"Are N average software versions better than 1 good version ? "

by

Les Hatton[1]

Programming Research Ltd., Glenbrook House, 1-11 Molesey Road,

Hersham, Surrey, U.K., KT12 4RH

Version #1.4, 14th. Jan, 1997

[1] Now at Oakwood Computing, UK. lesh@oakcomp.demon.co.uk;
http://www.oakcomp.demon.co.uk/

Abstract

The concept of using N parallel versions accompanied with some kind of voting system is a long-established one in high-integrity engineering. The independence of the channels produces a system which is far more reliable than one channel could be. In recent years, the concept has also been applied to systems containing software using diversity of design. In such systems, it is attractive to assume that software systems, like their hardware counterparts also fail independently. However, in a widely-quoted experiment, [1], [2] showed that this assumption is incorrect, and that programmers tended to commit certain classes of mistake dependently. It can then be argued that the benefit of having N independently developed software channels loses at least some of its appeal as the dependence of certain classes of error means that N channels are less immune to failure than N equivalent independent channels, as occurs typically in hardware implementations.

The above result then brings into question whether it is more cost-effective to develop one exceptionally good software channel or N less good channels. This is particularly relevant to aerospace systems with the Boeing 777 adopting the former policy and the Airbus adopting at least partially, the latter.

This paper attempts to resolve this issue by studying existing systems data and concludes that the evidence so far suggests that the N-version approach is significantly superior and its relative superiority is likely to increase in future.

Overview

Diversity is an attractive way of avoiding certain kinds of system failure, and has a long successful history in hardware systems. In essence, it works as follows. The components of a hardware system are replicated N times and are used in parallel with the same inputs. On the output side, the N outputs are compared and a single output forwarded based on the agreement between the N channels, which is monitored by a voting system. For example, if there were 3 channels, and 2 agreed but 1 didn't, a common strategy based on majority voting, would give a single output based on the two channels in agreement, whilst the third would be ignored, although it might of course generate a diagnostic. This is illustrated schematically in Figure 1.

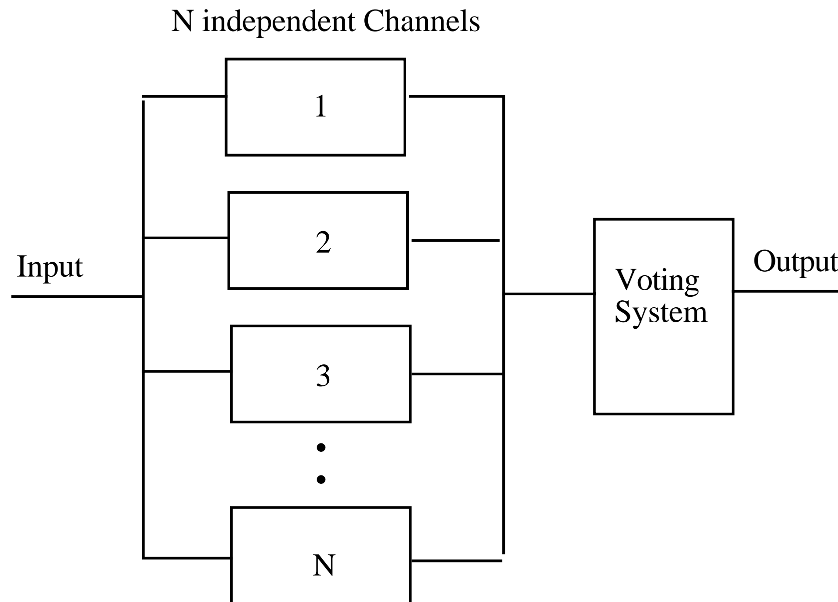

Figure 1. A schematic of an N-version diverse system. The N channels operate independently on the single input. The Voting System decides on a single output based on the N outputs of the independently operating channels.

The rationale behind this design strategy is that if the channels are independent, and the probability of failure of any one is p, and that it is further assumed that failures always lead to disagreement (with either a working channel or each other), and that the voting system is perfect, then the probability that at least two agree out of N is

$$P(M \geq 2) = 1 - P(M = 1) - P(M = 0)$$

where *M* is the number of channels which agree. The probability of failure of the system subject to the above assumptions is therefore:

$$1 - P(M \geq 2) = P(M = 0) + P(M = 1)$$

giving

$$1 - P(M \geq 2) = p^{N} + N(1 - p)p^{N-1}$$

So the use of N-version diversity leads to a factor of improvement over a single channel of:

$$\frac{p}{p^{N} + N(1-p)p^{N-1}} = \frac{1}{p^{N-1} + N(1-p)p^{N-2}} \qquad (1)$$

Table 1 gives an idea of the improvement for different values of N when p=1/2 and p=1/4.

Diversity N	p=0.1	p=0.01	p=0.001
3	3.57	33.56	333.56
4	27.02	2518.89	250187.62

Table 1. This shows the reliability improvement for diversity factors of N, for a single channel reliability of 0.1, 0.01 and 0.001.

The interesting point about this table is that it illustrates that when a single system is already relatively reliable (p=0.001), diversity improves things more dramatically as N increases than when the system is not so reliable to start with (p=0.1). It is important to re-iterate however the assumptions which led to this. First of all, that channels fail independently, and second, when they do, they neither agree with each other *nor* a working channel. We will not attempt to analyse the effects of a breakdown of these assumptions any further here.

The above technology has been used with great success in various areas of hardware development, both computerised and not. An example of a computerised implementation is provided by the design of the pressure transducer cable towed behind a seismic survey vessel, wherein 3 channel diversity is employed to communicate signals between adjacent sections of the cable, using a majority voting system. Another example this time from a critical system is the Sizewell B nuclear reactor primary protection system in the UK, however, in this case the same software is used in each channel, so the diversity is hardware-based only. Given the dramatic improvement in the reliability of computer hardware in recent years to the point where it is far more reliable than the software which runs on it, this approach seems a little unbalanced even given the relatively low cost of hardware. For example, this paper is being written on a Pentium PC using Word 7 under Window's 95. We have found that a typical spread of software use on this system exhibits a defect every 42 minutes on average, with a serious failure around every 5 hours. In contrast, the hardware of such systems appears to be several orders of magnitude more reliable.

A final and in this case, a non-computerised example, is provided by control cables in older technology aircraft. For example, on the McDonnell-Douglas DC10, there were three such cables passing through the aircraft to control the rear control surfaces. In theory, this would provide excellent protection against the failure of any one. A tragic example of the break-down of the assumption of independence

© Les Hatton, 1996-7.

was provided some years ago when the baggage door of a DC10 fell off when taking off at a French airport. This caused a de-pressurisation of the baggage compartment which then caused the floor to collapse. However, all three cables ran through the floor at the time, and all were severed, causing the aircraft to crash out of control, with considerable loss of life. Engineers call this *common-mode failure*. The engineering solution to prevent a future occurrence of this was to re-route one of the three through the ceiling of the aircraft. For further examples, refer to [3].

This is an excellent introduction to the concept of software diversity, because one of the principle objections as will be seen, is that common-mode failure is far more common in software than in hardware implementations and an estimate of just how much more common is crucial to the resolution of the primary question asked in the title of this paper.

To implement the concept of diversity in software systems, a requirements specification is prepared and then given to a number of different software development groups. The software groups work in strict isolation and preferably in different languages to deliver a software solution to the original requirements specification. In practice, each software solution is run in an independent computer, together forming a channel as shown in Figure 1.

The nature of diverse software channels is of course fundamentally different from hardware channels. Hardware channels are typically subject to the laws of physics, whereas of course software channels are creations of the mind. Hardware channels are subject to wear and tear, whereas software channels are generally not unless such factors as memory bit drop-out are considered. As a result, nearly all faults in software channels have always been present. There is another difference, and a very important one, that of cost. When constructing independent hardware channels, there are considerable economies of scale, and N such channels, although more expensive than one, are not prohibitively so. In contrast, N software channels on average cost N times one software channel, (equally convincing arguments were given by two different reviewers for circumstances leading to both greater and lesser than N times the cost so we have taken a notional average). Given this financial penalty, it is a matter of some importance to investigate the degree of common-mode failure present in diverse software channels to see if the increased cost is worth it. Fortunately, this problem has attracted considerable attention as a consequence, and some of the results will now be presented.

The dependence of N software channels

This was first studied in detail by [1] in a widely-cited paper. It is worthwhile discussing this experiment in some detail. The experiment was carried out simultaneously amongst 2 different University sites using 27 separate programmers, with experience varying from very little to very considerable. Only the programming language was common, in this case Pascal, so the results may well not apply to diverse channels involving multiple programming languages. The application programmed was a realistic defence-related one, relating to the launch of a missile, based on input radar information. The overall spread of sizes of the 27 different versions was between 327 and 1004 lines. Each version was required to undergo an acceptance test before being subjected to the 1,000,000 trials simulating the chosen real-time environment. This acceptance test required the correct answers be produced on 200 randomly generated test cases from 15 input datasets. These 200 tests were different for each version. This represents quite an exhaustive acceptance test, even for this kind of environment, and perhaps not surprisingly, the resulting 27 versions were individually found to be very reliable when subjected to the 1,000,000 trials, with individual probabilities of success on any particular input of between 0.99 and 1.

This paper specifically pointed out the non-independence of failure occurrence between the various versions but did not go on to attempt to quantify just how much degradation from the ideal would occur in a typical N-version system. This was the subject of a later paper, [2] , which revisited the data in the first paper and numerically seeded N-version systems with failure properties representative of the original sample of 27 programs. In these seeded trials, a 3-version system with majority voting was shown to offer only around 20-50 times more reliability than a single version typical of the sample. A later paper [4] studied this question further.

A detailed analysis

To contrast the empirical approach of [2], the fault details published by [2] and studied further by (Brilliant, Knight et al., 1990) will be analysed in a somewhat different way. As has been seen, this particular experiment required a thorough unit testing program before the faults recorded in a 1,000,000 trial simulation of real run-time conditions was carried out on each of the 27 versions. During this 1,000,000 trial run [1] detailed 45 faults spread amongst the 27 versions. To give some idea of the distribution, 6 of the versions exhibited zero faults during this trial, whilst one version exhibited 7, the most recorded in any of the versions. As has already been quoted, the versions had between 327 and 1004 lines. If an average of

around 600 lines is assumed, this means that 45 faults were found somewhat non-uniformly in roughly 27 * 600 = 16,200 lines of code, giving a notional fault density of approximately 3 per KLOC which compares very favourably with typical fault densities for high reliability systems quoted in [5] for example, and corresponds well with our observations that the individual versions were very reliable indeed.

First of all analysing the 45 faults, 36 seem almost certainly requirements associated and very likely to be language independent. The remaining 9 were associated with precision, which varies between language, and may not have occurred in another programming language. Hence, in this experiment, multiple programming languages could be expected to have made little difference affecting as it does, only around 25% of the faults.

To determine the approximate penalty of the loss of independence, a 3 channel system will be considered first, built using components with failure properties typical of this system. In all, the 27 versions exhibited a total of 18,962 failures in 1,000,000 trials, varying between 0 and 9656. Given that this one version contributed just over half of all the failures, it will be omitted in calculating the average to err in favour of the non-independence argument, giving a notional average of 358 failures per version on the 1,000,000 trials, or an average probability of failure on any trial of .0004.

In this experiment, 2 or 3 versions failed simultaneously a total of 894 times, (551 and 343 times respectively) spread across the 27 versions. From this, the number of likely 2 or 3 version failures for just 3 components must be inferred. This can be done by first noting that 6 versions were fault-free during the trials. Therefore the number of combinations of 2 versions amongst the remaining 21 fault-exhibiting versions is (21x20)/(2x1) = 210. It will now be assumed that the 551 2-version simultaneous faults were distributed evenly amongst these 210 combinations, giving (551/210) per combination, although it should be noted that in the original work of [1], they were not. This assumption does not seem to affect the overall result significantly. Now in a 3 channel system there are (3x2)/2 = 3 such combinations, so it can reasonably be inferred that there will be on average 3x(551/210) = 7.87, 2-version faults encountered during 1,000,000 trials. A similar reasoning suggests that there will be 343 faults distributed amongst ((21x20x19)/(3x2x1)) = 1330, 3-version combinations. A 3-channel system has 1 such combination leading to an expected 343/1330 = 0.26 3-version faults during the 1,000,000 trials.

Now to summarise, a system implemented with just one version would fail *on average* 358 times during the 1,000,000 trials, whereas a 3 channel version governed by majority polling would only fail whenever there was a 2 or 3 version coincident failure, which according to the simplified and highly averaged analysis above, would occur on average 7.87+0.26 ~ 8 times. In other words, the improvement ratio of a 3 channel system over a 1 channel system in the presence of non-independent channel failures of the frequency encountered in this experiment would be:

358/8 ~ 45

This agrees closely with the empirical results produced by [2]. Now from equation (1), for pure independence and N=3 and p=.0004, a 3 channel system would be 833.6 times more reliable than a single channel. So the loss of independence degrades the 3 channel to 1 channel improvement ratio from 833.6 to 1 down to around 45 to 1, although the value of 45 should be considered as somewhat pessimistic in view of the omission of the worst case to help give a stable estimate of the average. Of course, it can be argued that a single channel could have been chosen much worse (or better) than this average value, but this would require hindsight and only statistical arguments can be permitted here. The real question is whether a pessimistic but still substantial improvement factor of 45 for a 3 channel system over a single channel system is worth the expense of developing three individual versions. This is taken up in the next section.

How well can we make one channel ?

It is a continuing delusion amongst software development staff that we can make really good software if only we had the time. This is supported to a certain extent by the experiences of the NASA Shuttle development team, who by dint of spending resources denied to all but a privileged few, have achieved extraordinarily high reliability, [6]. Most organisations do not attach such importance to software reliability however even though there are well documented cases of massive loss, [3]. Even so, many techniques have come (and gone) which supposedly promote the goal of improved reliability. However, these all too often arise in the complete absence of measurement and so it is far from obvious what helps, by how much and why. In this regard, software development is much more akin to the fashion industry than an engineering industry. Such techniques as formal methods, various exotic design methodologies including those based on the now ubiquitous and entirely unsupported object-oriented paradigm, systematic testing and so on, have

© Les Hatton, 1996-7.

all been blindly promoted as "the way forward". Unfortunately, software systems have been unusually resistant to such advances and the defect density (discussed below) of software has remained about the same over the last 15 years or so, [7].

One of the standard ways of measuring the quality of software system is by use of the defect density, or the number of defects found per 1000 lines of source code (KLOC) *over some period of time long enough to represent reasonable use of the system*. Although this method has numerous deficiencies, it provides a reasonable rough guide. By this yardstick, 3-6 defects per KLOC represents high-quality software, e.g. [5], [7] and it is rare indeed to find systems below 1 defect per KLOC. Even by the use of formal specification techniques, 100% statement coverage and concurrent test plans in very high-integrity systems such as discussed by [8] when studying an air-traffic control system, the resulting system still contained around 0.7 defects per KLOC. So we can conclude that if we do everything we can to write good software, we might produce a 0.5 defects per KLOC system if we were lucky, which is only around 5-10 times better than the average for a good system. Such a system is of course of comparatively high reliability and the single air-traffic control system discussed by [8] failed just 6 times in 2 years, giving a notional MTBF of around once per 4 months of use, assuming 24 hour use of a single copy, its normal use. This of course compares exceptionally well with the PC reliability figures of one defect per 42 minutes quoted earlier, but the question nevertheless arises, "is even this good enough ?". In a fly-by-wire system, there are of course multiple copies flying concurrently, so system use time is clocked up very quickly indeed. It can be convincingly argued therefore that a MTBF of around a few months *per system* is simply inadequate and that very high reliability will have to be achieved through other means [9].

Comparing the two

Combining the figures above, it seems that using the best techniques we know how to use, we can perhaps improve the defect density of a single system by around a factor of 5-10 *on average* over that currently achieved for good systems. However, a 3 version system built using components typical of the average for good systems could produce around a factor of 45 improvement, again *on average*. So, is an additional factor of 5-9 worth a development cost of perhaps a factor of 2 extra ? This question can only be answered if the cost of failure is known. In consumer electronics, extraordinarily high recall costs strongly favour any technique which improves matters such as the N-version techniques described here. This is also the case for application areas of high integrity and heavy use, such as those in fly-by-

© Les Hatton, 1996-7.

wire aircraft, in which software development costs still represent a relatively small component of the overall cost. It should however be re-iterated that defect density and failure rate are not trivially related. It should also be remembered that it may be necessary to make only a small part of a critical system diverse, as highlighted by fault tree analysis or some similar technology.

The effects of progress

It has already been noted in equation (1), that improvement in the *average* quality of each component has a non-linear effect on the quality of an N-version system. If we make the not unreasonable assumption that the state of the art is always 10 times better than the average, then the improvement ratio of a majority voted 3 independent version system with all the previous assumptions in place over the current state of the art is:

$$\frac{\left(\frac{p}{10}\right)}{p^{N}+N(1-p)p^{N-1}}=\frac{0.1}{p^{N-1}+N(1-p)p^{N-2}} \qquad (2)$$

Table 2 gives an idea of the improvement for different values for a 3 independent version system when p=0.01 and p=0.001.

Diversity N	p=0.01	p=0.001
3	3.3	33

Table 2. This shows the reliability improvement factor for a 3 independent version system compared with a single state of the art system, for increasing individual component reliability assuming that the state of the art is always 10 times better than the average.

We can generalise the above result for a very high reliability population by defining the state of the art reliability at any point in time as q and the average reliability as p. The improvement factor of a 3 independent version system over a single 'best' system is then:

$$\frac{q}{p^{3}+3(1-p)p^{2}}$$

which for a very reliable population, i.e. $p \ll 1$, gives

$$\frac{q}{3p^2}$$

In other words, in an increasingly reliable population from which to select components for a 3 version system, the state of the art would have to improve as a function of the square of the population average in order to keep pace with the improvement of a 3 independent version majority voting system. So gradual population improvement appears to favour N version systems more and more over a single state of the art system.

Now these results strictly apply only for independent versions, although it can be expected that a qualitatively similar situation holds for the dependent case also. This is however outside the scope of this study and needs to be investigated.

Conclusions

The balance of data tends to suggest that N version techniques are preferable in software development when the cost of failure is high due to our inability to make one really good version whatever techniques we currently use. Even though the advantage is much less than that for N independent channels, the difference is still substantial and in this case gave a factor of 5-9 improvement on average for a majority voted 3 version system compared with a single version typical of the current state of the art.

Furthermore, a gradually improving population appears to favour N version techniques even more, but even with the current state of the art, N version techniques appear to provide an extra level of reliability we are unable to gain in any other way. This is strongly supported by the study of [10] who showed that such techniques enable major defects to be detected in complex systems which had escaped all other techniques over a period of many years. To reduce the cost of producing independently designed software channels, a policy of implementing such independence only in critical components is an attractive design possibility for critical systems.

Acknowledgements

We would like to acknowledge Tom Anderson for his encouragement and his very considerable expertise in this area both of which he made liberally available. The anonymous reviewers also contributed considerably to clarifying the arguments.

References

1. Knight, J.C. and N.G. Leveson, *An experimental evaluation of the assumption of independence in multi-version programming*. IEEE Transactions on Software Engineering, 1986. **12**(1): p. 96-109.

2. Knight, J.C. and N.G. Leveson. *An empirical study of failure probabilities in multi-version software*. in *FTCS-16*. 1986. Vienna.

3. Neumann, P.G., *Computer-Related Risks*. 1995, New York: Addison-Wesley. 367.

4. Brilliant, S.S., J.C. Knight, and N.G. Leveson, *Analysis of faults in an N-version software experiment.* IEEE Transactions on Software Engineering, 1990. **16**(2): p. 238-247.

5. Musa, J., Iannino., and Okumuto., *Software Reliability: Measurement, Prediction, Application.* 1987: McGraw-Hill.

6. Keller, T.W. *Achieving Error-Free Man-Rated Software*. in *2nd International Software Testing, Analysis and Review Conference*. 1993. Monterey, CA,.

7. Hatton, L., *Re-examining the fault density - component size connection.* IEEE Software, 1997. **14(2)**(March/April 1997): p. p. 89-97.

8. Pfleeger Lawrence, S. and L. Hatton, *Investigating the influence of formal methods.* IEEE Computer, Feb. 1997, 1997. **30**(2): p. pp 33-43.

9. Littlewood, B., *The Need for Evidence from Disparate Sources to Evaluate Software Safety*, in *Directions in Safety-critical Systems*, F. Redmill and T. Anderson, Editors. 1993, Springer-Verlag: London. p. 285.

10. Hatton, L. and A. Roberts, *How accurate is scientific software ?* IEEE Transactions on Software Engineering, 1994. **20**(10 (October 1994)): p. p. 785-797.

© Les Hatton, 1996-7.