

# ***Software Implemented Fault Tolerance – the ESVP Approach***

Goutam Kumar Saha

Senior Member IEEE

CA – 2 / 4 B, Baguiati, D.B. Nagar, Kolkata 700059, India.

[gksaha@rediffmail.com](mailto:gksaha@rediffmail.com) [sahagk@gmail.com](mailto:sahagk@gmail.com)

**Abstract:** This short paper describes a low cost technique for gaining software implemented fault tolerance without using design diversity based N versions redundancy in both software and hardware. The proposed approach uses an enhanced single-version programming (ESVP) scheme for an application that executes on a single machine. ESVP employs triplicate application program along with multistage voting at inputs and outputs in order to tolerate single-point operational faults in input data, output data, an individual processing module and in an individual voting module through error masking. It is not aimed to tolerate software design bugs. This is a useful tool for designing a reliable low-cost application system.

## ***1. Introduction***

Computer software has rapidly become an important and indispensable element in many aspects of our daily lives. In recent years, producing reliable software for real – time control systems has become a major interest of the industrial as well as the academic world. We expect such systems to operate reliably, even under extremely severe conditions. No matter how thoroughly we test, debug, modularize and verify design bugs will still plague our software. Moreover, software often gets corrupted or it experiences random errors in codes and data, while running in an industrial environment. Whatever measure we may take it is very difficult to make an industrial environment free from the potential threats of various electrical transients or noises. Software failures may lead to partial or total system crashes. In some cases for example, in Air - Traffic – Control, Nuclear – Power – Plant – Control systems, Space – Vehicle system, such crashes will

cost money, lives and can have other societal impacts. Our dependency on software continues to escalate. Software size and complexity continue to grow and find many new applications namely, safety assurance systems for nuclear power, life support systems in health care etc. Owing to the critical nature of these new applications, operational reliability is of paramount importance. Therefore, to achieve ultra reliability in industrial computing, it is must to adopt the strategy of defensive programming based on redundancy. It is referred to as fault – tolerant software. While industries clearly understand the need for fault tolerance, some hesitations occurs in actual practice due to (a) the additional cost of redundancies, (b) software complexity, and (c) the environment in which the system operates. However, with the current growth of software system complexity, we cannot afford to postpone the implementation of fault tolerance in critical software application areas specifically in various industrial control systems. Redundancy has been accepted as a viable approach for obtaining reliability with unreliable components. Fault – Tolerant software ensures system reliability by using protective redundancy at the software level.

Designing a low-cost software based fault tolerance tool has always been a challenging task to produce an affordable and reliable application system. All techniques for achieving fault tolerance depend upon the effective deployment and utilization of redundancy. Hardware - redundancy is normally concerned with extra transistors, logic gates, memory units, power supplies and the like, while software redundancy referred only to the extra programs provided to either to mask errors or to detect and recover errors. *Hardware redundancy* has been categorized into static (or masking) redundancy and dynamic redundancy. For static redundancy, redundant components are used within a system to provide fault tolerance so that the effects of a component failure are masked from, and not apparent to, the environment of that system. In contrast,

dynamic redundancy is employed just to provide an error detection capability within a system, and has to be supplemented by redundancy elsewhere in order to achieve fault tolerance. For example, the use of error-correcting codes in memory units can be regarded as an application of static redundancy. A simple parity code or checksums can be viewed as *dynamic redundancy* since tolerance of a memory fault could require redundancy elsewhere, for instance in the code of the operating system. The canonical example of the use of static redundancy is *Triple Modular Redundancy* (TMR) or an N-Modular Redundancy (NMR), where three or N identical sub-components and a majority voting circuit are used. If a majority among all outputs is reached, the minority outputs are all masked out. A TMR system consists of three copies of a module (each of identical design) and voting circuits to check the outputs of these identical modules and to select the value of the majority. The TMR system is therefore designed to tolerate the failure of any single copy of a module by only producing output on which at least two modules agree.

Fault masking is a structural redundancy technique that completely masks faults within a set of redundant modules. A number of identical modules execute the same functions, and their outputs are voted to remove errors created by a faulty module. Triple modular redundancy (TMR) is a commonly used form of fault masking in which the hardware circuitry is triplicated and voted. A TMR system fails whenever two modules in a redundant triplet create errors so that the vote is no longer valid. Hybrid redundancy is an extension of TMR in which the triplicated modules are backed up with additional spares, which are used to replace faulty modules allowing more faults to be tolerated. Voted systems require more than three times as much hardware as non-redundant systems, but they have the advantage that computations can continue without interruption when a fault occurs, allowing existing operating systems to be used. Dynamic recovery is required when

only one copy of a computation is running at a time (or in some cases two unchecked copies), and it involves automated self-repair. As in fault masking, the computing system is partitioned into modules backed up by spares as protective redundancy. In the case of dynamic recovery however, special mechanisms are required to detect faults in the modules, switch out a faulty module, switch in a spare, and instigate those software actions (rollback, initialization, retry, restart) necessary to restore and continue the computation. In single computers special hardware is required along with software to do this, while in multi-computers the function is often managed by the other processors.

Unlike a single voting module in a typical TMR system, the proposed ESVP scheme deploys multistage voting modules or multistage voters (as shown in fig.1) in order to mask individual voter failures. The ESVP system consists of four stage voters and triplicate processing modules. The ESVP has three voting modules for the input data. *The voting circuitry is also triplicated so that individual voter failures can also be corrected by the voting process.* As input data to an ESVP based application system, we use redundant copies of input data in order to mask erroneous input. Each of the three input voting modules (or the 1<sup>st</sup> stage voters) votes upon the copies of input data for majority inputs. The output data from each of the three voting modules are voted upon by a voting module (or the 2<sup>nd</sup> stage voter) to provide a correct input to the three identical copies of an application processing module. Three identical output voters (or the 3<sup>rd</sup> stage voters) vote upon the computation answers or output data from each of the three identical processing modules. Redundancy in processing modules is to mask erroneous computation. Redundancy in output voters is to mask individual output voter failures. The majority output data from each of the three output voting modules are voted upon by a final output voter module (or the 4<sup>th</sup> stage voter) in order to produce the final correct output from the ESVP system. Thus,

input voting modules mask errors caused by an individual faulty input voting module and the output voting modules mask errors caused by an individual faulty output voting module. Each of the eight voters is identical to each other. Each voter has three inputs and one output. In case input data are of changing nature for a typical application system, we need to buffer the input data. Like a TMR or NMR system, the ESVP system also relies on single design only. In other words, the ESVP system does not employ design diversity, whereas conventional N-Version Programming (NVP) scheme relies on N different versions (or N different designs) of a module. As the ESVP scheme uses three copies of a processing module (each of identical design), this is not as costly as a 3-version programming scheme is. Again, in TMR (or NMR), each identical module executes simultaneously or in parallel on three (or N) identical machines. But in ESVP, each identical module sequentially executes on a single machine only. Thus, in ESVP scheme, hardware cost also comes down to one-third of a TMR system. In an NVP scheme, N different versions of a module (each of different design) executes simultaneously on different N machines. Thus, in ESVP scheme, software development cost also comes down to one-third of an NVP scheme with three different versions. Similarly, hardware costs also comes down to one-third of an NVP system. The ESVP scheme when implemented on a single machine, is a low cost fault tolerant scheme that has an affordable overhead on time and memory space. However, the ESVP scheme, if implemented on three identical machines to run copies of an application simultaneously, will have almost similar overhead on both the execution time and memory space in comparison to the TMR or NVP scheme. Little extra execution time is to be afforded for multistage voting only. The proposed approach relies on replicas based on a typical triple modular redundancy (TMR) in input, output, and application-program and in voting code (as described in figure 1) also.

## ***2. Various Approaches to Fault – Tolerant Software***

A simple TMR uses three identical modules that receive identical inputs and should produce three identical outputs. A voter compares the outputs from the three modules and if all agree voter produces unanimous output. If a single fault occurs then voter gives output produced by majority. This is useful to tolerate a single point failure only. A Recovery *Block Scheme (RBS)* comprises three elements: (a) one primary *module* for executing critical software functions, (b) an acceptance *test* in order to test the primary module's output after each execution, and (c) one set of alternate modules performing the same function of the primary module. Here, we expect that the test condition will be met by the successful execution of either primary module or the alternate modules. When an acceptance *test* detects a primary module failure, an alternate module executes, and so on. If all the modules are exhausted or failed then the system crashes. In RBS modules are executed sequentially. This scheme is similar to the dynamic *redundancy* in hardware.

In the *N – Version Programming Scheme (NVPS)*, *N – independent* programs execute in parallel on identical input, and results are obtained by voting upon the outputs from individual programs. In order to ensure the development of independent program versions, we must use different algorithms, techniques, programming languages, environments and tools in each effort. In this scheme modules are executed in parallel. In critical systems with real – time deadlines, voting at program's end (as in the basic NVPS) may not be acceptable. Therefore, voting at intermediate points is must for real – time systems.

In the *Community – Error – Recovery Scheme (CERS)*, we need to compare results at intermediate points of computation. It offers higher fault tolerance than the basic NVPS.

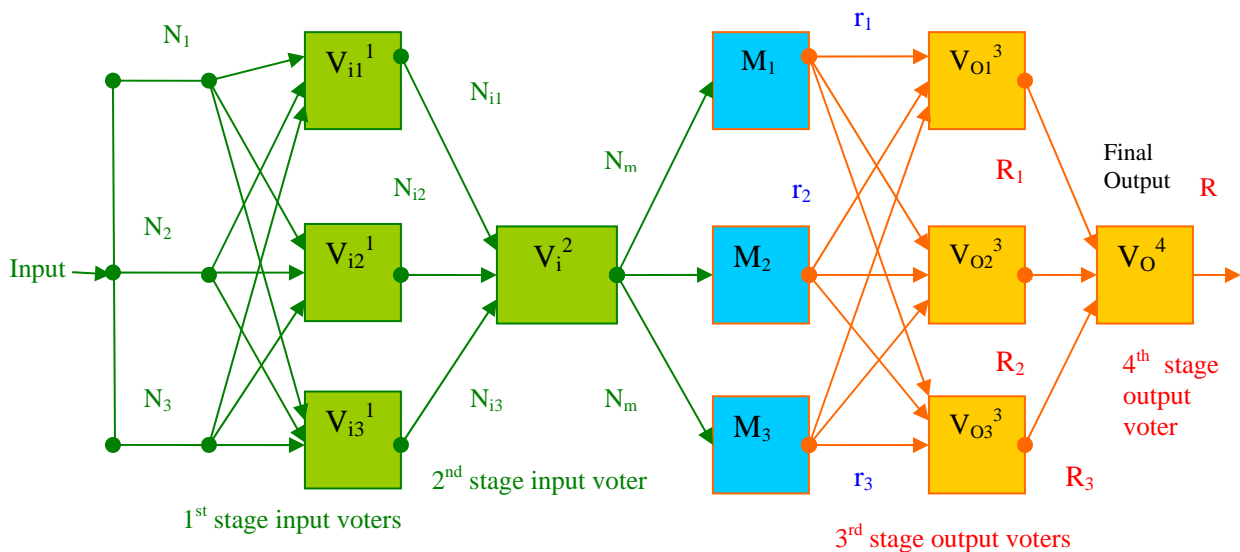
However, this approach requires the overhead in synchronizing the various software versions at comparison points. Here, on – line detection is carried out by an acceptance test.

An *N – Self – Checking Version Scheme (NSCVS)* adopts the intermediate voting at which every iteration is subject to an acceptance test or comparison checking. Whenever a version raises an exception, the correct output can be obtained from the remaining versions and then the execution of the industrial application software continues. Readers might refer to the works of [Lyu, Littlewood] for more related information.

### ***3. The ESVP Scheme***

In an *Enhanced – Single – Version - Programming (ESVP) Scheme*, the author proposes the use of replicas of a single enhanced version of an application program along with multistage voting codes for inputs and outputs. We store input data to an ESVP based application on triplicate input data variables before processing starts. Redundancy in input data variables is to mask an erroneous input data through the 1<sup>st</sup> stage input voters. Each voter of the 1<sup>st</sup> stage input-voters ( $V_i$ ) votes upon the three input data variables to get a majority input data. The 2<sup>nd</sup> stage voter votes upon the three input data in majority, coming from three input voters at the 1<sup>st</sup> stage. The majority-input data is passed on to each of the three identical processing-modules (M). Each processing module processes the input data sequentially (one after another) and produces output data and stores on a output data variable. At the end of computation of all the processing modules, we get three output values that are already stored on three different output data variables. Ideally, these three output data should be identical. These three output data are fed to each of the three output voters at the 3<sup>rd</sup> stage voters and each voter at this stage, votes upon the output data and outputs a majority output value. Thus, we get three major output data (one from each of the 3<sup>rd</sup> stage voter) from the 3<sup>rd</sup> stage output voters. These three major output data are

voted upon by the 4<sup>th</sup> stage output voter module to generate the final major output data and this is treated as the output of the ESVP system. This is enriched with enhanced processing logic for self-checking code [Goutam Kumar Saha] in order to produce correct outputs through error masking. This ESVP scheme is capable of masking an individual erroneous input or output data and an individual erroneous voter and an individual processing module. The proposed ESVP scheme is also capable of masking transient errors in input and output channels. This scheme is a low cost and an efficient solution in order to design in high fault tolerance. This scheme does not rely on multiple versions of program and hardware; rather it uses only an enhanced version of the application program. Again, it does not need multiple computing machines. It needs only one machine in order to run the application. This work is not intended to mask software design bugs; rather it masks environmental and operational faults during the life cycle of an application system. The ESVP scheme has been illustrated in the following figure-1.



**Figure 1. An ESVP Scheme with Multistage Voting**

### ***3.1. Discussion on ESVP***

In the above figure,  $V_{i1}^1$  stands for the 1<sup>st</sup> copy of a voting module for input data at stage-1.  $V_{O1}^3$  stands for the 1<sup>st</sup> copy of a voting module for output data at stage-3 and  $M_1$  stands for the 1<sup>st</sup> copy of an application module.  $M_2, M_3$  are the second and third copies of  $M_1$ . The first stage inputs-voting modules vote upon the triplicate input data ( $N_1, N_2, N_3$ ) and the second stage input-voting module is used to get the input data in majority ( $N_m$ ) on tolerating single point of failure in inputs. Replicas of  $V_i, V_o$  and  $M$  have been used in order to tolerate single point of failures at these respective modules. Each of the triplicate application program module (M) executes on similar input data ( $N_m$ ). The corresponding outputs ( $r_1, r_2, r_3$ ) from each identical application module are voted upon by the 3<sup>rd</sup> stage identical voting modules ( $V_{O1}^3, V_{O2}^3, V_{O3}^3$ ) and the corresponding outputs ( $R_1, R_2, R_3$ ) from these voting modules are again voted by the 4<sup>th</sup> stage voting module ( $V_O^4$ ) for producing final correct output ( $R$ ) on tolerating single-point failure in voting modules. Multistage voting modules have been used to tolerate single point of failure in inputs (input signals might experience delays, errors in converting etc.) or outputs, and in their corresponding voting codes. If all inputs to a voter agree, then a voter produces unanimous output. Otherwise, if a single fault occurs then voter gives output by majority. Triplicate application modules (M) have been used to tolerate single point failure at the application code through masking permanent or transient bit errors during the operation of the system. We need triplicate application modules to tolerate one permanent or transient fault. In order to tolerate two faults, we need five replicas of an application module. In other words, in the ESVP scheme, we need at least  $f + 2$  replicas to tolerate  $f$  faults. All the modules undergo sequential execution when an ESVP based application runs on a single reliable faster machine.

The time redundancy of this ESVP scheme is  $((3 * T_{vi} + T_{vi}) + 3 * T_m + (3 * T_{vo} + T_{vo}))$ , where  $T_{vi}$  is the execution time of an input voting module and  $T_{vo}$  is the execution time of an output voting module.  $T_m$  is the execution time taken by one processing module for a typical application. Voting module is implemented by XOR ing. For comparison of three input or output data, we need three XOR operations only. A compiler carries out this operation most efficiently. In other words, time redundancy here is  $(8 * T_v + 3 * T_m)$ , where  $T_v = T_{vi} = T_{vo}$  ( $\cong 3$  XOR operation time). For a medium or large size application, voting time is negligibly smaller than the execution time of an application. In other words, when  $8 * T_v \ll 3 * T_m$ , the time redundancy is about  $3 * T_m$  (that is, time overhead is three times the execution time of an application without any fault tolerance fix). Thus, time redundancy here, is on an average of the order of 3 only (similar to a sequential TMR based system using one machine). Again, the size of a voting-code is also negligible with respect to a medium or large size application. Thus, memory space redundancy is also of the order of 3 only (similar to a TMR scheme). In other words, time and memory space redundancy of the ESVP scheme is acceptable here because its overhead is similar to that of an well accepted, conventional and a typical TMR based scheme only. Software development cost here is almost one-third of an NVP scheme. Hardware cost is also one-third of an NVP scheme with three different versions. In order to get faster performance (similar to an NVP scheme with three design variants), system designers might use this ESVP scheme on a multiprocessor environment of course on spending a higher cost on hardware and on bearing more time overheads on synchronization tasks.

### 3.2 An ESVP-based Application

As a benchmark, we have taken the problem of finding out whether a given number is a prime or not. The following steps demonstrate how to implement ESVP scheme on this typical application system. If a number is prime, then processing module returns "1", otherwise "0". A voter module returns a value that is in majority among its three input data. If there is no majority among all input data to a voter, then that voter returns "-1" to indicate "no majority" and in such case of application crash, we need to reload and restart the application.

Step 0. Read  $N$  /\* read a number \*/

Step 1. Set  $N_1, N_2, N_3 = N$  /\* three copies of  $N$  on input data variables \*/

Step 2.  $N_{i1} = \mathbf{V}_{i1}^1(N_1, N_2, N_3)$  /\* 1<sup>st</sup> stage 1<sup>st</sup> input voter returns the major input data \*/

Step 3.  $N_{i2} = \mathbf{V}_{i2}^1(N_1, N_2, N_3)$  /\* 1<sup>st</sup> stage 2<sup>nd</sup> input voter returns the major input data \*/

Step 4.  $N_{i3} = \mathbf{V}_{i3}^1(N_1, N_2, N_3)$  /\* 1<sup>st</sup> stage 3<sup>rd</sup> input voter returns the major input data \*/

Step 5. If  $(N_{i1} \neq N_{i2} \neq N_{i3} \neq -1)$  Then

$N_m = \mathbf{V}_i^2(N_1, N_2, N_3)$  /\* 2<sup>nd</sup> stage input voter returns final major ( $N_m$ ) input data \*/

Else /\* No majority is found among the copies of input data to an application, input errors \*/

*Restart the application* /\* tolerate transient data errors \*/

[End of If structure]

Step 6. If  $(N_m \neq -1)$  Then

$r_1 = \mathbf{M}_1(N_m)$  /\* Variable  $r_1$  holds the computed answer from the 1<sup>st</sup> copy of the processing module \*/

$r_2 = \mathbf{M}_2(N_m)$  /\* Variable  $r_2$  holds the computed answer from the 2<sup>nd</sup> copy of the

processing module \*/

$r_3 = M_3(Nm)$  /\* Variable  $r_3$  holds the computed answer from the 3<sup>rd</sup> copy of the

processing module \*/

Else /\* no majority among the computed answers from all the identical processing modules;

all the Processing Modules are erroneous \*/

*Restart the application*

[End of If structure]

Step 7.  $R_1 = V_{o1}^3(r_1, r_2, r_3)$  /\* 1<sup>st</sup> output voter at the 3<sup>rd</sup> stage returns the major answer (kept in

$R_1$  variable \*/

Step 8.  $R_2 = V_{o2}^3(r_1, r_2, r_3)$  /\* 2<sup>nd</sup> output voter at the 3<sup>rd</sup> stage returns the major answer (kept

in  $R_2$  variable \*/

Step 9.  $R_3 = V_{o3}^3(r_1, r_2, r_3)$  /\* 3<sup>rd</sup> output voter at the 3<sup>rd</sup> stage returns the major answer (kept

in  $R_3$  variable \*/

Step 10. If ( $R_1 \neq R_2 \neq R_3 \neq -1$ ) Then

$R = V_o^4(R_1, R_2, R_3)$  /\*The output voter at the 4<sup>th</sup> stage returns the major

answer ( $R$ ) \*/

If ( $R \neq -1$ ) Then

*Display the final answer "R"*

Else

*Restart the application* /\* to tolerate multiple erroneous output data \*/

[End of If structure]

Else

*Restart the application*

[End of If structure]

{End of the ESVP application}

### 3.3 Comparison of Overheads

The major drawback of error detection and fault tolerance by software means come from the increase in execution time and the memory area overhead. On studying over a simple program (as a benchmark) of finding out whether a given number is a prime or not., the overhead factors are listed below in Table 1. It is observed that the enhanced single-version programming scheme leads to an acceptable performance in comparison to other conventional schemes. Cost of an application based on ESVP is only 1/3 rd of a TMR system.

<b>Program Approach</b>	<b>Time Overhead</b>	<b>Memory Overhead</b>
Triple Modular Redundancy (TMR) using a Uniprocessor system	> 3	> 3.2
Hamming	>10.1	< 3
Enhanced Single-Version Programming using a Uniprocessor system	3.1 and < 3.5	< 3.4
ESVP using a multiprocessor system	>1	< 3.3
Triple Modular Redundancy or an NVP using multiple machines.	> 1	< 3.25

**Table 1. Overhead Comparisons**

### 4. Conclusion

This paper has described the proposed Enhanced Single-Version Programming scheme of fault tolerance in a very lucid style. ESVP scheme has been compared with various conventional schemes like TMR, NMR and NVP schemes. The proposed ESVP approach demands a thorough study of an application system in order to minimize software design bugs. ESVP is intended to tolerate operational faults during the life cycle of an application. It is assumed that software code

is correct. System design engineers having sound knowledge on an application system will find it a very useful and economical tool for designing various industrial application systems with built in higher fault tolerance through application semantic protective codes. This is also useful for designing in dependable computing, software safety, and system reliability by adopting such single-version (with minimum modular redundancy) of both the software and hardware. It is a very low cost and an effective solution for designing in various software fault tolerant scientific, industrial and commodity application systems as well.

### ***References***

- [1] M.R. Lyu, Ed. "Handbook of Software Reliability Engineering," IEEE Computer Society Press and McGraw-Hill, 1996, New York.
- [2] Goutam Kumar Saha, "Transient Software Fault Tolerance Through Recovery", ACM Ubiquity, Vol. 4(29), Association for Computing Machines (ACM) Press, 2003, USA.
- [3] Goutam Kumar Saha, "Beyond The Conventional Techniques of Software Fault Tolerance," ACM Ubiquity, Vol.4(47), 2004, ACM Press, USA.
- [4] B. Littlewood, P. Popov, L. Strigini, and N. Shryane, "Modelling the Effects of Combining Diverse Software Fault Removal Techniques," IEEE Trans. Software Engineering, SE-26, 12, 2000, 1157-1167.
- [5] B. Littlewood and D.R. Muller, "Conceptual Modelling of Coincident Failures in Multi-Version Software," IEEE Trans. Software Engineering, SE-15, 12, 1989, 1596-1614.

### ***Author's Biography:***

Goutam Kumar Saha ([gsaha@acm.org](mailto:gsaha@acm.org), [sahagk@gmail.com](mailto:sahagk@gmail.com)) has been working as a Computer Scientist for last eighteen years. He has also taught the Post Graduate Computer courses. He has worked in various renowned research organizations namely at *LRDE*, Defence Research & Development Organization (*DRDO*), Bangalore, at Electronics Research & Development Centre of India (*ER&DCI*) Calcutta. He is presently working as a Scientist-F in the Centre for Development of Advanced Computing (*CDAC*), Kolkata. He is a senior member in IEEE

(USA), Computer Society of India (CSI). He is a Fellow Member in MSPI (New Delhi), IMS (Goa) and in IETE (New Delhi). He is also a member of the ITS Working Group of the WWW Consortium (W3C). He received various grants & awards from foreign and national reputed institutions. He has authored around one hundred research papers on fault tolerant & dependable & secure computing and on Natural Language Engineering. He is a referee for CSI Journal, AMSE Journal, IJCPOL, JZUS and IEEE Potentials. He is an associate editor of the ACM Ubiquity, ACM Press (USA).

Source: Ubiquity -- Volume 7, Issue 31  
(August 15, 2006 - August 22, 2006)

<<http://www.acm.org/ubiquity>>