Why Local Redundancy is Better

Herbert Hecht
SoHaR Incorporated
Culver City, California
herb@sohar.com

Abstract:

The paper calls attention to the difference between redundancy of small scope (local redundancy) and large scope or global redundancy. It shows that local redundancy permits reduction of the maintenance effort that is particularly significant for distributed and dispersed systems. Effective measures for achieving local redundancy are described together with a methodology for assessing the redundancy structure for attainment of optimum benefits.

Key Words: Redundancy, scope of redundancy, distributed systems, dispersed systems, maintenance cost

1. Introduction

Distributed systems are proliferating and are being used in increasingly critical and complex operations. The criticality brings with it a demand for uninterrupted service, and the complexity brings with it a demand for highly skilled maintenance personnel. Frequently the nodes of the systems are widely dispersed and located in inhospitable environments. All these factors increase the cost of maintenance so that reduction of this cost becomes a key to success of the system.

In this paper we show that local redundancy, also called redundancy of small scope, can increase the period of maintenance free operation after a hardware failure, and that it permits the use of simple measures for software redundancy. Also, local redundancy, almost by definition, provides better identification of the initiating event of what can be a complex failure process, thereby pinpointing the maintenance requirements. This also permits assignment of maintenance personnel with specific skills to perform the required repair, a more economical operation than dispatching senior staff with broad troubleshooting experience.

In the section that follows immediately we show the benefits of redundancy of small scope for hardware, particularly for long operation without maintenance. Section 3 describes measures for local fault tolerance for software, and Section 4 examines the benefits of these measures for distributed systems. Section 5 is an overview of the practical steps that can be taken to design systems for effective use of local redundancy, and Section 6 presents conclusions.

2. Background

Hardware designers have long known that the failure probability of a redundant system can be reduced by partitioning it into a number of individually redundant subsystems[1]. The benefits are particularly significant for non-repairable systems, such as in spacecraft. An example of applying partitioning to a flight control system is shown in Figure 1.

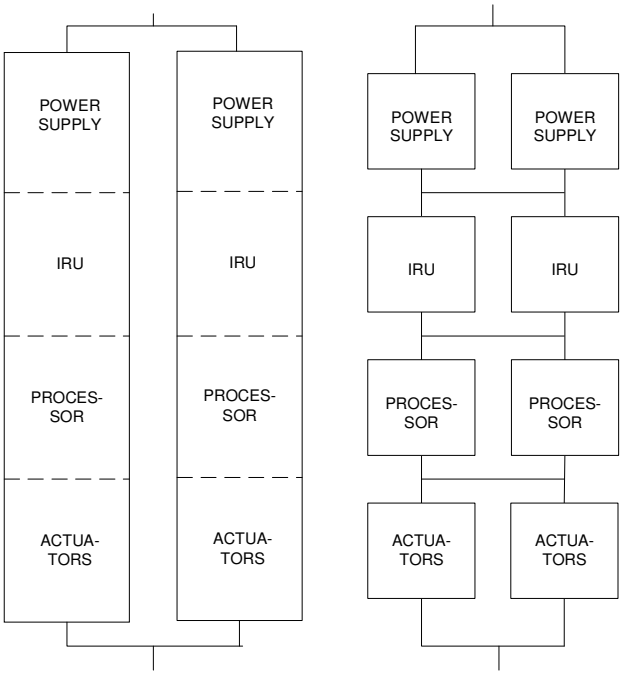

Figure 1 Dual and Partitioned Redundancy

A conventional dual redundant system is shown on the left and a partitioned system is shown on the right. We designate the failure probability of a single string of the dual redundant system by *f*. Then the failure probability of the entire systems if given by

$$F_D = f^2 \qquad \text{(Eq. 1)}$$

If each of the four components of the single string has the same failure probability (each *f/4*), the failure probability of the partitioned system on the right is

$$F_P = 4 \times (f/4)^2 \qquad \text{(Eq. 2)}$$

For exponential failure assumptions ($f = 1 - e^{-\lambda t}$) and $\lambda = 1$ the resultant failure probabilities as a function of time are plotted in Figure 2.

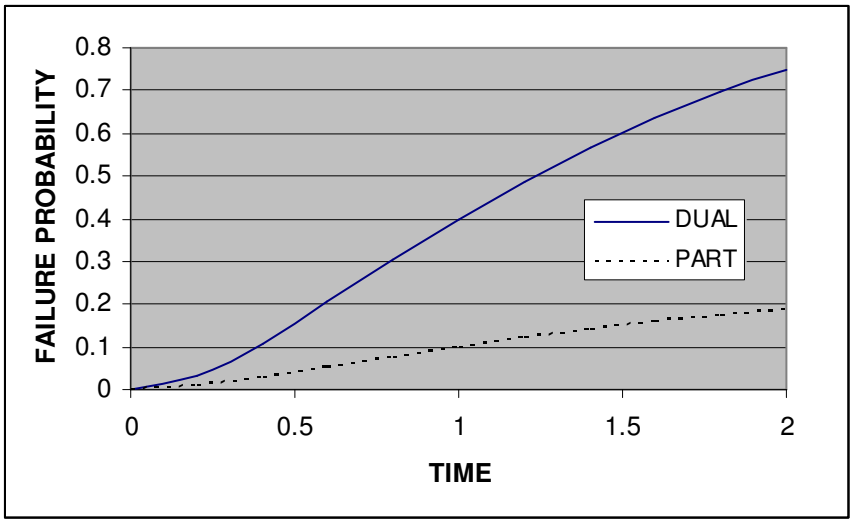

Figure 2 Failure Probability for Dual and Partitioned Systems

The benefits of partitioning are seen to increase with operating time. Remember that the graph deals with non-repairable systems. For repairable systems the benefit is largely dependent on the access time to repair. Where maintenance personnel and spare parts are always available the operation is near the left (zero) end of the time scale and the difference in failure probability between the two implementations of redundancy is very small. But for systems operating in the field, where repair may be delayed because of unavailability of technicians or replacement parts, there may be advantages of partitioning the system in order to obtain assurance of not losing the service due to failure of both parts of the redundancy structure. Also, as systems become more complex they require specialized test equipment and technicians with specific training, both of which can contribute to delays in the repair.

There is no free lunch. The reduced failure probability of the partitioned system is achieved by adding error detection and switchover provisions that are buried behind the horizontal connection lines if Figure 1. But we will show shortly that in software intensive systems error detection and recovery are usually easy to implement.

3. Software Applications

Redundancy in software does not necessarily take the form of replication shown in Figure 1. Replication with elements of identical design, which is the norm for hardware, is not desirable for software because program faults are usually in the design, and if the replicates share the design they will also share the faults. But there are alternatives to physical replication that are frequently preferable and these are discussed below.

Because these techniques achieve recovery almost instantaneously, the motivation for redundancy of small scope that is present in the hardware domain does not carry over. Rather it is the avoidance of dispersing erroneous data to communicating processes that constitutes the major advantage for local redundancy in software, and particularly for software in distributed systems. The listing of redundancy and error detection techniques is intended to show that there are many economical possibilities for implementing local error recovery.

While the examples of software redundancy that are described here are effective for permitting continuation of the overall program they do not correct the initial fault that caused deviation from routine execution. That correction requires analysis of the cause for the anomalous operation and subsequent revision of the program. To facilitate these steps it is necessary to capture the events leading to invocation of the redundancy provisions. Redundancy of small scope provides better localization of the entry into anomalous execution and is therefore of great help in finding and fixing the initial fault.

One of the simplest forms of redundancy that does not require replication is to *re-run* a program segment if an error has been detected. This is sometime referred to as *temporal redundancy*[2] and is effective when the failure was due a transient unavailability of a resource or due to loss of synchronization between data sources.

A more radical form of temporal redundancy is the *restart* of a major component, including re-initialization. This is effective against many forms of corrupted storage and communication anomalies. Re-start usually requires much more time to complete than is available in a single iteration of a real-time program. The system served must therefore be put into a safe state until the software component becomes available again. Some techniques for achieving this are shown in the following paragraphs.

Default values for output variables generated by a failed segment can frequently be used as a temporary measure to ride out the effects of a failure. Default values can be either pre-assigned or dynamically generated. A common example of the latter category is to use the previously generated value of a variable. A more sophisticated technique is to use the previous value plus an increment that represents the trend of the variable.

In a*nalytical redundancy* an approximation of the value of a currently inaccessible variable is generated as a function of accessible values of related variables. Examples are the use of airspeed plus a stored value of wind velocity to compute groundspeed, or the use of barometric altitude plus a stored value of height of the terrain to compute radar altitude. The latter implementation is shown in Figure 3. The right pointing arrow between the two circles is the stored terrain altitude. The analytical radar altitude is intended for short term use only.

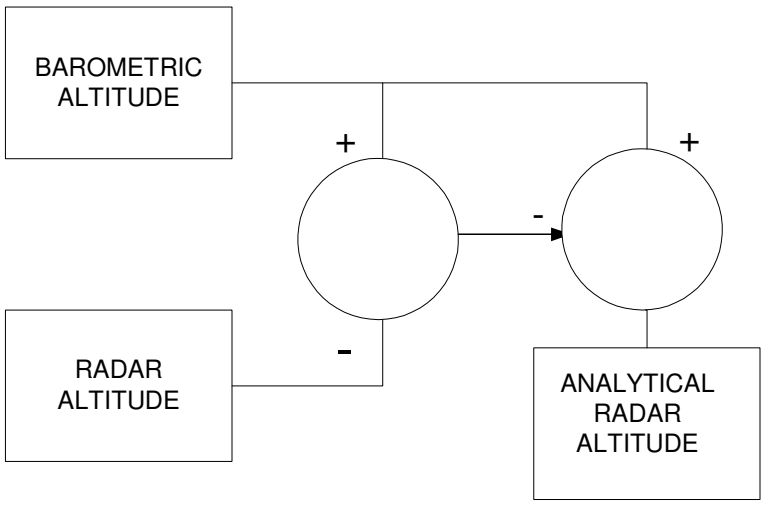

Figure 3 Analytical Redundancy

The use of an *alternate routine* represents the closest software analog to hardware redundancy. To avoid introduction of identical faults into the primary and the alternate routines the design of the latter should be as different as possible. Assignment to different programming teams reduces but does not eliminate the possibility of correlated errors in the primary and the alternate[3]. This form of redundancy is not restricted to two versions; multi-version (n-version) implementations have been described[4].

The description of these techniques so far has emphasized means of recovering from faults. In the following we will present some error detection techniques that can be used to invoke the recovery provisions. Corruption of data transmitted between processors or retrieved from storage can be identified by *error detecting codes* that will typically initiate a request for re-transmission or a repeat of memory access. Where *error correcting codes* are employed the correction can in most cases be accomplished locally without requiring the repetition. Codes can also be used to check the correctness of many arithmetic operations.

Assertions represent another large class of error detection mechanisms. In programs that compute *physical variables* (position and its derivatives, mass, energy, etc.) laws of conservation and continuity can be exploited to detect deviations from correct execution. An example of an assertion based on physical laws is

$$If \;\; v_1 - v_0 > A: error$$

where the difference between two velocities measured over a known time interval cannot be greater than the acceleration capability, A, of the vehicle propulsion plant.

For *financial transactions* assertions can frequently be constructed from the requirement that

New balance = old balance + deposits – withdrawals

This relationship can also be used for error detection in inventory control programs.

Where there are physical or conventional restrictions on *transition between states*, assertions that detect violation of these restrictions can detect errors in program execution. An example is that a telephone cannot transition from “busy” to “ringing”. When such a transition is detected an appropriate recovery routing should be invoked.

Watchdog timers may be thought of as implementing the assertion

$$Execution\ time < Max\ execution\ time$$

and represent a widely applicable means of errors due to improper loop termination criteria, poorly structured search routines, and omitted exception handling.

Inverse operations represent another important mechanism for error detection. Multiplication after a division represents a trivial example. A more practical one is multiplying the original and the inverse after a matrix inversion. Counting the original and the sorted set after a sort operation also falls into this heading.

4. Importance to Distributed Systems

Most of the findings presented above are particularly important for distributed systems for the following reasons

a. These systems are complex, usually require specialized test equipment and trained maintenance personnel, and they may be geographically dispersed, creating further difficulties for maintenance. These factors demand small scope of redundancy for the hardware to increase the survival probability if maintenance to repair a first failure is delayed.
b. The distributed and possibly dispersed attributes of these systems carry with them a high risk of exporting contaminated data and make it difficult to pinpoint causes of software failure. Local implementation of software redundancy reduces the risk with regard to both of these problem areas.
c. There is intense hardware-software interaction that makes it often difficult to identify the origin of a failure and makes failure detection at the local level very important.
d. There is a high degree of change activity for both hardware and software, making it mandatory that the redundancy provisions are encapsulated at the local level.
e. Looking into the future we expect both an increase in complexity and greater difficulty in providing the highly trained maintenance staff that will be required for these systems. Small scope redundancy increases the probability of unattended operation and facilitates both hardware and software maintenance.

5. Practical Steps

In the foregoing we have presented the advantages of local redundancy without dwelling on how it can be implemented. Now we will discuss a practical approach.
In hardware the starting point for evaluating the need for redundancy has frequently been a failure modes and effects analysis (FMEA) or an expanded form of it called failure modes, effects and criticality analysis (FMECA)[5]. Column headings for an FMEA worksheet are shown in Figure 4. To evaluate the need for redundancy we first examine the severity column (7) in which the Roman numeral I usually designates catastrophic system level failures (potential loss of life or severe injuries) and Roman II is used for the next most severe level, usually potential loss of mission. Where there is a significant (what is significant is decided by regulatory agencies or project management) probability of occurrence (8) of severity I and II failures, and where other means of reducing the failure probability have been exhausted, the system engineering staff will usually provide redundancy. This results in appropriate entries in the detection method (9) and compensation (10) columns. Compensation is the term used in this FMEA format for the back-up mechanism. The failure probability for a component after redundancy is approximately

$$F = F_0 \times (1 - c) \qquad \text{(Eq. 3)}$$

where F_0 is the pre-redundancy failure probability and c represents the coverage of the redundancy provisions (combined detection and compensation). Thus, if the coverage, c, is 0.99 $F = 0.01F_0$.

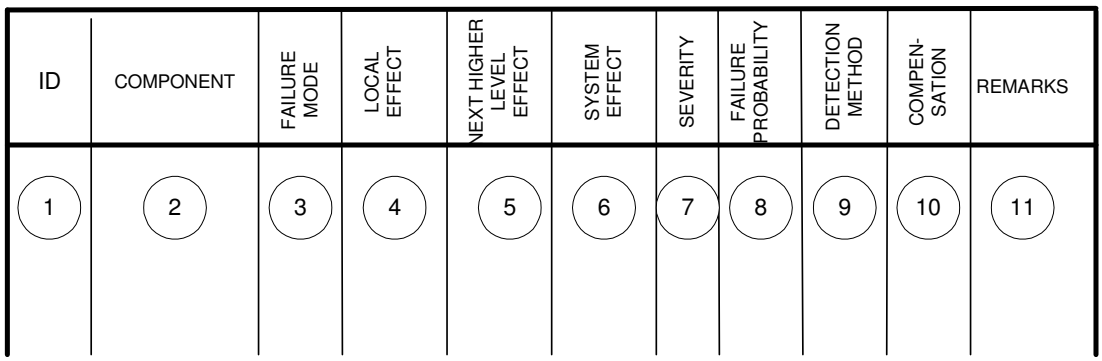

Figure 4 FMEA Worksheet

For hardware the entries into the components column (2) are obtained from a parts list or from a schematic. The same is not possible for software. Functional partitioning has been used[6] but there is no consensus on what is a "function". In particular, error handling may be omitted on a list of functions that is derived from a requirements document that typically concentrates on the routine operational aspects. Where software is developed with UML methodology[7] a definition of components can be derived from class lists, and SoHaR has developed a tool, MOVAT, for automating the generation of an FMEA from the UML database[8]. Use of this tool facilitates generating a combined hardware/software FMEA that provides excellent visibility into the error detection and compensation provisions. This is a suitable starting point for determining where additional local redundancy provisions need to be added. Since UML systems definitions (including hardware) are now being standardized, a comprehensive methodology for FMEA generation may become possible.

A review for allocation of redundancy provision in distributed systems using the methodology discussed in Section 4 will include the following steps

- Generate FMEA worksheets covering both hardware and software
- Identify high severity I and II failure modes with significant failure probability or, where a criticality analysis has been prepared, identify high criticality failure modes
- Determine whether detection and compensation provisions exist; if they do not, explore the feasibility of adding them
- Determine whether the detection provisions address the local failure effects; if they do not, explore the feasibility of adding detection closer to the local effects level
- Review the compensation provisions for compatibility with the detection provisions; the scope of the compensation should not be smaller than that of the detection, and preferably not much larger. Table 1 lists combinations of detection and compensation provisions that have been found to work well in at least some environments. The table may be tailored to suit the needs of the current project.

- Assess the logging and reporting provisions to verify that all error detection and compensation events are made visible to maintenance personnel and project management.

Table 1. Application of Error Detection and Compensation Measures

Compensation	Error Detection			
	Check Codes	W'dog Timer	Assertions	Inverse Oper.
Re-run	x	x	x	
Re-start	x	x	x	
Default Value		x	x	x
Analytical Redundancy			x	x
Alternative Routine			x	x

6. Conclusions

The range of applications of distributed systems is constantly expanding and complexity as well as criticality is increasing. The cost of maintenance is frequently a determining factor in the success of a system. Local redundancy can reduce the maintenance workload by

1. permitting longer periods of satisfactory system operation after a hardware failure
2. permitting less costly redundancy measures for software failures (e. g., re-running a routing vs. restart of a program)
3. providing a precise indication of the original failure event when cascading failures occur
4. requiring only the specialized maintenance skills for dealing with the local failure rather than the broader skills required for system trouble shooting

The key to designing redundancy such that it provides these benefits is a comprehensive FMEA, covering hardware and software. Tools for computer-aided generation of such an FMEA are becoming available. Their use may become an essential part of the design and application of distributed systems.

References

[1] H. Hecht and J. J. Stiffler, "Redundancy Allocation in the Fault Tolerant Spaceborne Computer", *Digest, Eighth Annual International Conference on Fault Tolerant Computing,* IEEE Cat No. 78CH1286-4C, p. 197 June 1978

[2] Algirdas Avizienis, "The Four-Universe Information System Model for the Study of Fault Tolerance", *Digest of Papers 12th Fault Tolerant Computing Symposium,* IEEE Cat 82CH1760-8, pp. 1157-1191

[3] D. E. Eckhardt, A. K. Caglayan, J. C. Knight, et al., "An experimental evaluation of software redundancy as a strategy for improving reliability", *IEEE Trans. Software Engineering,* vol 17 no 7, July 1991, pp. 692 - 702

[4] A. Avizienis and J. P. J. Kelly, "Fault Tolerance by Design Diversity: Concepts and Experiments", Computer, v.17 no. 8, pp. 67-80, August 1984

[5] U. S. Department of Defense, MIL-STD-1629, "Procedures for Performing a Failure Mode, Effects and Criticality Analysis"

[6] Bowles, J. B. and Chi Wan, " Software Failure Modes and Effects Analysis for a Small Embedded Control System**"**, *Proc. of the 2001 Reliability and Maintainability Symposium,* Philadelphia PA, January 2001, pp. 1 – 6.

[7] Boggs, Wendy and Michael, *Mastering UML with Rational Rose*, Sybex, 2002

[8] Herbert Hecht, Xuegao An and Myron Hecht, "Computer-Aided Software FMEA for UML Based Software", *Proc. 2004 Reliability and Maintainability Symposium",* Los Angeles, January 2004