-implemented fault tolerance) was meant to introduce a fault tolerance scheme that contrasted with the hardware-intensive approach of MIT's FTMP (fault-tolerant multiprocessor). The Stanford and MIT design teams strived to achieve a system failure rate goal of 10^{-9} per hour over a 10-hour flight, which is typical of avionics safety requirements. Some fundamental results on, and methods for, clock synchronization emerged from the SIFT project. To prevent errors from propagating in SIFT, processors obtained multiple copies of data from different memories over different buses (local voting).

The COTS approach to fault tolerance has some inherent limitations. Some modern microprocessors have dependability features built in: they may use parity and other codes in memory, TLB, and microcode store; they may take advantage of retry features at various levels, from bus transmissions to full instructions; they may have provide machine check facilities and registers to hold the check results. According to Avizienis, however, these features are often not documented enough to allow users to build on them, the protection provided is nonsystematic and uneven, recovery options may be limited to shutdown and restart, description of error handling is scattered among a lot of other detail, and there is no top-down view of the features and their interrelationships [Aviz97].

Manufacturers can incorporate both more advanced and new features, and at times have experimented with a number of mechanisms, but until recently, the low volume of the application base hindered commercial viability.

23.2 Robust Data Structures

Stored and transmitted data can be protected against unwanted changes through encoding, but coding does not protect the structure of the data. Consider, for example, an ordered list of numbers. Individual numbers can be protected by encoding and the set of values can be protected by a checksum; the ordering of data, however, remains unprotected with either scheme. Some protection against an inadvertent change in ordering can be provided by a weighted checksum. Another idea is to provide within the array a field that records the difference between each element and the one that follows it. A natural question at this point is whether we can devise general schemes for protecting data structures of common interest.

Let us first consider linked lists (Fig. 23.rlist). [Details to be supplied.]

Robust data structures provide fairly good protection with little design effort or run-time overhead

Audits can be performed during idle time
Reuse possibility makes the method even more effective

Robustness features to protect the structure can be combined with coding methods (such as checksums) to protect the content

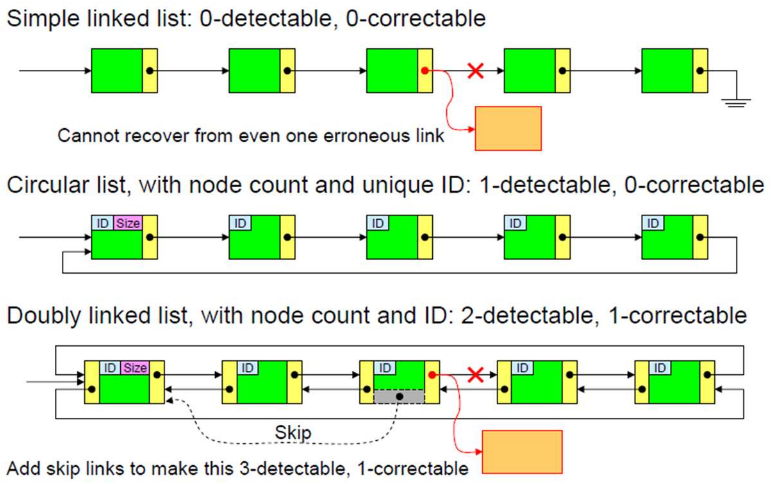

Simple linked list: 0-detectable, 0-correctable
Cannot recover from even one erroneous link
Circular list, with node count and unique ID: 1-detectable, 0-correctable
ID
Size
Doubly linked list, with node count and ID: 2-detectable, 1-correctable
Skip
Add skip links to make this 3-detectable, 1-correctable

Fig. 23.rlist Robustness of linked lists.

Other robust data structures of interest include trees, FIFOs, stacks or LIFOs, heaps, and queues. In general, a linked data structure is 2-detectable and 1-correctable iff the link network is 2-connected.

Binary trees have weak connectivity and are thus quite vulnerable to corrupted or missing links. One way to strengthen the connectivity of trees is to add parent links and/or threads (links that connect a node to higher-level nodes). An example of a thread link is shown in Fig. 23.rtree. Threads can be added with little overhead by taking advantage of unused leaf links (one bit in every node can be used to identify leaves, thus freeing their link fields for other uses).

Adding redundancy to data structures has three types of cost:

- Storage requirements for the additional information
- Slightly more difficult updating procedures
- Time overhead for periodic checking of structural integrity

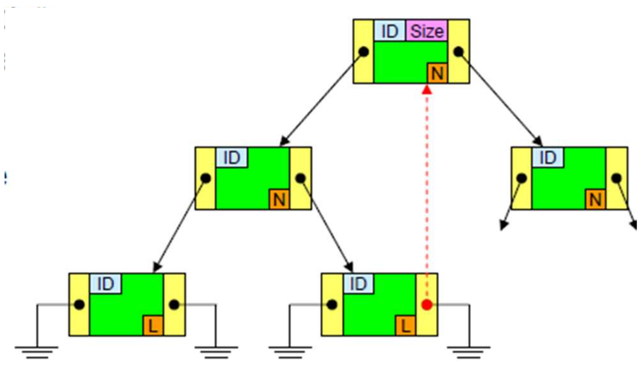


Fig. 23.rtree Improving the robustness of a binary tree.

23.3 Data Diversity and Fusion

Alternate formulations of the same information (input re-expression) is known as data diversity. For example, a rectangle can be specified by its two sides x and y, by the length z of its diameters and the angle α between them, or by the radii r and R of its inscribed and circumscribed circles. As shown in Fig. 23.ddiv, diverse representations lead to diverse calculations, thus reducing the chance of encountering the same errors during multiple computations.

The inverse of input re-expression is output fusion. When information is provided by diverse sources, perhaps with different resolutions or formats, the task of reconciling the differences in order to derive a more reliable assessment of the prevailing conditions is quite nontrivial. [Elaborate.]

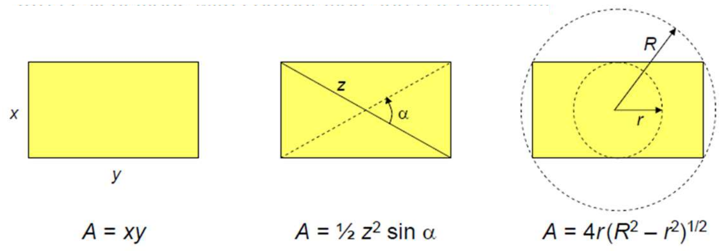


Fig. 23.ddiv Diverse representations and associated area calculations for a rectangle.

23.4 Self-Checking Algorithms

It is sometimes possible to design algorithms and associated data structures to that the computation becomes resilient to both representational and computational errors. A prime example is provided by a method known as algorithm-based malfunction tolerance, which is more widely known in the literarure by the acronym ABFT (algorithm-based fault tolerance).

Consider the 3 × 3 matrix M shown in Fig. 23.abmt1. Adding modulo-8 row checksums and column checksums results in the row-checksum matrix M_r and column checksum matrix M_c, respectively. Including both sets of checksums, with the lower-right matrix element set to the checksum of the row checksums or of the column checksums (it can be shown that the result is the same either way) lead to the full checksum matrix M_f, a representation of M that allows the correction of any single error in the matrix elements and detection of up to 3 errors; some patterns of 4 errors, such as in the 4 elements enclosed in the dashed box in Fig. 23.abmt1, may go undetected.

In addition to correction or detection of representational errors, as outlined in the preceding paragraph, the matrix representation depicted in Fig. 23.abmt1 allows matrix multiplication to be performed on encoded matrices, thus helping with the detection of errors resulting from incorrect arithmetic operations (Fig. abmt2).

Matrix M

$$M = \begin{pmatrix} 2 & 1 & 6 \\ 5 & 3 & 4 \\ 3 & 2 & 7 \end{pmatrix}$$

Row checksum matrix

$$M_r = \begin{pmatrix} 2 & 1 & 6 & 1 \\ 5 & 3 & 4 & 4 \\ 3 & 2 & 7 & 4 \end{pmatrix}$$

Column checksum matrix

$$M_c = \begin{pmatrix} 2 & 1 & 6 \\ 5 & 3 & 4 \\ 3 & 2 & 7 \\ 2 & 6 & 1 \end{pmatrix}$$

Full checksum matrix

$$M_f = \begin{pmatrix} 2 & \boxed{\begin{matrix} 1 & 6 \\ 3 & 4 \end{matrix}} & \begin{matrix} 1 \\ 4 \end{matrix} \\ \begin{matrix} 5 \\ 3 \\ 2 \end{matrix} & \begin{matrix} 2 & 7 \\ 6 & 1 \end{matrix} & \begin{matrix} 4 \\ 1 \end{matrix} \end{pmatrix}$$

Fig. 23.abmt A 3 × 3 matrix and its modulo-8 row checksum, column checksum, and full checksum matrices.

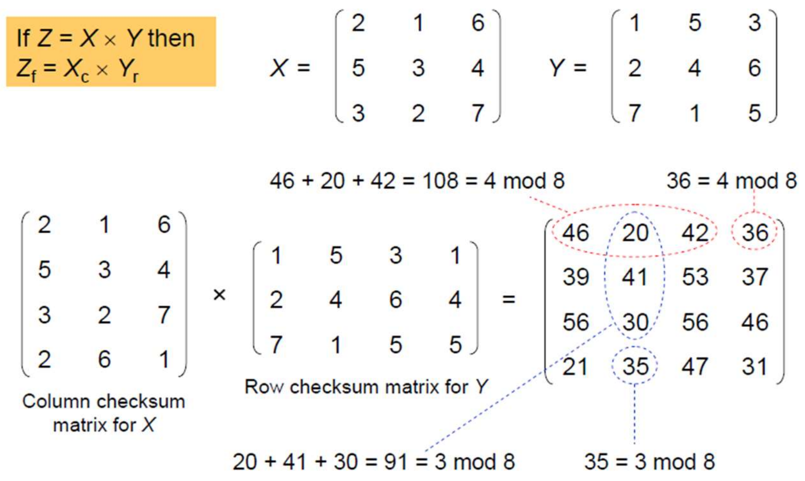

If $Z = X \times Y$ then
$Z_f = X_c \times Y_r$
$X = \begin{pmatrix} 2 & 1 & 6 \\ 5 & 3 & 4 \\ 3 & 2 & 7 \end{pmatrix}$
$Y = \begin{pmatrix} 1 & 5 & 3 \\ 2 & 4 & 6 \\ 7 & 1 & 5 \end{pmatrix}$
46 + 20 + 42 = 108 = 4 mod 8
36 = 4 mod 8
$\begin{pmatrix} 2 & 1 & 6 \\ 5 & 3 & 4 \\ 3 & 2 & 7 \\ 2 & 6 & 1 \end{pmatrix} \times \begin{pmatrix} 1 & 5 & 3 & 1 \\ 2 & 4 & 6 & 4 \\ 7 & 1 & 5 & 5 \end{pmatrix} = \begin{pmatrix} 46 & 20 & 42 & 36 \\ 39 & 41 & 53 & 37 \\ 56 & 30 & 56 & 46 \\ 21 & 35 & 47 & 31 \end{pmatrix}$
Column checksum matrix for X
Row checksum matrix for Y
20 + 41 + 30 = 91 = 3 mod 8
35 = 3 mod 8

Fig. 23.abmt2 Multiplication of matrices with row and column checksums.

23.5 Self-Adapting Algorithms

This section to be written.

23.6 Other Algorithmic Methods

This section to be written.

Problems

23.1 Algorithm-based malfunction tolerance

Consider extending the algorithm-based malfunction tolerance scheme for matrix computations by adding a second check row and a second check column. The added row and column for a matrix M will contain modulo-m' residues of the column and row checksums for the full-checksum matrix M_f, where m' is relatively prime with respect to the original modulus m used in forming M_f.

a. Independent of whether the scheme is practical for matrix computations, derive its error detection and correction capabilities.
b. Discuss whether the matrix multiplication algorithm can be made to work for the new encoding.

23.2 Time redundancy at the application level

We are given a probabilistic algorithm for solving a problem that does not lend itself to deterministic solution. The given algorithm has been tested on a large number of problem instances and is known to produce a correct solution in 82% of the cases. The algorithm makes random choices during its execution, so different runs of the algorithm can be considered statistically independent as far as correctness of the result is concerned. Discuss whether and how the algorithm can be used to produce a solution that is correct at the 99.99% confidence level.

23.3 Quantifying the reliability of programs

Carbin et al. [Carb16] have suggested that reliability of programs under soft errors in the underlying hardware can be quantified by estimating the probability of correctness for variable values and verifying that they exceed predefined thresholds. Discuss the method proposed in the paper with regard to the following.

a. How realistic it is to determine the desired correctness probabilities or lower bounds for them.
b. The fraction of all program failures that can be attributed to soft errors of the kinds considered.
c. Options for corrective actions should the estimated correctness probabilities be unacceptable.
d. Whether the methods proposed might be adaptable to cover other causes of program failures.

23.x Title

Problem intro

a. xxx
b. xxx
c. xxx
d. xxx

References and Further Readings

[Amma88] Ammann, P.E. and J.C. Knight, “Data Diversity: An Approach to Software Fault Tolerance,” *IEEE Trans. Computers*, Vol. 37, No. 4, pp. 418-425, April 1988.

[Aviz97] Avizienis, A., “Toward Systematic Design of Fault-Tolerant Systems,” *IEEE Computer*, Vol. 30, No. 4, pp. 51-58, April 1997.

[Carb16] Carbin, M., S. Misailovic, and M. C. Rinard, “Verifying Quantitative Reliability for Programs that Execute on Unreliable Hardware,” *Communications of the ACM*, Vol. 59, No. 8, pp. 83-91, August 2016.

[Fino09] Finocchi, I., F. Grandoni, and G. F. Italiano, “Optimal Resilient Sorting and Searching in the Presence of Memory Faults,” *Theoretical Computer Science*, Vol. 410, No. 44, pp. 4457-4470, October 2009.

[Golo06] Goloubeva, O., M. Rebaudengo, M. S. Reorda, and M. Violante, *Software-Implemented Hardware Fault Tolerance*, Springer, 2006.

[Huan84] Huang, K. H. and J. A. Abraham, “Algorithm-Based Fault Tolerance for Matrix Operations,” *IEEE Trans. Computers*, Vol. 33, No. 6, pp. 518-528, June 1984.

[John89] Johnson, B. W., *Design and Analysis of Fault-Tolerant Digital Systems*, Addison-Wesley, 1989.

[Kant90] Kant, K. and A. Ravichandran, “Synthesizing Robust Data Structures—An Introduction,” *IEEE Trans. Computers*, Vol. 39, No. 2, pp. 161-173, February 1990.

[Siew92] Siewiorek, D. P., and R. S. Swarz, *Reliable Computer Systems: Design and Evaluation*, Digital Press, 2nd ed., 1992. Also: A. K. Peters, 1998.

[Tayl80] Taylor, D. J., J. P. Black, and D. E. Morgan, “Redundancy in Data Structures: Improving Software Fault Tolerance,” *IEEE Trans. Software Engineering*, Vol. 6, No. 6, pp. 585-594, November 1980.

[Vija97] Vijay, M. and R. Mittal, “Algorithm-Based Fault Tolerance: A Review,” *Microprocessors and Microsystems*, Vol. 21, pp. 151-161, 1997.

24 Software Redundancy

"Those parts of the system that you can hit with a hammer (not advised) are called hardware; those program instructions that you can only curse at are called software."

Anonymous

". . . even perfect program verification can only establish that a program meets its specification. The hardest part of the software task is arriving at a complete and consistent specification, and much of the essence of building a program is in fact the debugging of the specification."

Fredrick P. Brooks, Jr., Essence and Accidents of Software Engineering

Topics in This Chapter

Software and hardware malfunctions are on the surface quite different. It is sometimes argued that software does not age, that programs are not subject to external interference or transient faults, and that all software-related problems are due to design flaws. Thus, software replication does not help, the argument continues, because all copies of the software will have the same flaws. If you tend to agree with the arguments above, you will be quite surprised to learn in this chapter about the use of more or less similar techniques for dealing with both hardware and software malfunctions. In fact combining both classes of methods is our best bet in building ultradependable systems.

24.1 Software Dependability

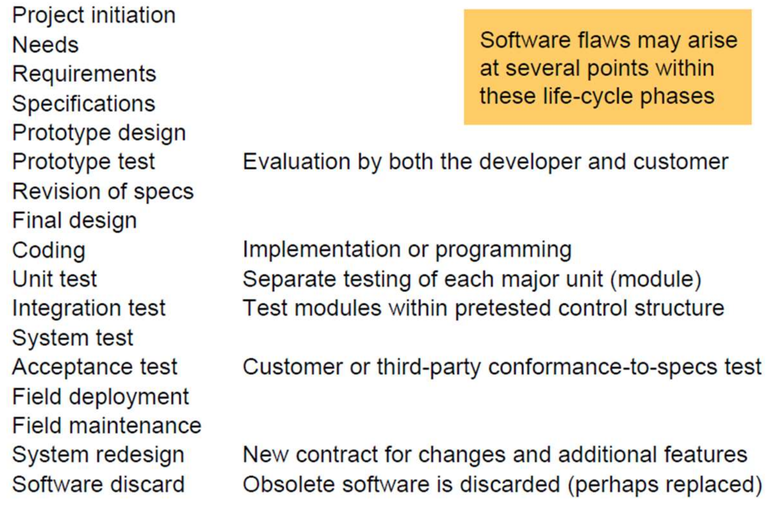
Imagine the following product disclaimers:

> *For a steam iron:* There is no guarantee, explicit or implied, that this device will remove wrinkles from clothing or that it will not lead to the user's electrocution. The manufacturer is not liable for any bodily harm or property damage resulting from the operation of this device.
>
> *For an electric toaster:* The name "toaster" for this product is just a symbolic identifier. There is no guarantee, explicit or implied, that the device will prepare toast. Bread slices inserted in the product may be burnt from time to time, triggering smoke detectors or causing fires. By opening the package, the user acknowledges that s/he is willing to assume sole responsibility for any damages resulting from the product's operation.

You may hesitate before buying such a steam iron or toaster, yet this is how we purchase commodity software. Software producers and marketers, far from postulating dependable operation, do not even promise correct functional behavior! The situation is only slightly better for custom software, produced to exacting functional and reliability specifications.

Software unreliability is caused predominantly by design slips, not by operational deviations. Latent design slips, which form the main mechanisms for software malfunctions, are becoming common in hardware as well, given the phenomenal levels of complexity in modern systems.

The curse of complexity is best illustrated through an example. The 7-Eleven convenience store chain reportedly spent some $9M to make its point-of-sale software Y2K-compliant for its 5200 stores, shortly before the year-2000 problem (caused by the use of 2 digits for the year field in some databases, leading to the problem of years 1900 and 2000 becoming indistinguishable) was to hit the world's computerized information systems. The modified software was subjected to 10,000 tests, all of which were successful. The company's management and information system professionals were relieved, as the system worked flawlessly throughout the year 2000. On January 1, 2001, however, the system began rejecting credit cards, because it somehow "believed" the year to be 1901. The problem was identified and corrected within a day, but it left the lasting message that removing one bug can sometimes just transform or relocate the problem, rather than fix it.



Software flaws may arise at several points within these life-cycle phases

Project initiation	
Needs	
Requirements	
Specifications	
Prototype design	
Prototype test	Evaluation by both the developer and customer
Revision of specs	
Final design	
Coding	Implementation or programming
Unit test	Separate testing of each major unit (module)
Integration test	Test modules within pretested control structure
System test	
Acceptance test	Customer or third-party conformance-to-specs test
Field deployment	
Field maintenance	
System redesign	New contract for changes and additional features
Software discard	Obsolete software is discarded (perhaps replaced)

Fig. 24.sdlc Phases in software development life cycle where flaws might creep in.

To see where flaws might be introduced in software, it is instructive to examine the various phases in the software development life cycle (Fig. 24.sdlc). The specifications stage and later are only relevant if commodity software cannot satisfy the requirements

Beginning with unit test (see Fig. 24.sdlc), major structural and logical problems remaining in a piece of software are removed early on. What remains after extensive verification and validation is a collection of tiny flaws which surface under rare conditions or particular combinations of circumstances, thus giving software failure a statistical nature. Software usually contains one or more flaws per thousand lines of code, with < 1 flaw considered good (linux has been estimated to have 0.1). If there are f flaws in a software component, the hazard rate, that is, rate of failure occurrence per hour, is kf, with k being the constant of proportionality which is determined experimentally (e.g., k = 0.0001). Software reliability is then modeled by:

$$R(t) = e^{-kft} \qquad (24.1.\text{swrel})$$

According to this model, the only way to improve software reliability is to reduce the number of residual flaws through more rigorous verification and/or testing.

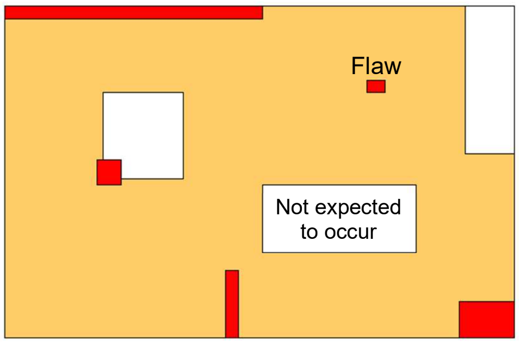


Fig. 24.swflaw Residual software flaws within the input space.

Given extensive testing, the residual flaws in software are by nature difficult to detect. They manifest themselves primarily for unusual combinations of inputs and program states (the so-called "corner cases"), schematically represented in Fig. 24.swflaw. Light shading is used to denote the parts of input/state space for which the software is free from flaws. Unshaded regions represent input/state combinations that are known to be impossible, thus making them irrelevant to proper functioning of the software. Dark shading is used for trouble spots, which have been missed during testing. Occasionally, during the use of a released piece of software, a user's operating conditions will hit one of these trouble spots, thus exposing the associated flaw. Once a flaw has been exposed, it is dealt with through a new release of the software (particularly if the flaw is deemed important in the sense of its potentials for affecting many other users) or through a software patch.

For a while, there was some resistance to the idea of treating software malfunctions in a probabilistic fashion, much like what we do with hardware-caused malfunctions. The argument went that software flaws, which are due to design errors, either exist or do not exist and that they do not emerge probabilistically. However, as discussed in the preceding paragraph, software flaws are often exposed by rare combinations of inputs and internal states, it does make sense to assume that there is a certain probability distribution, derivable from input distributions, for a software malfunction to occur.

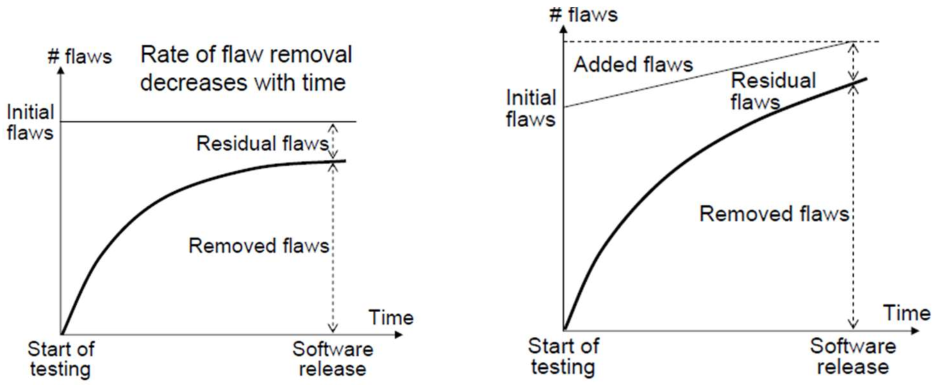
The idea of using software redundancy to counteract uncertainties in design quality and residual bugs is a natural one. However, it is not clear what form the redundancy should take: it is certainly not helpful to replicate the same piece of software, with identical internal flaws/bugs. We will tackle this topic in the last three sections of this chapter.

24.2 Software Malfunction Models

A software flaw or bug can lead to operational error for certain combinations of inputs and system states, causing a software-induced failure. Informally, the term “software failure” is used to denote any software-related dependability problem. Flaw removal can be modeled in various ways, two of which are depicted in Fig. 24.flawr. When removing existing flaws does not introduce any new flaws, we have the optimistic model of Fig. 24.flawr-a. Flaw removal is quick in early stages, but as more flaws are removed, it becomes more difficult to pinpoint the remaining ones, leading to a reduction in flaw removal rate. The more realistic model of Fig. 24.flawr-b assumes that the number of new flaws introduced is proportional to the removal rate. The following example is based on the simpler model of Fig. 24.flawr-a.

Example 24.flaw: Modeling the software flaw removal process Assume that no new flaws are introduced as we remove existing flaws in a piece of software estimated to have F_0 = 132 flaws initially and that the flaw removal rate linearly decreases with time. Model the reliability of this software as a function of time.

Solution: Let F be the number of residual flaws and τ be the testing time in months. From the problem statement, we can write $dF(\tau)/d\tau = -(a - b\tau)$, leading to $F(\tau) = F_0 - a\tau(1 - b\tau/(2a))$. The hazard function is then $z(\tau) = kF(\tau)$, where k is a constant of proportionality, and $R(t) = e^{-kF(\tau)t}$. Taking k to be 0.000132, we find $R(t) = \exp(-0.000132(130 - 30\tau(1 - \tau/16))t)$. If testing is done for τ = 8 months, the reliability equation becomes $e^{-0.00132t}$, which corresponds to an MTTF of 758 hours. Note that no testing would have resulted in an MTTF of 58 hours and that testing for 2, 4, and 6 months would have led to MTTFs of 98, 189, and 433 hours, respectively.



(a) Removing flaws, without generating new ones (b) New flaws introduced are proportional to removal rate

Fig. 24.flawr Modeling flaw removal from software.

Linearly decreasing flaw removal rate isn't the only option in modeling. Constant flaw removal rate has also been considered, but it does not lead to a very realistic model. Exponentially decreasing flaw removal rate is more realistic than linearly decreasing, since flaw removal rate never really becomes 0. Model constants can be estimated via:

- Using a handbook: public ones, or compiled from in-house data
- Matching moments (mean, 2nd moment, . . .) to flaw removal data
- Least-squares estimation, particularly with multiple data sets
- Maximum-likelihood estimation (a statistical method)

In addition to the exponential reliability model based on estimates of the remaining number of flaws in a piece of software, leading to a constant hazard rate or a hazard rate function for a specific amount of testing, a phenomenon similar to wearout in the case of hardware has been observed for software. Of course, software does not wear out or age in the same sense as hardware. Yet we do observe some deterioration in the performance of a piece of software that has been running for a long time. This wearout phenomenon along with the large number of flaws before testing make the defect-related bathtub curve of Fig. 5.btc also applicable to software.

The primary reasons for software aging include accumulation of junk in the state part of the system (which is reversible via restoration) and long-term cumulative effects of updates via patching and the like. As the software's structure deviates from its original clean form, unexpected failures begin to occur. Eventually software becomes so mangled that it must be discarded and redeveloped from scratch.

24.3 Software Verification and Validation

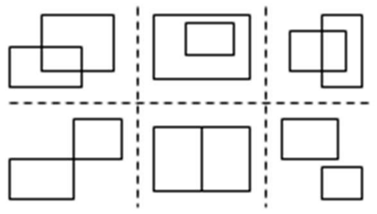
Basic concepts and terms in software verification and validation

Verification: "Are we building the system right?" (meets specifications)
Validation: "Are we building the right system?" (meets requirements)

Both verification and validation use testing as well as formal methods

Software testing	**Formal methods**
Exhaustive testing impossible	Program correctness proof
Test with many typical inputs	Formal specification
Identify and test fringe cases	Model checking

Example: overlap of rectangles



Examples: safety/security-critical

Railway interlocking system [Hlavaty 2001]

Cryptography device [Kirby 1999]

Smart cards [Requet 2000]

Automated lab analysis test equipment [Bicarregui 1997]

Formal proofs for software verification

Program to find the greatest common divisor of integers $m > 0$ and $n > 0$

```
input m and n        ------------ m and n are positive integers
x := m
y := n               ------------ x and y are positive integers, x = m, y = n
while x ≠ y          ------------ Loop invariant: x > 0, y > 0, gcd(x, y) = gcd(m, n)
   if x < y
   then y := y – x
   else x := x – y
   endif
endwhile
output x             ------------ x = gcd(m , n)
```

Steps 1-3: "partial correctness"
Step 4: ensures "total correctness"

The four steps of a correctness proof relating to a program loop:
1. Loop invariant implied by the assertion before the loop (precondition)
2. If satisfied before an iteration begins, then also satisfied at the end
3. Loop invariant and exit condition imply the assertion after the loop
4. Loop executes a finite number of times (termination condition)

Software flaw tolerance

Flaw avoidance strategies include (structured) design methodologies, software reuse, and formal methods

Given that a complex piece of software will contain bugs, can we use redundancy to reduce the probability of software-induced failures?

The ideas of masking redundancy, standby redundancy, and self-checking design have been shown to be applicable to software, leading to various types of fault-tolerant software

"Flaw tolerance" is a better term; "fault tolerance" has been overused

Masking redundancy: N-version programming

Standby redundancy: the recovery-block scheme

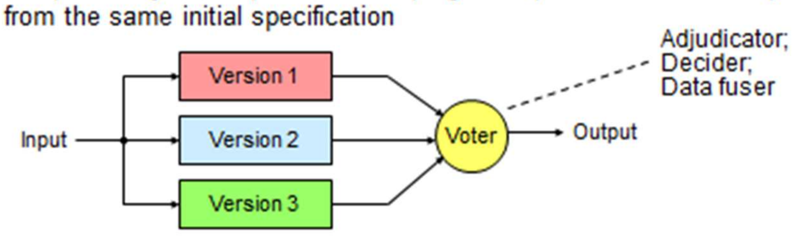
Self-checking design: N-self-checking programming

Sources: *Software Fault Tolerance*, ed. by M.R. Lyu, Wiley, 2005 (on-line book at http://www.cse.cuhk.edu.hk/~lyu/book/sft/index.html)
Also, "Software Fault Tolerance: A Tutorial," 2000 (NASA report, available on-line)

24.4 N-Version Programming

Introduction to N-version programming and justifications for it.

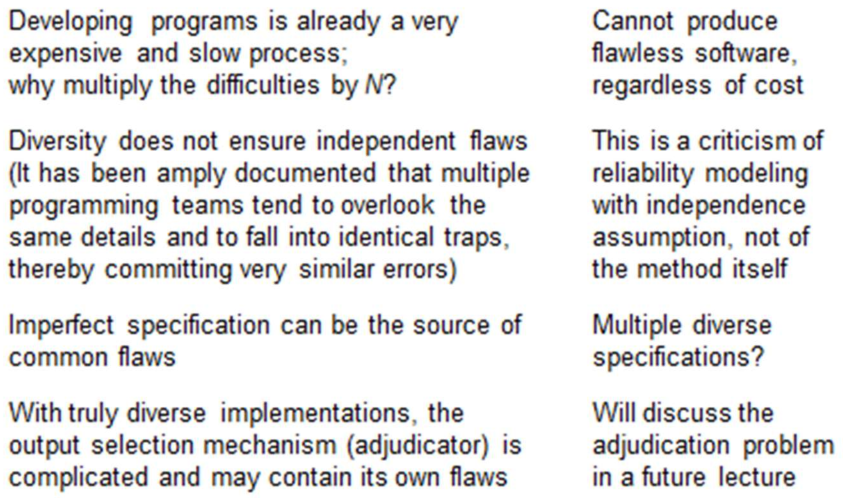
Independently develop *N* different programs (known as "versions") from the same initial specification



The greater the diversity in the *N* versions, the less likely that they will have flaws that produce correlated errors

Diversity in:
Programming teams (personnel and structure)
Software architecture
Algorithms used
Programming languages
Verification tools and methods
Data (input re-expression and output adjustment)

Some objections to N-version programming, and responses to them.

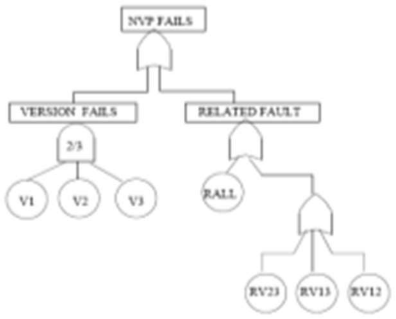


Developing programs is already a very expensive and slow process; why multiply the difficulties by *N*?	Cannot produce flawless software, regardless of cost
Diversity does not ensure independent flaws (It has been amply documented that multiple programming teams tend to overlook the same details and to fall into identical traps, thereby committing very similar errors)	This is a criticism of reliability modeling with independence assumption, not of the method itself
Imperfect specification can be the source of common flaws	Multiple diverse specifications?
With truly diverse implementations, the output selection mechanism (adjudicator) is complicated and may contain its own flaws	Will discuss the adjudication problem in a future lecture

Reliability modeling for N-version programs.

Fault-tree model: the version shown here is fairly simple, but the power of the method comes in handy when combined hardware/software modeling is attempted

Probabilities of coincident flaws are estimated from experimental failure data



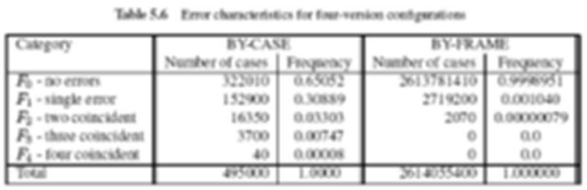


Table 5.6 Error characteristics for four-version configurations

Category	BY-CASE		BY-FRAME	
	Number of cases	Frequency	Number of cases	Frequency
F_0 - no errors	322010	0.65052	2613781410	0.9998951
F_1 - single error	152900	0.30889	2719200	0.001040
F_2 - two coincident	16350	0.03303	2070	0.00000079
F_3 - three coincident	3700	0.00747	0	0.0
F_4 - four coincident	40	0.00008	0	0.0
Total	495000	1.0000	2614055400	1.000000

Source: Dugan & Lyu, 1994 and 1995

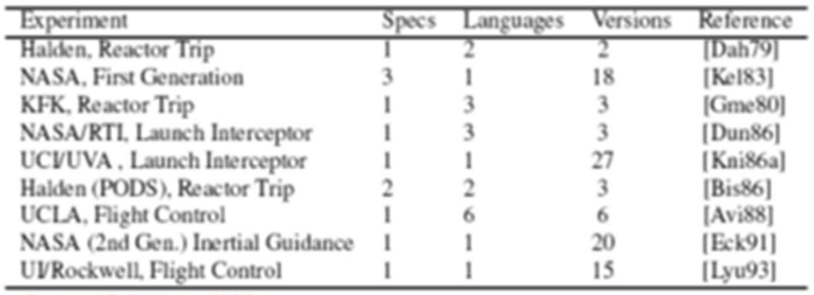
Some applications of N-version programming.

Back-to-back testing: multiple versions can help in the testing process

B777 flight computer: 3 diverse processors running diverse software

Airbus A320/330/340 flight control: 4 dissimilar hardware/software modules drive two independent sets of actuators

Some experiments in N-version programming



Experiment	Specs	Languages	Versions	Reference
Halden, Reactor Trip	1	2	2	[Dah79]
NASA, First Generation	3	1	18	[Kel83]
KFK, Reactor Trip	1	3	3	[Gme80]
NASA/RTI, Launch Interceptor	1	3	3	[Dun86]
UCI/UVA , Launch Interceptor	1	1	27	[Kni86a]
Halden (PODS), Reactor Trip	2	2	3	[Bis86]
UCLA, Flight Control	1	6	6	[Avi88]
NASA (2nd Gen.) Inertial Guidance	1	1	20	[Eck91]
UI/Rockwell, Flight Control	1	1	15	[Lyu93]

Source: P. Bishop, 1995

24.5 The Recovery Block Method

The recovery block method may be viewed as the software counterpart to standby sparing for hardware. Suppose we can verify the results obtained by a software module by subjecting them to an acceptance test. For now, let us assume that the acceptance test is perfect in the sense of not missing any erroneous results and not producing any false positive. Implications of imperfect acceptance tests will be discussed later. With these assumptions, one can organize a number of different software modules all performing the same computation in the form of a recovery block.

```
Recovery block:                                              (24.5.rb)
ensure    acceptance test       ; e.g., sorted list
by        primary module        ; e.g., quicksort
else by   first alternate       ; e.g., bubblesort
  .
  .
  .
else by   last alternate        ; e.g., insertion sort
else fail
```

The program structure 24.5.rb encapsulates a primary software module, which is executed to completion and its results subjected to the acceptance test. Passing of the acceptance test, which occurs in a vast majority of cases, terminates the recovery block. Failing the test triggers the execution of the first alternate module, with the process repeated as above: for each alternate, passing of the acceptance test terminates the block and failing it triggers the execution of the next alternate. A failure event is indicated once all alternates have been tried without success.

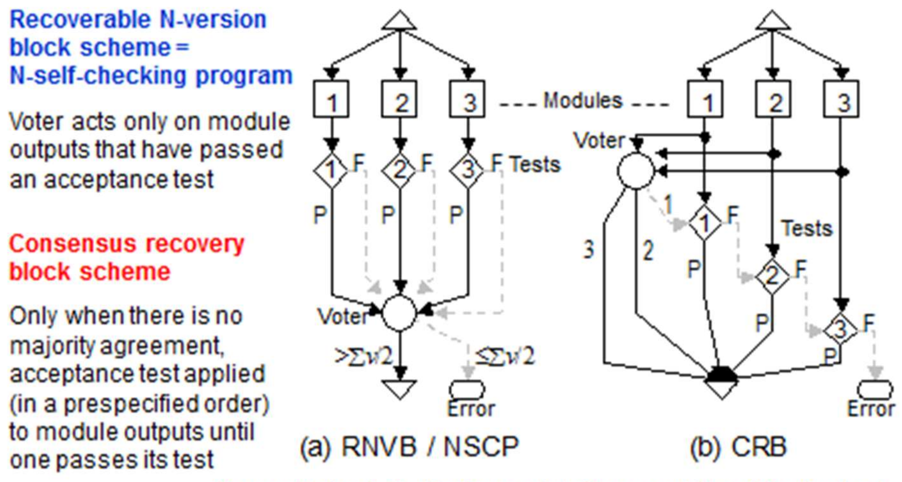
The comments next to the pseudocode lines in the program structure 24.5.rb provide an example in which the task to be performed is that of sorting a list. The primary module uses the quicksort algorithm, which has very good average-case running time but is rather complicated in terms of programming, and thus prone to residual software bugs. The first alternate uses the bubblesort algorithm, which is not as fast, but much easier to write and debug. The longer running time of bubblesort may not be problematic, given that we expect the alternates to be executed rarely. As we go down the list of the alternates, even simpler but perhaps lower-performing algorithms may be utilized. In this way, diversity can be provided among the alternates can be provided, while also reducing the development cost relative to N-version programming. Design diversity makes it more likely for one of the alternate modules to succeed when the primary module fails to produce an acceptable result.

The acceptance test for our sorting example can take the form of a linear scan of the output list to verify that its elements are in nondescending or nonascending order, depending on the direction of sorting. Such an acceptance test will detect an improperly sorted list, but may not catch the problem when the output list does not consist of exactly the same set of values as the input list. We can of course make the acceptance test as comprehensive as desired, but a price is paid in both software development effort and running time, given that the acceptance test is on the critical path.

In general, the acceptance test can range from a simple reasonableness check to a sophisticated and thorough validation effort. Note that performing the computation a second time and comparing the two sets of results can be viewed as a form of acceptance testing, in which the acceptance test module is of the same order of complexity as the main computational module. Computations that have simple inverses lend themselves to efficient acceptance testing. For example, the results of root finding for a polynomial can be readily verified by polynomial evaluation.

24.6 Hybrid Software Redundancy

The various software redundancy methods, including N-version programming and recovery blocks can be unified in a way that allows the discovery of novel combinations of replication and acceptance testing and to facilitate the comparison of existing methods and new proposed methods. We begin by representing two hybrid redundancy schemes in the form of block diagrams composed of software modules and acceptance tests.



Recoverable N-version block scheme = N-self-checking program

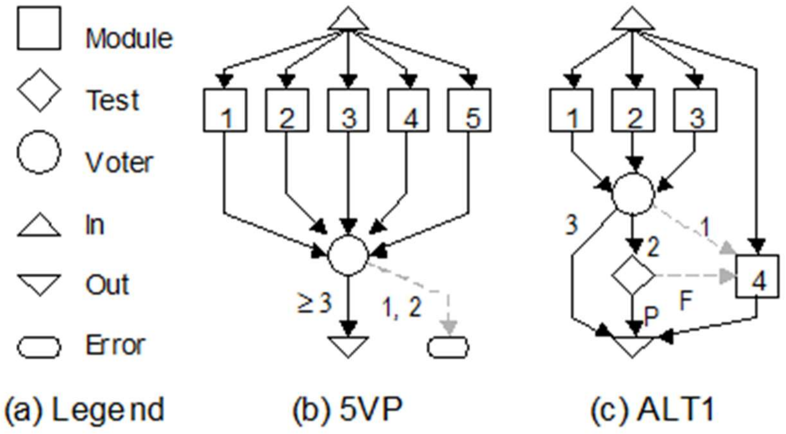
Voter acts only on module outputs that have passed an acceptance test

Consensus recovery block scheme

Only when there is no majority agreement, acceptance test applied (in a prespecified order) to module outputs until one passes its test

Source: Parhami, B., "An Approach to Component-Based Synthesis of Fault-Tolerant Software," *Informatica*, Vol. 25, pp. 533-543, Nov. 2001.

We now present the elements of a general notation that facilitates the study and synthesis of other software redundancy schemes.



Source: Parhami, B., "An Approach to Component-Based Synthesis of Fault-Tolerant Software," *Informatica*, Vol. 25, pp. 533-543, Nov. 2001.

Problems

24.1 The year-2038 design flaw

Study the so-called "year 2038" (abbreviated Y2038 or Y2K38) problem and write a two-page report about it. In your report, discuss implications of the Y2038 problem to computer system reliability as well as similarities and differences with the infamous Y2K problem.

24.2 Multiversion programming

The $t/(n-1)$ version programming [Xu97] combines the ideas of N-version programming, discussed in this chapter, and the malfunction diagnosis techniques of Chapter 17. Write a two-page report describing the method and its advantages relative to other software redundancy schemes.

24.3 Software aging and rejuvenation

A concept that has emerged in recent years to counteract the effects of software aging is that of software rejuvenation. Research the notions of software aging and software rejuvenation and present your findings in a two-page report (typed, single-space). In your report, provide precise definitions for the notions introduced, and paint an accurate picture of the application domains and impact of each method discussed. A good starting point for finding relevant references is [Silv09].

24.4 Acceptance testing [Kore02]

The correct output, z, of some program has as its probability density function the truncated exponential function given below, where L is a known positive constant: $f(z)$ = if $0 \le z \le L$ then $\mu e^{-\mu z}/(1 - e^{-\mu L})$ else 0. On any particular input, the program fails with probability q, in which case it produces an arbitrary value with uniform distribution in $[0, L]$. The penalty of producing an incorrect value is E, while that of producing no value at all is S, where E and S are known constants. An acceptance test is to be set up in the form of a range check which rejects any output that does not fall in $[0, R]$. Find the optimal value of R for which the expected total penalty is minimized.

24.5 N-channel computation in software and hardware

Discuss the similarities and differences between N-version programming and the NMR (replication and voting) method as used for hardware. Include in your discussion both implementation aspects and reliability modeling considerations.

24.6 Software debugging [Kore02]

Let the probability of uncovering a bug after applying t seconds of testing to a software module, given that it has at least one bug, be $1 - e^{-\mu t}$. You believe at the outset, perhaps based on past experience with similar software, that the probability of having at least one bug in your software is $q = 1 - p$. Assume that after t seconds of testing, you fail to find a bug.

a. Prove: prob{the software is bug-free | t seconds of testing revealed no bugs} = $1/[1 + (q/p)e^{-\mu t}]$

b. Assuming $q = 0.9$, plot the variation of the confidence factor of part a for $\mu = 0.001$, 0.01, and 0.1, as t varies between 0 and 10 000.

c. Assuming $\mu = 0.01$, plot the variation of the confidence factor of part a for $q = 0.9$, 0.7, and 0.5, as t varies between 0 and 10 000.

d. Discuss the results of parts b and c and draw appropriate conclusions from them.

24.7 Program correctness proof

Consider the program fragment: $s = 0$; $k = 0$; while $k \leq n$ do $s = s + 2k$; $k = k + 1$; endwhile

a. Prove that the program fragment computes $f(n) = n(n + 1)$ when $n > 0$ is an integer.

b. What does the fragment compute when n is a positive real number?

c. What will the fragment compute if we reorder the two statements inside the while loop?

24.8 Voting for 3-version software

The 3 versions of a program produce the following sets of output values at consecutive voting points, with the voter output also shown. Determine the voting algorithm used in decision schemes a and b. Discuss and fully justify your answers.

Data set	Values produced			Decision a	Decision b
1	48.3	48.2	48.1	48.2	48.2
2	48.2	48.7	48.0	48.2	48.3
3	48.1	49.4	47.7	48.1	48.4
4	48.3	51.3	47.9	48.3	48.1
5	48.0	52.6	48.2	48.2	48.1
6	48.3	53.7	48.1	48.3	48.2
7	48.1	54.5	47.9	48.1	48.0

24.9 Software redundancy in a Mars mission

In a one-page, single-spaced report, describe the role of software redundancy in how NASA's Curiosity Rover reached Mars and functioned as intended by its designers [Holz14].

24.10 The leap-second problem

Read the article [Sava15] about the impact of leap seconds on the reliable operation of computer systems. In one single-space typed page, describe the problem, along with possible solutions or workarounds.

24.11 Verification of program reliability

Carbin, Misailovic, and Rinard [Carb16] propose an interesting method for quantifying program reliability. Study the cited paper and outline in one typewritten page the essense of their method and its practical implications. Pay special attention to the malfunction model assumed.

24.12 Dependability considerations for controllers

Read the paper [Alka19] and address the following questions in a one-page report.

a. What is different about controllers compared with other components, such as memory modules, CPUs, GPUs, and the like?

b. What are the two most important techniques described in the paper for improving controller reliability?

c. What are the redundancy and other cost factors associated with the methods of part b?

d. Why do the authors focus on FPGA-based implementation of controllers?

References and Further Readings

[Alka19] Alkady, G. I., R. M. Daoud, H. H. Amer, M. Y. ElSalamouny, and I. Adly, "Failures in Fault-Tolerant FPGA-Based Controllers—A Case Study," *Microprocessors and Microsystems*, Vol. 64, pp. 178-184, 2019.

[Arms10] Armstrong, J., "Erlang," *Communications of the ACM*, Vol. 53, No. 9, pp. 68-75, September 2010. {Erlang is an open-source language that allows for easy programming of multicore systems and fault-tolerant distributed applications.}

[Bica97] Bicarregui, J., J. Dick, B. Matthews, and E. Woods, "Making the Most of Formal Specification through Animation, Testing, and Proof," *Science of Computer Programming*, Vol. 29, Nos. 1-2, pp. 53-78, July 1997.

[Carb16] Carbin, M., S. Misailovic, and M. C. Rinard, "Verifying Quantitative Reliability for Programs that Execute on Unreliable Hardware," *Communications of the ACM*, Vol. 59, No. 8, pp. 83-91, August 2016.

[Hlav01] Hlavaty, T., L. Preucil, and P. Stepan, "Case Study: Formal Specification and Verification of Railway Interlocking System," *Proc. 27th Euromicro Conf.*, pp. 258-263, 2001.

[Hlav04] Hlavacek, I., J. Chleboun, and I. Babuska, *Uncertain Input Data Problems and the Worst Scenario Method*, Elsevier, 2004.

[Holz14] Holzmann, G. J., "Mars Code," *Commuications of the ACM*, Vol. 57, No. 2, pp, 64-73, February 2014.

[Kirb99] Kirby, J., M. Archer, and C. Heitmeyer, "Applying Formal Methods to an Information Security Device: An Experience Report," *Proc. 4th IEEE Int'l Symp. High-Assurance Systems Engineering*, 1999.

[Kore02] Koren, I. and C. M. Krishna, *Fault-Tolerant Systems*, Morgan Kaufmann, 2007.

[Lyu05] Lyu, M. R. (ed.), *Software Fault Tolerance*, Wiley, 2005. Available online at: http://www.cse.cuhk.edu.hk/~lyu/book/sft/index.html

[NASA00] US National Aeronautics and Space Administration, "Software Fault Tolerance: A Tutorial," NASA report, 2000.

[Parh01] Parhami, B., "An Approach to Component-Based Synthesis of Fault-Tolerant Software," *Informatica*, Vol. 25, pp. 533-543, November 2001.

[Requ00] Requet, A. and G. Bossu, "Embedding Formally Proved Code in a Smart Card: Converting B to C," *Proc. Int'l Conf. Formal Engineering Methods*, pp. 15-22, 2000.

[Roth89] Rothermel, K. and C. Mohan, "ARIES/NT: A Recovery Method Based on Write-Ahead Logging for Nested Transactions," IBM Thomas J. Watson Research Division, 1989.

[Sala14] Salako, K. and L. Strigini, "When Does 'Diversity' in Development Reduce Common Failures? Insights from Probabilistic Modeling," *IEEE Trans. Dependable and Secure Systems*, Vol. 11, No. 2, pp. 193-206, March/April 2014.

[Sava15] Savage, N., "Split Second," *Communications of the ACM*, Vol. 58, No. 9, pp. 12-14, September 2015.

[Silv09] Silva, L. M., J. Alonso, and J. Torres, "Using Virtualization to Improve Software Rejuvenation," *IEEE Trans. Computers*, Vol. 58, No. 11, November 2009, pp. 1525-1538.

[Xu97] Xu, J. and B. Randell, "Software Fault Tolerance: $t/(n-1)$-Variant Programming," *IEEE Trans. Reliability*, Vol. 46, No. 1, pp. 60-68, March 1997.