Software Rejuvenation

ROBERT HANMER, Alcatel-Lucent

Software rejuvenation is a technique of proactive fault tolerance that designs the system for periodic reboots. This paper contains three patterns of which Software Rejuvenation is the first. The second, Count the Black Sheep, provides a solution to the problem of knowing precisely what fault has activated. This information is needed to recover quickly and to be able to provide effective long-term treatment. N-Version Programming completes this collection. Multiple versions of an implementation are created to prevent incorrect understanding of the requirements causing operational failures. All the versions will be executed simultaneously and the correct output chosen.

Categories and Subject Descriptors: **C.4: [Performance of Systems]** *Fault tolerance, Reliability, availability, and serviceability*, **D.2.11 [Software Engineering]:** Software Architecture –*Patterns*, **K.6.3 [Management of Computing and Information Systems]:** Software Management – *Software Development*

General Terms: reliability, design, management

Additional Key Words and Phrases: fault tolerance, availability, patterns, software engineering

INTRODUCTION

This paper contains patterns on REJUVENATION, using ERROR COUNTING as a form of fault isolation and N-VERSION PROGRAMMING. These three patterns supplement the set of patterns in **Patterns for Fault Tolerant Software** [Hanmer 2007]. The book focused on software fault tolerance techniques that individuals or small teams could implement themselves. REJUVENATION and ERROR COUNTING fit within that scope, N-Version Programming requires a higher level of project and management support to staff multiple teams, and does not fit within the initial scope of the book. After publication of this book in 2007 I have been working on patterns that round out a pattern-style treatment of software fault tolerance with patterns that did not fit within the context of the book.

The terms *fault*, *error* and *failure* have specific meanings.

> A system **failure** occurs when the delivered service no longer complies with the *specification*, the latter being an agreed upon description of the system's expected function and/or service. An **error** is that part of the system state that is liable to lead to subsequent failure; an error affecting the service is an indication that a failure occurs or has occurred. The adjudged or hypothesized cause of an error is a **fault**. [Laprie 1991]

Software faults are classified into two types based upon their characteristics; they are either Bohrbugs or Mandelbugs. "Bohrbugs" are faults that activate consistently in well-defined circumstances. Bohrbugs don't change with time. Programming faults that can be detected during code inspection are Bohrbugs because they are constant and reproducible.

"Mandelbugs" are faults with complex activation and/or error propagation properties. This complexity arises when there is a time lag between the activation and the occurrence of an error

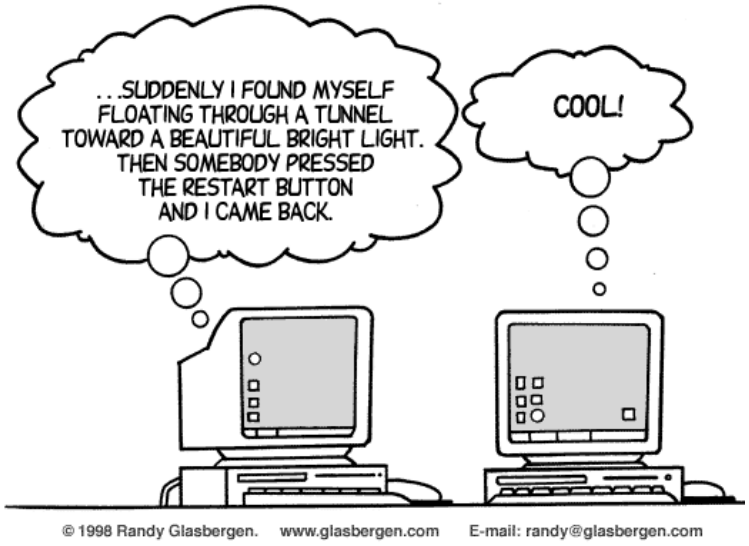
Author's address: R. Hanmer, 2000 Lucent Lane, Naperville, IL 60563; email Robert.hanmer@alcatel-lucent.

or when there are indirect factors that influence the activation. For example, interactions of the system with its environment, the timing of inputs and operations relative to each other or the interaction of operation sequencing. Mandelbugs, unlike Bohrbugs, are difficult to locate because the error and failures might not be near the actual fault activation in code/operation location or time.

Some Mandelbugs can be related to time period during which the software/system have been operating. These result from the accumulation of internal errors or when the activation of the Mandelbug is somehow triggered by the total time that the system has been operating. A Mandelbug is either an "aging-related" fault or a "non-aging-related" fault. [Grottke et al. 2010]

---

1. REJUVENATION[1] **



Used with permission.

… The software is in an application that requires high availability. Software gets crusty/rusty/degraded over time and suffers from aging-related Mandelbugs. These eventually lead to failure of some sort.

The cost of unplanned outages is higher than the cost of planned outages. In unplanned outages system state is lost and needs to be rebuilt, revenue-generating activities terminate abnormally resulting in a loss of revenue and customers are dissatisfied. Planned outages can be graceful and often invisible from the customer's perspective.

❖ ❖ ❖

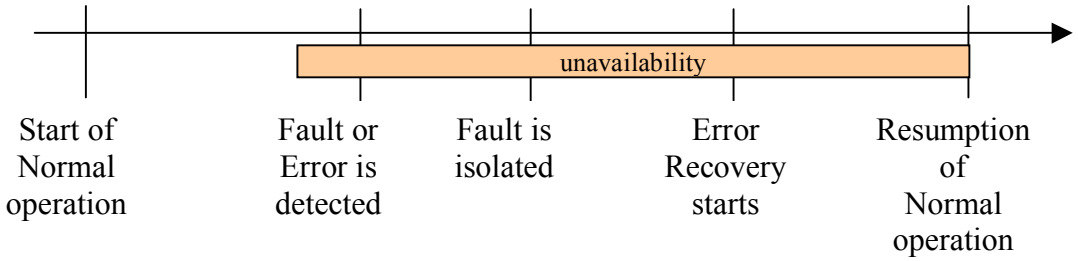
Software degrades until it breaks. Can we avoid the costs of failures from degradation?

The normal stages of software life begin with its initialization and the beginning of its processing. It works flawlessly for a time hopefully a long time). While it is working it is slowly degrading. Little pieces of unused memory are reserved for uses that never occur, bits of information are saved even though they won't be used again, etc. Eventually a software fault will activate. Perhaps it runs out of memory or some other resource, or get confused by the existence of

[1] This pattern is based on Microreboot work first published by Candea et al. [2004]. The general concept of rejuvenation was first described from Bell Labs by Kintalla et. al [1995].

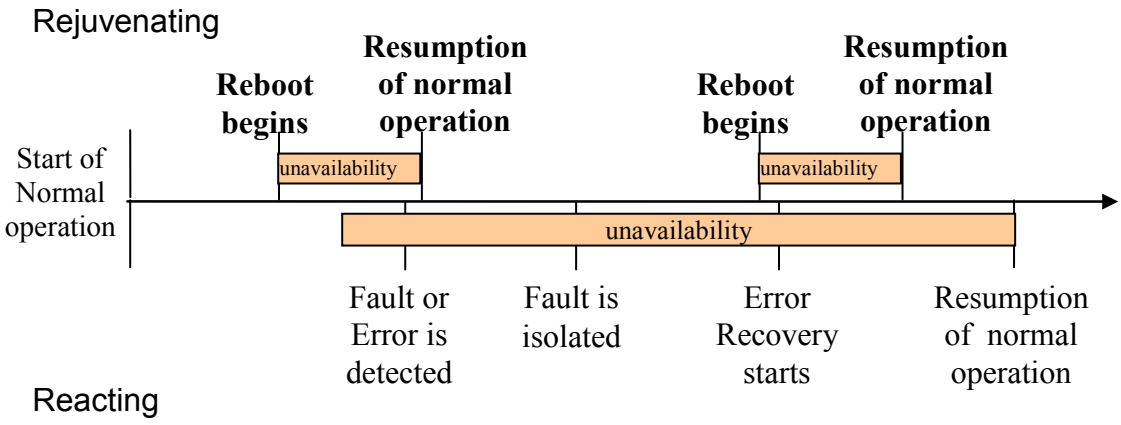
unused data. When this occurs, an error is likely to happen which might be followed by some type of failure.

In software that is designed for high availability, there are mechanisms in place that will detect the fault (or in many cases the error that the fault causes). The fault will be detected and isolated and recovery mechanisms will be tried. Sometimes recovery from the fault is achieved through simple steps; sometimes more aggressive steps will be required, which might require ESCALATION (9)[2]. This all takes time, as shown in Timeline I.



Timeline I: Usual fault detection, isolation and recovery stages.

Rebooting or restarting is one of the many recovery actions that are possible. This sometimes takes the place as the last step in an ESCALATION (9) path. By the time the reboot takes place, a long time has usually passed since the error occurred; refer to the bottom part of Timeline II. Look to the top part of Timeline II to see what happens when the restart is done proactively and not as the result of error recovery. The cost of planned downtime (i.e. time when there isn't a detection and isolation phase) is lower than the cost of unplanned downtime. The graceful shutdown and reboot is the rejuvenation of the software.



Timeline II: (top) Rejuvenating, (bottom) Reacting

The usual way of reporting a system's availability is to report its Mean Time To Failure (MTTF) and its Mean Time to Repair (MTTR). MTTR is based upon the actions the system takes to recover from a fault that activates. MTTF is the time between fault activations. Systems designed for high availability have extremely high MTTF, which makes validating their actual

[2] Numbers within parenthesis refer to the reference number of the pattern within **Patterns for Fault Tolerant Software** [Hanmer 2007].

MTTF difficult. It is usually done by monitoring many systems for a period of time and extrapolating the failure results, or else it is done through statistical methods. MTTR is much easier for the customer to monitor; the customer sees how effective and timely recovery is accomplished. The total time of short planned rejuvenations can be less than the time it takes to recover from a single error. [Fox 2002]

Deciding when to perform a rejuvenating reboot can be done in several ways. Which method is best will depend on the system's context. The decision can be made during execution, for example when resources are getting low indicating that there is a memory leak about to become an error. [Huang et. al 1995] If the rejuvenation is triggered based on execution time detection then SOMEONE IN CHARGE (8) is used to monitor and trigger the rejuvenation.

The open source tool Monit [Monit 2010] has the ability to monitor processes and resources and it can be configured to take proactive action to rejuvenate the system. The example below shows how Monit can be used to check resource usage and restart the Apache httpd daemon if resource usage exceeds predetermined limits. Alerts are sent when the CPU usage of the http daemon and its child processes raises beyond 60% for over two cycles. The httpd daemon is restarted if the CPU usage is over 80% for five cycles or the memory usage over 100MB for five cycles [Yoder 2010]:

Monit example

```
check process apache with pidfile /var/run/httpd.d
start program = "/etc/init.d/httpd start"
stop program  = "/etc/init.d/httpd stop"
if cpu > 40% for 2 cycles then alert
if totalcpu > 60% for 2 cycles then alert
if totalcpu > 80% for 5 cycles then restart
if mem > 100 MB for 5 cycles then stop
```

The planning can alternatively be done in advance either by modeling or by empirical measurements during development testing. A hybrid approach uses SOMEONE IN CHARGE (8) to periodically take measurements that are used to dynamically compute when rejuvenation should occur. [Trivedi et. al 2000, Vaidyanathan and Trivedi 2005] How the decision is made is another issue altogether which is the study of academic research.

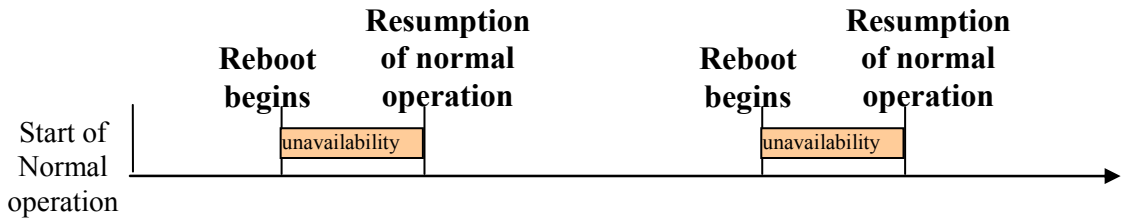
Rejuvenation can occur at any of several scopes: system, application, process, or thread. The entire system can be restarted or the tiniest part of an execution flow can be restarted. The fastest rejuvenation will occur at the smallest scope, but rejuvenation at that scope is not always possible.

Rejuvenation doesn't require REDUNDANCY (1) in the system. If the part of the system to be rejuvenated is redundant, then it is possible to rejuvenate without losing any service. [Vaidyanathan and Trivedi 2005)

The rejuvenation mechanisms should be surrounded by traditional fault tolerance mechanisms such as detection and recovery to ensure that in the unlikely case that the rejuvenation does not work that the system will still be able to recover through traditional means.

Therefore,

Periodically rejuvenate a software item by shutting it down and restarting it.



Timeline III: Rejuvenation

If rejuvenation is not done often enough the system will have unplanned downtime as it goes through the detection, isolation and recovery cycles associated with errors. If rejuvenation is done too often then the system is unavailable for longer than it needs to be for optimal normal operation.

It is easy to confuse rejuvenation with typical fault tolerance. For example, the Service Reanimator tool [2010] watches for errors, such as when a monitored processes terminates prematurely, and then takes action restarting the process. This is fault tolerance, where the detection of an error (terminated process) is done and the error recovery step restarts the process. Proactive rejuvenation anticipates errors by stopping and restarting processes before they have a chance to stop erroneously.

Rejuvenation helps prevent errors from aging-related Mandelbugs. Rejuvenation sometimes helps with non-aging-related Mandelbugs when the rejuvenation takes place between the fault activation and the manifestation as an error. Rejuvenation doesn't help cope with Bohrbugs which occur in very specific circumstances and are not dependant upon system degradation.

❖ ❖ ❖

Rejuvenation events will cause the software (or part of the software) to be unavailable for the duration of the restart. In some situations this will be included within a system's planned downtime allocation. In other situations it will be unobservable and won't need to be accounted for (e. g. when there's REDUNDANCY (1)). The duration of the rejuvenating restart must be computed into the overall availability predictions for the system.

Rejuvenation can be implemented by mechanisms as simple as an entry in a time-related job scheduler such as the Unix `crontab` [1993] scheduler. An example implementation watches for virtual machines to run out of resources and restarts it before it fails. [Yoder 2010] Open source tools such as Monit [Monit 2010] can be added to the system to perform the monitoring.

Businesses quite often do a regular reboot as part of a scheduled maintenance to help prevent aging-related Mandelbugs. A database server might for example be rebooted each night during a period of low activity to reduce the probability of errors during busy times. This is an example of a kind of ROUTINE EXERCISE (23).

ROUTINE EXERCISES (23) are done to ensure that the fault recovery mechanisms and the REDUNDANCY (3) elements in a system do not have latent errors and that they will operate correctly when they are needed. Rejuvenation is closely related, but is done to proactively eliminate aging-related Mandelbugs, not to periodically check recovery mechanisms.

Rejuvenation also does not require inherent redundancy in the system. The mechanisms and techniques are similar, but the intent is different.

Sometimes restarting a system with a mission critical section would be so disastrous that it cannot be done. In these cases the system must be made fully redundant with seamless fault tolerance to prevent unplanned downtime from ever occurring. With this infrastructure rejuvenation can be implemented because the system has been provided with the ability to seamlessly migrate or FAILOVER (36) to redundant elements. …

---

2. COUNT THE BLACK SHEEP[3] *

Also known as “Error Counting”

Photography by Lynn Ede www.cheltenhamdailyphoto-lynn.blogspot.com

… Errors are occurring in the system. Tye system might have been processing the errors or it might have been RIDING OVER TRANSIENTS (26) and ignoring them.

The system is in a highly reliable or highly available environment. Many different components work together to provide the required functionality. They can be either distributed systems where the components are geographically close together or networked systems. The components have assigned responsibilities, and might be decomposable into smaller components. REDUNDANCY (3) exists on some level making it difficult to identify precisely the component in the system that contains the fault.

The system is well-understood. The kinds of errors and their common triggering faults are known and can be categorized. Bohrbug should be corrected, Mandelbugs should be categorized. You are designing the detection and recovery system with requirements to prevent failures.

❖ ❖ ❖

[3] This strategy is discussed in [Meyers et al 1977].

Errors keep occurring. Identification of where the errors occur and originate (i.e. what fault activated) so that the error processing is appropriate. Faults must be identified before they can be treated.

Faults caused by Bohrbugs and Mandelbugs reoccur and cause errors whenever their activation conditions are met. If nothing is done to correct the faults or to avoid the conditions the errors will reoccur. Reoccurring errors increase the likelihood that a failure will be observed. After errors reoccur enough times the pattern emerges and they can be treated correctly.

The faults that caused the error must be identified in order to know how to correctly recover from the error. In complex systems there can be several faults that cause the same error, or errors that look very similar. FAULT CORRELATION (12) is one of the first steps in the detection phase of error handling. During system development or operation error and fault correlations should be identified.

As an example, the signature of a memory exception error should be saved and correlated with a memory fault. There may be several types of memory exception errors, for example in different regions of memory. There might be several different faults that cause the errors such as memory allocation or deallocation faults, invalid operations, etc. Knowing the type of error and its causing fault increases the probability that selected recovery actions will be appropriate and will actually remedy the error.

The system can either process each error immediately when it occurs, using the best available correlation of which particular fault is present, or it can RIDE OVER TRANSIENTS (26) and ignore the error, but keep track of its occurrence for future reference. A third alternative is to combine both options and use ESCALATION (9) techniques to extend and adapt the actions taking into consideration the improved understanding of the error and its real triggering fault.

When an error occurs for the first time it should be processed, and a tally kept. Incrementing the counter that a particular error occurred is sometimes called “pegging a count”. If the error reoccurs within a short time period (which can be determined by a LEAKY BUCKET COUNTER (27)) error processing should again be performed. But if the error keeps occurring then ESCALATION (9) to a more drastic processing technique should be tried. Error specificcounts are needed to drive the ESCALATION.

If the error reoccurs when the LEAKY BUCKET COUNTER (27) is no longer watching for reoccurrences there is still benefit to pegging the count and using it. Errors outside the threshold of the LEAKY BUCKET imply that the error is occurring slowly enough that error processing won’t ESCALATE (9) to a more drastic error processing action. Pegging a count can help the system’s maintenance personnel and developers identify and treat the fault in a way that it can be eliminated in a future release of the system.

The system might be logging errors but the log is probably intended for reporting purposes, rather than for diagnostic purposes. The error processing infrastructure and the logging infrastructure can work together though. When an error occurs and is correlated with a certain fault a record is sent to the logging system. The logging system can provide longer term storage of what errors have occurred.

Therefore,

The system should keep a table of errors and the number of times that each fault that causes the error occurs. Use this table to identify the faults that need to be treated or that should cause ESCALATION (9).

Error ID	Correlated Fault	Count of Error Processing correlated to fault	Errors before Escalation?
A	fa1	1	1
B	fb1	3	5
B	fb2	0	5
C	fc1	0	10
D	fd1	15	20
D	fd2	6	10
D	fd3	0	1
E	fe1	2	2
F	ff2	5	10

Figure 1. Example error and fault correlation table

From the data, the pattern of which fault is causing the most errors begins to appear. The table can also be used a priori to specify whether escalation should occur whenever the error occurs. The number of times that an error occurs and that a particular fault is identified as the most likely cause of the error is an indication of when the fault should be treated and removed from the system.

❖ ❖ ❖

FAULT INSERTION TESTING [not written] and accelerated life testing are both effective techniques to create the list of faults which should be counted and proactively treated. If neither technique is available the list of faults can be accumulated over time through logs of errors and error recovery.

The errors that occur might be totally unexpected. The assumption here has been that the faults and the errors that they caused were known and as a result they could be "covered" by detection and recovery. The coverage factor indicates the percentage of errors that occur that are recovered automatically within a specified time period given that an error has occurred. High coverage factors are desirable, but are more expensive to develop and to test. Fault insertion testing is also helpful to improve and to quantify a system's coverage factor. The record of error and fault occurrences must be able to record that unexpected errors have occurred.

A process of recording errors and faults to aid correlation and to aid the prioritization of treatment must not lead to a false sense of security or to delays in error processing. Error tabulating and error processing must be done in parallel to maximize the system reliability.

The faults that cause errors repeatedly should be the target of fault treatment activities and corrected in the next release of the software. Tests that can recreate the error are excellent candidates for the system's regression test suites. The table of observed errors and faults that are correlated with the errors will give a ranking that can be used to select which faults to treat. This is a benefit even if the pattern of errors and faults remains hidden. …

---

3. N-VERSION PROGRAMMING

… The system needs to be both reliable and available with as few latent faults as possible; Fault Prevention is of paramount importance. For example, a spacecraft carrying astronauts on a round trip journey to space needs to execute flawlessly to return those astronauts safely to the ground.

It also needs to be highly available so that the astronauts have access to the systems continuously. The system will include REDUNDANCY (3) to provide for continuous availability.

A new system is being started. Requirements specifications are being created afresh for this system. Only the most basic parts of previous systems will be reused in the new system, the new system is not the evolution of an existing system.

Resources are unlimited. Reliability is so important that the project has at least an implicit carte blanche to design and build the most reliable system possible with the highest quality possible.

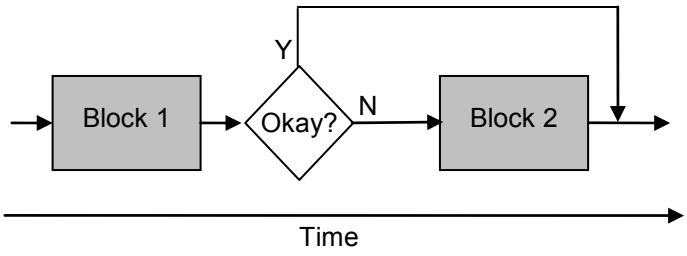
❖ ❖ ❖

Incorrect understandings of system specifications lead to faults. Incorrect understandings can be either corrected or passed to others whenever architects and designers compare notes or direct each others work. How can faults due to incorrect understandings of specifications be eliminated?

A major component of fault tolerance is REDUNDANCY (3), either spatial or temporal redundancy. Spatial redundancy is the redundancy of different components performing the same task in parallel. Temporal redundancy refers to redundancy in time. The same task is done serially, or at non-serial different times.

Redundancy can be designed into the system's hardware or software. Hardware redundancy involves providing multiple system components, such as multiple processors, network interfaces or other, specific peripheral devices. Redundant hardware generally implements spatial redundancy. Additional hardware, in the form of specialized circuits or arbiters, is required to combine the results from the redundant hardware into the single result that will be acted upon.

Redundancy in software is typically done temporally. A common way of implementing this is through a RECOVERY BLOCK (4) structure. In this structure a version of the code is executed and the results checked. The check determines if the result is correct, or within acceptable parameters. If the check passes, the system continues operation. If the result does not pass the check, then another bock of code is executed. The results of this block are then also checked. Two or more blocks can be used.



ensure *Successful Execution*
by *executing primary block*
else by *executing secondary block #1*
else by *executing secondary block #2*
…
else by *executing secondary block #n*
else *trigger exception ().*

Figure 2. Two representations of Recovery Blocks (4)

Another way of achieving software redundancy is to execute multiple versions of the same code serially. The results are compared through VOTING (21) to select the result to be acted upon. This concept of multiple versions was proposed as early as 1837:

> When the formula to be computed is complicated, it may be algebraically arranged for computation in two or more totally distinct ways, and two or more sets of cards [software programs] may be made. If the same constants are now employed with each set, and if under these circumstances the results agree, we may then be quite secure of the accuracy of them all. [Babbage 1837]

The versions that are executed can be either the same version or different versions. The advantage of using the same version code version is to reduce costs. The disadvantage is that the same software when confronted with the same stimuli will produce the same results, which may be erroneous.

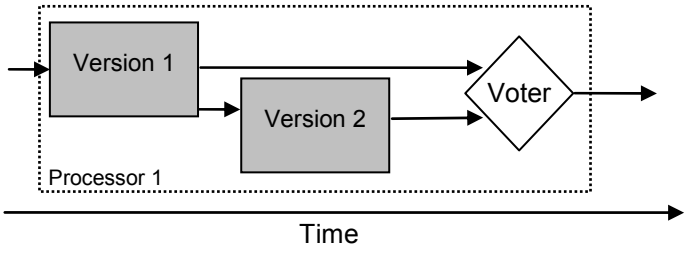


Figure 3. Multiple serial executions with voting

The serial execution of the versions takes time. All the instances execute to completion. The assumption is that they are all executed on the same processor.

To speed up processing, parallel processors (or processor cores) could be executing the different versions simultaneously, turning the temporal redundancy into spatial redundancy. As in the case of hardware spatial redundancy, an external voter or arbiter is required to select the result that should be acted upon. A variation would put the voter into one of the existing processors, as another software process. This will introduce single points of failure in case the voting processor goes unexpectedly out of service.

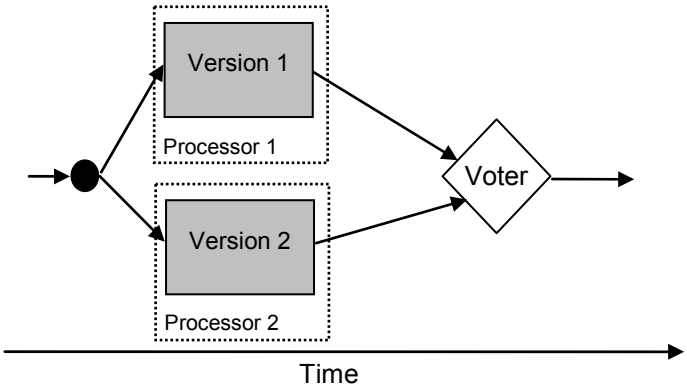


Figure 4. Multiple parallel executions with voting

Faults in the processor hardware will result in the versions behaving differently. But if the processors are error free, and the inputs are identical, then the same version of the software code will produce the same results. Those results might be either correct or incorrect. Fault prevention in the software development will have resulted in high quality software which is nearly error free. So these two versions should always compute identical and correct results. However, it is widely believed that no software is 100% fault free. When a software defect is encountered both versions will make the same errors.

To reduce this risk of all the software containing the same faults, use different versions of the software. At its simplest, have the same development team produce two (or more) versions of the software without copying anything from one version to another. This can still result in the same fault being present in the multiple versions, since the fault might have come from a misunderstanding of the system's specification. Unless this misunderstanding is corrected then all the versions will have the same incorrect design and implementation.

To further reduce the risk of the same fault being introduced into multiple versions, use different development teams to produce the different versions of the software. If they are allowed to collaborate and share their understanding of the specifications however then misunderstandings and errors can still result.

Therefore,

Build the system using "n" different development teams to interpret the specification and to implement it. Use different programming languages and different implementation systems to prevent language specific faults from being present in the completed system.

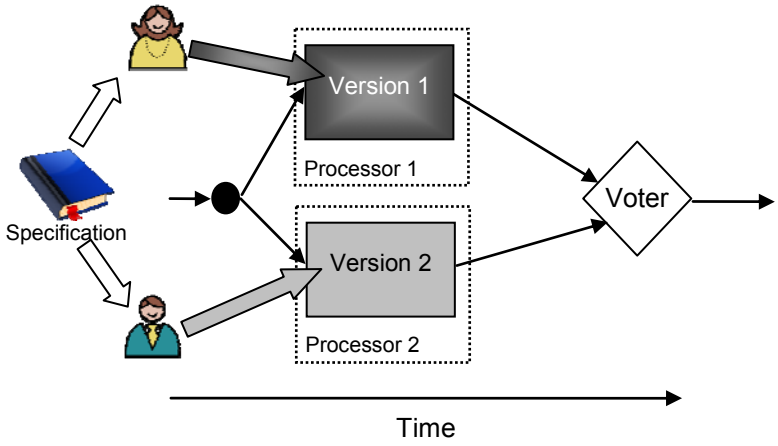


Figure 5. N-Version Programming

❖ ❖ ❖

N-version programming was first invented by Chen and Avizienis [1978]. It has been much discussed in the literatures; the references contain only a snapshot of the related publications.

Serious questions about the effectiveness have been raised in different published studies, showing that common errors do occur between separate, different development teams. [Knight and Leveson 1986; Knight and Leveson 1990; Cai et. Al 2005] Even given this criticism the technique is still widely viewed as effective. …

ACKNOWLEDGEMENTS

Rejuvenation

Thanks to shepherd Joe Yoder and the PLoP Reliability and Trust workshop group of Eunsak Kang, Christof Hannebauer, Cedric Bouhours, Kiran Kumar, Ernst Oberortner, Matt Hansen and Vivek Gondi for their helpful suggestions.

Count the Black Sheep

Thanks to shepherd Wolfgang Herzner and the PLoP Security and Quality writer's workshop group of Ed Fernandez, Amir Raveh, James Nobel, Brian Foote, David Pearce, Steven Hill, Yuji Kobayashi, Hironari Washizaki, and Takao Okubo for their helpful suggestions.

N-Version Programming

Thanks to Anjali Das who shepherded N-VERSION PROGRAMMING. Also Eduardo Fernandez and Michael Pont who offered suggestions during shepherding and to my PLoP Writer's Workshop group.

REFERENCES

Babbage, C. "On the mathematical powers of the calculating engine," December 1837 (unpublished manuscript) Buxton MS7, Museum of the History off Science, Oxford. In B. Randell, editor. The Origins of Digital Computers: Selected Papers. Springer, New York, pages 17-52, 1974.

Chen, L., and A. Avizienis, "N-Version Programming: A Fault-Tolerance Approach to Reliability of Software Operation," Digest of Papers FTCS-8: Eight Annual International Conference on Fault-Tolerant Computing, Toulouse, pp. 3-9 (June 1978).

G. Candea, S. Kawamoto, Y. Fujiki, G. Friedman, and A. Fox, 2004. Microreboot — A technique for cheap recovery. In Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation - Volume 6 (San Francisco, CA, December 06 - 08, 2004). Operating Systems Design and Implementation. USENIX ssociation, Berkeley, CA, pp 3-3.

Cai, X, M. R. Lyu and M. A. Vouk. "Experimental Evaluation of Reliability Features of N-Version Programming." Proc. 16th IEEE Intl. Symp. on Software Reliability Engineering, Nov. 2005, pp 161-170.

crontab. Linux crontab(1) manual page. http://www.linuxmanpages.com/man1/crontab.1.php

"MTTR '>>' MTTF", Armando Fox, June 2002 ROC Retreat, http://roc.cs.berkeley.edu/retreats/summer_02/slides/fox.pdf, accessed August 30, 2010, unpublished.

M.Grottke, A. P. Nikora and K. S. Trivedi, 2010, "An Empirical Investigation of Fault Types in Space Mission System Software," Proceedings of the 2010 IEEE/IFIP International Conference on Dependable Systems and Networks (DSN), IEEE/IFIP, Chicago, IL, 2010, pp 447-456.

Hanmer, R. Patterns for Fault Tolerant Software. Chichester, UK: John Wiley & Sons, 2007.

Y. Huang, C. Kintala, N. Kolettis, N. D. Fulton, 1995, Software Rejuvenation: Analysis, Module and Applications, Twenty-Fifth International Symposium on Fault-Tolerant Computing (Pasadena, CA, IEEE Computer Society, 381-390

R. Hanmer, V. Mendiratta, 2010, Rejuvenation with workload migration, proceedings of 2nd Workshop on Proactive Failure Avoidance, Recovery, and Maintenance (PFARM), Dependable Systems and Networks, IEEE/IFIP, Chicago, IL, IEEE/IFIPS, 2010, pp 80-85.

Knight, J. C. and N. G. Leveson, "An Experimental Evaluation of the Assumption of Independence in Multi-version Programming," IEEE Transactions on Software Engineering, Vol. SE-12, No. 1 (January 1986), pp 96-109.

Knight, J. C. and N. G. Leveson, "A reply to the criticisms of the Knight & Leveson experiment," SIGSOFT Softw. Eng. Notes 15, 1 (Jan. 1990), 24-35.

K. S. Trivedi, K. Vaidyanathan, and K. Goseva-Popstojanova. 2000. Modeling and Analysis of Software Aging and Rejuvenation. In Proceedings of the 33rd Annual Simulation Symposium (April 16 - 22, 2000). IEEE Computer Society, Washington, DC, pp. 270-279.

Laprie, J. C. Dependability: Basic Concepts and Terminology. New York: Springer-Verlag, 1991, p 4.

Monit. http://mmonit.com/monit/ accessed July 24, 2010.

Meyers, M. N., W. A. Routt, and K. W. Yoder, "Maintenance Software," Bell System Technical Journal, Vol. 56, No. 7, September 1977, pp 1139-1167.

Service Reanimator: http://www.veloci.dk/index.asp?visnu=srea/srea.htm accessed Sept. 12, 2010.

K. Vaidyanathan, K.S. Trivedi. 2005. A Comprehensive Model for Software Rejuvenation. IEEE Trans. Dependable Secur. Comput. 2, 2 (Apr. 2005), 124-137.

J. Yoder, personal communication, July 23, 2010.