Software Rejuvenation

Robert Hanmer
Alcatel-Lucent
2000 Lucent Lane
Naperville, IL 60653 USA
robert.hanmer@alcatel-lucent.com
hanmer@acm.org

Abstract

Software rejuvenation is a technique of proactive fault tolerance that designs the system for periodic reboots. Micro-rejuvenation is a technique of small reboots done frequently to extend the time without a failure. This paper contains three patterns. Micro-rejuvenation should be the primary focus of shepherding and workshopping for PLoP 2010. The other two have been previously workshopped.

While working on a paper for another conference Veena Mendiratta and I started looking at "proactive fault recovery". This concept seems like a new thing but has been around and published since the mid-90's. It was a concept that I felt didn't fit within the immediate, direct context of my book **Patterns for Fault Tolerant Software**, so no proactive fault recovery patterns were included there. Since its publication 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 book focused on techniques that individuals or small teams could implement themselves. This paper contains patterns on REJUVENATION, N-VERSION PROGRAMMING and using ERROR COUNTING as a form of fault isolation.

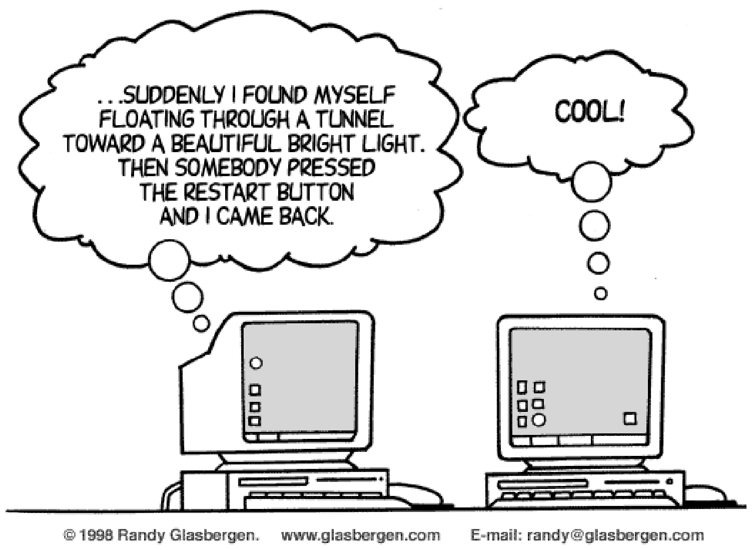
Researchers no longer study faults as a single type of thing. Instead software faults are classified into several types based upon their characteristics. "Bohrbugs" are faults that activate consistently in well-defined circumstances. These include programming faults that could be detected during code inspection and testing because they are fixed are 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 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. A fault is either a Mandelbug or a Bohrbug.

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. [GNT10]

For PLoP 2010, the emphasis of the workshop should be on the pattern Micro-Rejuvenation since the other patterns have been workshopped at previous PLoPs.

1. Rejuvenation[1]


...SUDDENLY I FOUND MYSELF FLOATING THROUGH A TUNNEL TOWARD A BEAUTIFUL BRIGHT LIGHT. THEN SOMEBODY PRESSED THE RESTART BUTTON AND I CAME BACK.
COOL!
© 1998 Randy Glasbergen. www.glasbergen.com E-mail: randy@glasbergen.com

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.[2]

❖ ❖ ❖

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

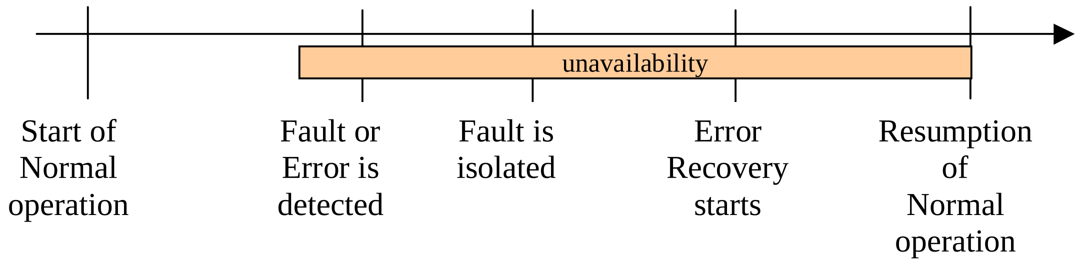
The normal stages of software life begin with its initialization and the beginning of its processing. It works flawlessly for a time (hopefully a long time). While it is working it is slowly degrading. Little pieces of unused memory are reserved for uses that never occur, bits of information are saved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

[1] This pattern is based on Microreboot work first published by Candea and Fox [CKFFF04]. The general concept of rejuvenation was first described from Bell Labs by Kintalla et. Al [HKKF95].

[2] Failures are deviations from correction operation as defined in the specifications. They are what the customer sees. Faults on the other hand are the actual defects.

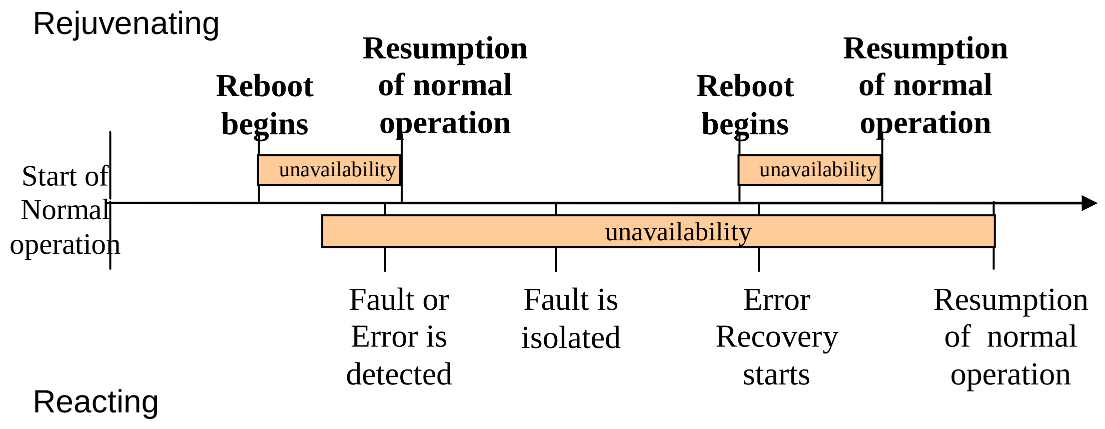
resource, or get confused by the existence of unused data. When this occurs, an error is likely to happen which might be followed by some type of failure.

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)[3]. This all takes time, as shown in Timeline I.


unavailability
Start of Normal operation
Fault or Error is detected
Fault is isolated
Error Recovery starts
Resumption of Normal operation

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 planned downtime. This *Rejuvenation* is the graceful shutdown and the reboot of the
Softwar


Rejuvenating
Reboot begins
Resumption of normal operation
unavailability
Reboot begins
Resumption of normal operation
unavailability
Start of Normal operation
unavailability
Fault or Error is detected
Fault is isolated
Error Recovery starts
Resumption of normal operation
Reacting

[3] Numbers within parenthesis refer to the reference number of the pattern with **Patterns for Fault Tolerant Software** [Hanmer07].

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 recover from an fault that activates. MTTF is the time between fault activations. Systems design for high availability have high MTTF, which makes validating MTTF difficult. It is usually done by monitoring many systems for a period of time and extrapolating the failure results, or through statistical methods. MTTR is much easier for the customer to monitor; the customer sees how effective and timely recovery is accomplished. The total of short periods of planned rejuvenation can be less than the time it takes to recover from a single fault. [Fox-ROC ppt]

Deciding when to reboot can be done in the moment, for example when resources are get low indicating that there is a memory leak about to become an error. [HKKF95] If the rejuvenation is triggered base on execution time detection then SOMEONE IN CHARGE (8) is used to monitor and trigger the rejuvenation.

The open source tool Monit [Monit10] 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 daemon is restarted if the CPU usage is over 80% for five cycles or the memory usage over 100Mb for five cycles[Yoder10]:

```
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 empirical measurements during development testing or by modeling. A hybrid approach uses SOMEONE IN CHARGE (8) to periodically take measurements that are used to dynamically compute when rejuvenation should occur. [KVG00, VT05] How it gets decided is another issue altogether that is the study of academic research.

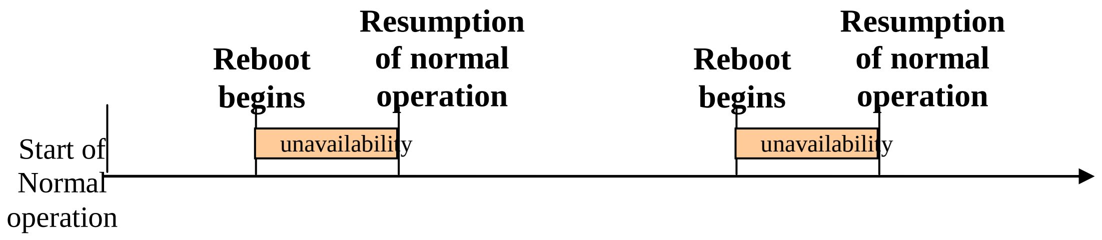
Rejuvenation can occur at any of several scope: system, application, process, or thread. The entire system can be restarted or the tiniest part of an execution flow can be restarted. The fastest rejuvenation will occur at the smallest scope, but rejuvenation at that scope is not always possible.

Rejuvenation doesn't require REDUNDANCY (1) in the system. If the part of the system to be rejuvenated is redundant than it is possible to rejuvenate without losing any service. [HM10]

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.


Resumption of normal operation
Reboot begins
Resumption of normal operation
Reboot begins
Start of Normal operation
unavailability
unavailability

Timeline III: Rejuvenation

If rejuvenation is not done often enough the system will suffer unplanned downtime as it goes through the detection, isolation and recovery cycles. 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, Service Reanimator [Srea10] watch 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 the error and stops and restarts the process before it has a chance to stop erroneously.

Rejuvenation helps with aging-related Mandelbugs. Sometimes it will help 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.

❖ ❖ ❖

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)).

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

Rejuvenation can be implemented by mechanisms as simple as a cron job that watches for signs of degradation and causes a restart or reboot. An example implementation watches for virtual machines to run out of resources and restarts it before it fails. [Yoder10] Open source tools such as Monit [Monit10] 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, for example a database server might 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.

…

2. COUNT THE BLACK SHEEP[4]

Also known as "Error Counting"

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

… Errors are occurring in the system. You might have been processing them all or you might have been RIDING OVER TRANSIENTS (26) and ignoring them.

You're 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 responsibility, 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.

❖ ❖ ❖

Errors keep occurring. You are trying to identify where they are originating (i.e. what fault activated) so that the appropriate error processing can be applied. Identifying the fault is needed before it can be treated.

Faults, especially software faults, will reoccur and cause errors whenever they receive the same stimuli. If nothing is done to correct the faults or to avoid the stimuli the errors will reoccur. Reoccurring errors increase the likelihood that a failure will be observed. After errors reoccur enough times you can start to see the pattern [sorry for the pun] and can treat them correctly.

---

[4] This strategy is discussed in 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.

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.

When errors occur the system can either process each error, 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.

If an error occurs once 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 counts 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 implies 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 software.

The system might be logging the 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 the error occurs and is correlated with a certain fault a record is sent to the logging system. The logging system can provide the longer term storage of what errors have occurred.

Therefore,

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

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] is an effective technique to create the list of faults which should be counted,

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 in prioritization of treatment must not lead to a false sense of security and to delays in error processing. Error tabulating and error processing must done in parallel with the objective of making the system the most reliable.

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. It needs to be reliable and have as few 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 an evolutionary product.

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.

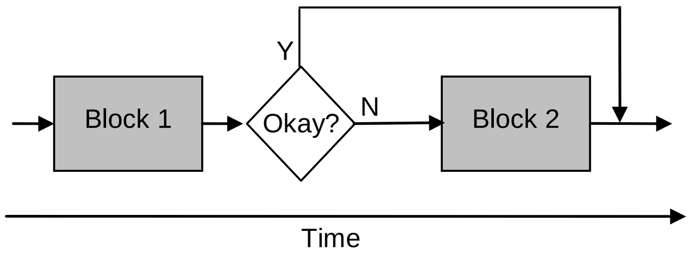
❖ ❖ ❖

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

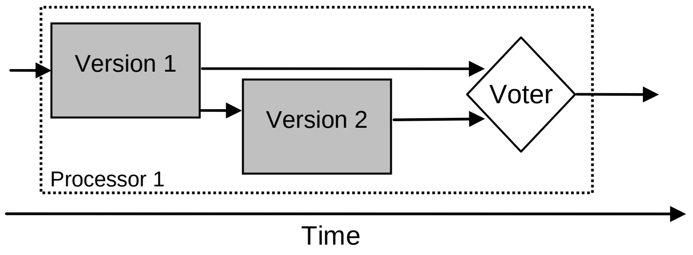
Redundancy in software is typically done temporally. A common way of implementing this is through a Recovery Block (4) structure. In this structure a version of the code is executed and the results checked. The check determines if the result is correct, or within acceptable parameters. If the check passes 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 ().
```

Two representations of Recovery Blocks (4)

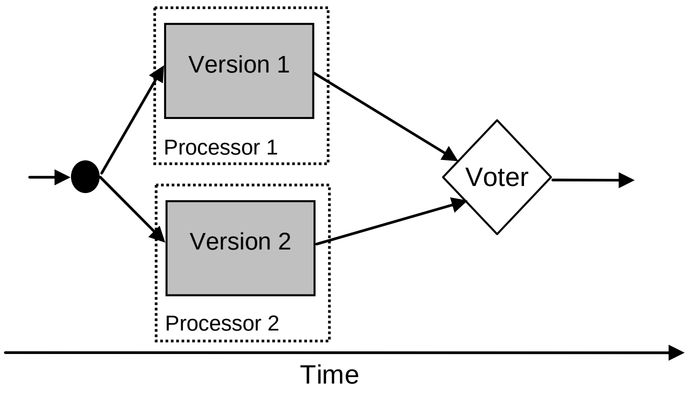
Another way of achieving software redundancy is to execute multiple versions of the same code serially. The results are compared through Voting (21) to select the result to be acted upon. The versions that are executed can be either the same version or different versions. The advantage of using the same 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.



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 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.



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 [CITE]. When a software defect is encountered both versions will make the same errors.

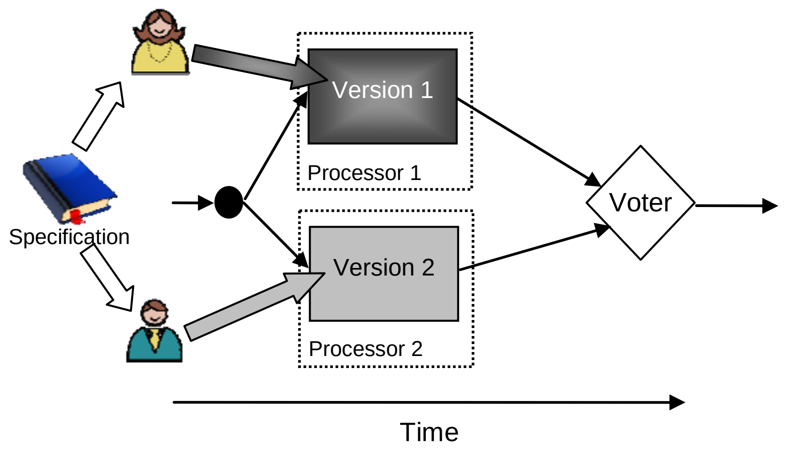
To reduce this risk of all the software containing the same faults, use different versions of the software. At a its simplest, have the same development team produce two (or more) versions of the software without copying anything from one version to another. This can still result in the same fault being present in the multiple versions, since the fault might have come from a misunderstanding of the system's specification. Unless the misunderstanding is corrected then all the versions will have the same incorrect design and implementation. 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. [Ba37]

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 languages and different implementation to prevent language specific faults from being present in the completed system.



N-Version Programming

❖ ❖ ❖

N-version programming was first invented by [CA78]. It has been much discussed in the literatures, the references contain only a snapshot of the related publications. Known uses:

Serious questions about the effectiveness have been raised in different published studies. What questions? Even given this criticism the technique is still widely viewed as effective. …

References

[GNT10] 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.

Micro-Rejuvenation

[CKFFF04] 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.

[Hanmer07] Hanmer, R. Patterns for Fault Tolerant Software. Chichester, UK: John Wiley & Sons, 2007.

[HKKF95] 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

[HM10] 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.

[KVG00] 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). SS. IEEE Computer Society, Washington, DC, 270.

[Monit10] Monit. http://mmonit.com/monit/ accessed July 24, 2010.

[Srea10] Service Reanimator: http://www.veloci.dk/index.asp?visnu=srea/srea.htm accessed Sept. 12, 2010.

[VT05] K. Vaidyanathan, K.S. Trivedi. 2005. A Comprehensive Model for Software Rejuvenation. IEEE Trans. Dependable Secur. Comput. 2, 2 (Apr. 2005), 124-137.

[Yoder10] Joe Yoder, personal communication, July 23, 2010.

N-Version Programming

[Ba37] 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.

[CA78] 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).

[CLV05] 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.

[Hanmer07] Hanmer, R. Patterns for Fault Tolerant Software. Chichester, UK: John Wiley & Sons, 2007.

[KL86] 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.

[KL90] 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.