# DESIGNING GENERAL-PURPOSE FAULT-TOLERANT DISTRIBUTED SYSTEMS - A LAYERED APPROACH*

Amiya R. Nayak[†] Wen-Ben Jone[‡] and Sunil R. Das[§]

*Abstract*

General-purpose distributed systems comprised of computing nodes with different characteristics and connected by high-speed communication networks are very popular these days. The development of a dependable distributed system, however, necessitates the use of various techniques including fault tolerance to avert occurrences of failures or system malfunction. The *ad hoc* techniques of adding redundancy to improve reliability are not always suitable in these circumstances because of excessive design cost. Redundancies have to be allocated at various hardware and software levels in order to optimize their utilization in the system. This paper considers the design of general-purpose fault-tolerant distributed systems based on a layered approach. The benefits of the layered approach in the process of allocation of redundancy and fault tolerance at various system levels are presented and analyzed in the paper.

## 1 Introduction

The development of fault-tolerant distributed systems requires the combined utilization of a wide range of techniques, including that of fault tolerance intended to cope with the effects of faults and avert the occurrence of failures or at least to warn a user that errors have been introduced into the system.

A distributed system consists of a set of autonomous computing nodes connected by some kind of communication network as shown in Figure 1. Each node may have different characteristics and is capable of providing different software services to other nodes in the system. A distributed system is intended to support many different applications and to execute concurrently many unrelated requests that could compete for both hardware and software resources.

The task of designing and understanding fault-tolerant distributed systems has been addressed in [3], in which the author, conceptually, divides a computer system into several levels of abstraction such as the hardware (or processor) level, the operating system (or system software) level, and the application level as shown in Figure 2. The application level is the highest, and the hardware level is the lowest level of abstraction in the system hierarchy. In such hierarchy, a higher level of abstraction always receives service from the lower levels. In this paper, we consider these three levels of abstraction in our discussion.

In the design of fault-tolerant computing systems, measures for detecting and tolerating faults can be applied at all levels. Applying fault tolerance in the lower levels is useful in that it frees the designers of the higher levels from consideration of faults in the lower levels, shares the cost of fault-tolerant design among all the applications that use the same lower-level service, and allows the designer to employ relatively cheap techniques to cover known frequent faults.

A key issue in designing multi-layered fault-tolerant systems is how to balance the amount of redundancy at the various levels of a system, in order to obtain the best possible overall cost/performance/dependability results. The "end-to-end" arguments in the design of layered systems given in [8] indicate that too much of

[1]This research was supported in part by the Natural Sciences and Engineering Research Council of Canada under Grant A 4750.

[2]School of Computer Science, Carleton University, Ottawa, Ontario, Canada K1S 5B6.

[3]Dept. of Computer Science & Information Engineering, National Chung Cheng University, Chiayi 62107, Taiwan, R.O.C.

[4]Dept. of Electrical Engineering, Faculty of Engineering, University of Ottawa, Ottawa, Ontario, Canada K1N 6N5.

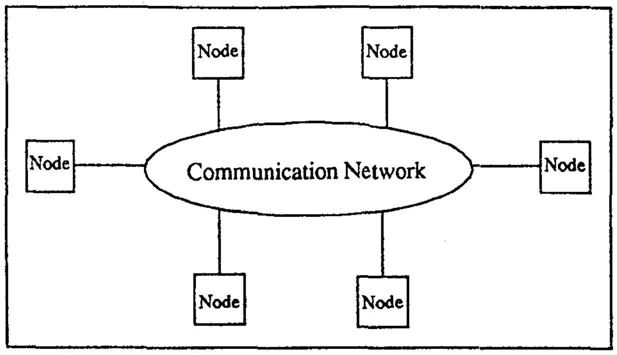

Figure 1: The schematic view of a distributed system

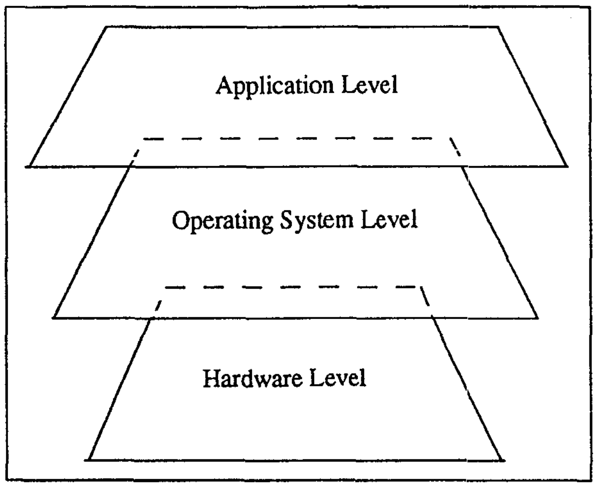

Figure 2: Three-Level System Hierarchy

redundancy, at the lower levels of abstraction of a system might be wasteful from an overall cost/effectiveness point of view, and it may be cost-effective to save on lower level redundancy mechanisms and put more effort into fault tolerance in the higher levels to avoid duplication of effort. On the other hand, a small investment at a lower level of abstraction can often contribute to substantial cost saving and speed improvements at higher levels of abstraction and can result in lower overall cost. Therefore, a balance in the amount of redundancy at various levels of abstraction of a system is necessary.

## 2 Fault Tolerance Techniques

The key ingredient in all fault tolerance techniques is redundancy. Redundancy is simply the addition of *information*, *resources*, or *time* beyond what is needed for normal system operation. The redundancy may take several forms, including information redundancy, hardware redundancy, software redundancy, and time redundancy.

*Information redundancy* is the addition of information beyond what is required to implement a function. A good example of information redundancy is an error-detecting code. Perhaps the simplest form of error detection coding is the single-bit parity check. Other forms of error detection coding are: checksums and arithmetic codes [9]. Checksums are most applicable when blocks of data are to be transferred from one point to another. The primary advantage of the arithmetic codes is that they allow hardware that performs arithmetic operations to be easily checked. Once the location of the error is known, the data bit can be corrected by a single-parity code. Error-correcting codes not only detect errors but also correct them. Possibly the most common extension of parity is the Hamming error- correcting code. Hamming codes are designed to detect double errors and correct single errors. Other forms of single and multiple error-correcting codes also exist and can be found in [6].

*Hardware redundancy* is the physical replication of hardware for the purpose of detecting and tolerating faults. A common form of this redundancy is the triple modular redundancy, or TMR [9]. The purpose of TMR is to mask single faults by triplicating hardware and voting on the results. The general form of TMR is the N-Modular redundancy (NMR). Other form of hardware redundancy involves incorporating fault detection and fault recovery into the system at the expense of eliminating fault-masking capability. Example of this is the standby replacement. In this configuration, one unit is operational while one or more units are standbys. Various error detection schemes are used to determine when the on-line unit has failed; if a failure is detected, the on-line unit is removed from operation and replaced with a standby.

*Software redundancy* is simply the addition of extra software to provide some fault tolerance features. The type of redundancy may range from complete duplication of software to the addition of small programs to perform validity checks. Probably the most common form of software redundancy is the validity, or reasonableness check. A second type of software redundancy is the periodic self- test. A third example of software redundancy is the use of multiple copies of programs. This version of software redundancy is typically referred to as "N-version programming" [1]. In this approach, multiple versions of a software run simultaneously on multiple processors or sequentially on a single processor; the results are compared to provide a means of fault detection. The N-version, independently written programs provide protection against both hardware and software faults. The hardware equivalent of N-version programming is the N-modular redundancy.

*Time redundancy* uses additional time to provide fault detection and, sometimes, fault tolerance. It can be employed to distinguish between permanent and transient failures. The processor performs the computations one or more times after detecting first error; if the error condition clears, the processor can assume that fault was transient. In another form of time redundancy, the same processor performs the same computations multiple times using different coding schemes in each case with the assumption that the fault may manifest itself in different ways depending upon the particular code being used. Unfortunately, the time redundancy technique cannot be used to detect permanent failures.

All the fault tolerance techniques mentioned above can be separated into two major categories: static redundancy and dynamic redundancy. *Static redundancy* techniques instantly correct errors and do not require higher system-level intervention to do so. The use of TMR or error correcting codes are some examples of the static redundancy technique. *Dynamic redundancy*, on the other hand, requires interruption of system availability while the fault is isolated. Once the fault is isolated, the redundant resource is switched on, and the system is recovered. For a real-time system, if the error occurs in the middle of an operation, some program rollback is necessary to discard the bad data and to recover as much good data as possible. The primary advantage of dynamic redundancy techniques is that they can be implemented with fewer resources.

Hybrid redundancy systems combine static and dynamic redundancy approaches. Static redundancy provides error masking and detecting capability. Dynamic redundancy provides a set of spares and a switching network for reconfiguration.

Fault-tolerant systems have taken one of two architectural directions: *loosely-coupled* systems have relied primarily on software to ensure reliable operation, while *tightly-coupled* systems have provided a hardware solution. A great deal of work in the area of fault-tolerant computing has been done in recent years, much of it in the support of the U.S. space program and special applications such as electronic telephone switching systems [9]. The use of various fault tolerance techniques in commercial computer systems are well documented in [5, 7, 9].

## 3 Layered Approach – Philosophy

In the past, *ad hoc* methods were employed to add redundancy to existing designs to improve their reliability. There are two major drawbacks to such an approach: the massive redundancy needed is enormously expensive to design, and it has a negative impact on system performance. This can be overcome using the following layered approach. First, the fault tolerance must be integrated into the development of a system from its inception to allow the system designer to exploit inherent characteristics such as parallelism and concurrency that will automatically minimize redundancy and thus minimize any negative impact on system performance. Second, redundancy and fault tolerance should be allocated at different layers of abstraction of a system. Allocating redundancy at various levels of a system can optimize hardware and software usage.

The advantage of the layered approach is the efficient utilization of the redundant resources, which decreases both the hardware overhead and the recovery time. When a fault is diagnosed at the lowest level of system hierarchy and the recovery is not possible at that level, then steps can be taken to mask the fault at the next higher level in the hierarchy. This procedure can be repeated until every failure at any level can be handled at that level or at a level above in the hierarchy.

Some limited application of this concept has been reported in the literature [11] where benefits related to specific system design have been investigated through savings in hardware cost and recovery time.

## 4 Redundancy Management

Based on the layered approach, two major approaches to the incorporation of fault tolerance into systems may be followed which represents two extremes of possible choices. In the first approach, called the *structured approach*, the most profitable fault-tolerant techniques are applied to different levels of abstraction. This has the merit of controlling extra complexity while introducing fault tolerance into the system. In fact, since the set of faults needed to be considered is isolated to within a single level plus a set of well-defined failures of the underlying levels, the provision of fault tolerance in that level is usually very simple and easy to control. However, the approach may cause the loss of efficiency and performance. First, runtime costs due to fault tolerance in each level can be quite significant resulting in a very high runtime overhead, even in the absence of failures. Second, the fault-tolerant techniques in different levels may overlap heavily leading to poor performance.

The second approach is called the *integrated approach*. In this approach, redundancy is still spread over different levels but the redundancy management and the corrective action in the event of failures are concentrated only in some higher level. Failures in lower level are propagated upwards and are masked by some previously selected higher level. This approach is obviously complementary to the structured approach. The overlap of fault-tolerant mechanisms and techniques could be controlled and minimized so as to improve efficiency and performance. However, such a way of incorporating fault tolerance into systems can be quite complex.

### 4.1 Hardware or Processor Level

With the advances in microprocessor and VLSI technology, a reasonable level of redundancy at the processor level is justifiable on a cost basis. A wide range of fault-tolerant options are available for this level.

It is possible to implement the redundancy management mechanisms such as the TMR scheme which can mask hardware failures directly in hardware. Reconfiguration and recovery should be performed by the hardware as much as possible without any software intervention. The sparing policy can be used at the processor level, which agrees with the widespread utilization of microprocessors. A fault affecting a processor leads to replacement of that processor. In some cases more than one processor is replaced; in fact, fault-free processors may be replaced to simplify the recovery strategy.

### 4.2 Operating System or System Software Level

The system software (or operating system) in a distributed system has a key role to play in dealing with processor or communication failures. System-level fault-diagnosis mechanism, in which processors test each other, is a good choice for this level. For system-level diagnosis, a faulty processor is easily identified if at least three processors are involved in testing each other because a fault sysndrome is generated that isolates the faulty processor. Several systems attempt to mask hardware failures at the operating system level, so that application software can continue to run without interruption. For example, masking of CPU bus or disk controller failures is done by the operating system of Tandem [2].

### 4.3 Application Software Level

Application level is considered to be the highest level of abstraction of the system and must have the capability to mask any lower-level failure, both hardware and software. Literature on software fault tolerance contains several proposals for structuring application-level fault tolerance in distributed applications. They come in the following forms:

- system-supported roll-back and recovery,
- modular redundancy with masking, as in multiple version programming,
- atomic transactions, and
- fault-tolerant algorithms.

Software fault tolerance needs redundancy of software design or design diversity. The well-documented techniques for tolerating software design faults include recovery blocks (RB), N-version programming, N self-checking programming, etc. Several schemes for application level fault tolerance have been reported in [10].

## 5 CONCLUSIONS

It is recognized that redundancies have to be allocated at various hardware and software levels in order to optimize their utilization in the system. It is also recognized that a balance in the use of right amount of redundancy at various levels of a system is necessary to optimize hardware and software use. But, very little knowledge is available today that could guide a designer in choosing among possible redundancy management options at various levels of s system. Lack of analytical and experimental information on various redundancy techniques or on the failure behaviors of various system components has made such choices more difficult. However, redundancy management based on the layered philosophy has the potential of utilizing redundancy effectively and efficiently, thereby reducing both hardware overhead and recovery time.

## References

[1] A. Avizienis, "The N-Version Approach for Fault-Tolerant Systems", *IEEE Trans. on Software Engineering*, Vol. SE-11, No. 12, Dec. 1985, pp. 1491-1501.

[2] J. Bartlett, "A NonStop Kernel", *ACM 8th Symp. on Operating Systems Principles*, Dec. 1981, pp. 22-29.

[3] F. Cristian, "Understanding Fault-Tolerant Distributed Systems," *Communications of the ACM*, Vol. 34, No. 2, Feb. 1991, pp. 56-78.

[4] J. Lala, L. Alger, "Hardware and Software Fault Tolerance: A Unified Architecture Approach," *18th Int. Symp. on Fault-Tolerant Computing*, 1988, pp. 240-245.

[5] J. C. Laprie, J. Arlat, C. Beounes, and K. Kanoun, "Definition and Analysis of Hardware and Software Fault-Tolerant Architectures", *IEEE Computers*, Vol. 23, No. 7, July 1990, pp. 39-51.

[6] D. K. Pradhan, Ed., *Fault-Tolerant Computing, Theory and Techniques*, Vol. I & II, Prentice-Hall, Englewood Cliffs, NJ, 1986.

[7] D. A. Rennels, "Fault-Tolerant Computing - Concepts and Examples", *IEEE Trans. on Computers*, Vol. C-33, No. 12, Dec. 1984, pp. 1116-1129.

[8] J. Saltzer, D. Reed and D. Clark, "End-to-end Arguments in System Design," *ACM Trans. Computer Systems*, Vol. 2, No. 4, Nov. 1984.

[9] D. P. Siewiorek and R. S. Swartz, *The Theory and Practice of Reliable System Design*, Digital Press, Bedford, MA, 1982.

[10] L. Strigini and F. Di Giandomenico, "Flexible Schemes for Application-Level Fault Tolerance," *IEEE 10th Symp. on Reliable Distributed Systems*, 1991, pp. 86-95.

[11] R. Yanney, V. Nickel and D. Bender, "Fault-tolerant Computer Systems", *TRW Electronics & Defense Sector Quest*, 1987, pp. 35-49.