# Infrastructure-Based Design of Fault-Tolerant Systems:
## How to Get High-Confidence Computing for All

Algirdas Avižienis

UCLA Computer Science Department
University of California, Los Angeles, CA 90095, U.S.A.
and
Vytautas Magnus University, Kaunas, Lithuania

**Abstract**

The origin of the concept of fault tolerance and the evolution of guidelines for the systematic design of fault-tolerant systems is reviewed. The current formulation of the guidelines, called a design paradigm, is presented. The problem of using off-the-shelf subsystems in a fault-tolerant system is discussed. In conclusion, an analogy of complex fault-tolerant systems and living organisms is suggested as a means to advance the understanding and successful application of fault tolerance.

## INTRODUCTION: FAULT TOLERANCE - WHY AND HOW?

As computing and communications become irreplaceable tools of modern society, one fundamental principle becomes evident: the more benefits our computing and communication systems can contribute to the wellbeing and quality of life of the human race, the more harm they can cause when they fail to perform their functions, or perform them incorrectly. Let us consider the control of air, rail, and subway traffic, the emergency response systems of our cities, the flight controls of airliners, the safety systems of nuclear power plants, and most of all, the rapidly growing dependence of health care delivery on high-performance computing and communications. Lives and fortunes are likely to be lost when those systems fail.

At the same time, the threats to dependable operation are also growing in scope and severity. Left-over design faults ("bugs" and "glitches") cause system crashes at peak demand time, causing severe inconvenience and financial losses. Complex systems suffer stability problems due to unforeseen interactions of overlapping fault events and mismatches of defense mechanisms. "Hackers" and individuals with criminal intent invade systems and cause disruptions, misuse, and damage. Accidents lead to the severing of communications links that serve entire regions. Finally, it is unfortunately likely that "info-terrorists" will attempt to cause similar damage with malicious intent.

Fault tolerance is our best guarantee that "high-confidence" systems will not betray the trust of their builders and users by failing to serve as expected because of physical, design, or human-machine interaction faults, or because of viruses and other malicious attempts to disrupt their essential services.

There are two different ways to build a fault-tolerant system. The first is the "bottom-up" approach: to design an infrastructure of autonomously fault-tolerant subsystems(microprocessors, memories, sensors, displays, etc.), and to integrate it with global fault tolerance functions: reconfiguration, externally supported recovery, etc..

The second "top-down" approach is to build a system using pre-existing ("off-the-shelf") subsystems that may have little or no fault tolerance at all, and then to implement a global monitoring function that provides fault tolerance. This is the prevailing practice at the present.

The goal of this paper is to discuss both approaches and to present a case for the long-range merits of the "bottom-up" technique. An analogy of fault tolerance to the immune system and other defenses of the human body is proposed that should advance the understanding and acceptance of bottom-up fault tolerance by developers and users of high-confidence systems.

## ORIGIN OF THE FAULT TOLERANCE CONCEPT

The concept of fault tolerance was originally presented in 1967 [1] as follows:

*"We say that a system is fault-tolerant if its programs can be properly executed despite the occurrence of logic faults."*

The creation of the fault tolerance concept was the consequence of the convergence of three developments.

First, there were efforts by computer designers to increase the reliability of computing by the use of practical techniques. As soon as the first computers were built and turned on, it became apparent that physical defects and design mistakes could not be avoided by careful design and by the choice of good components alone. For this reason, designers of early computers employed redundant structures to mask failed components, error control codes and duplication or triplication with voting to detect or correct information errors, diagnostic techniques to locate failed components, and automatic switch-overs to replace failed subsystems [2].

Second, in parallel with the evolution of engineering techniques, the general problem of building reliable systems from unreliable components was addressed by some of the founders of computer science, most eminently J. von Neumann in 1952 [3], and E. F. Moore with C. E. Shannon in 1956 [4]. W. H. Pierce unified the theories of masking redundancy [5].

Third was the new challenge of building unmanned spacecraft for interplanetary exploration that was assigned by NASA to Caltech's Jet Propulsion Laboratory (JPL) in late 1958. Mission lengths of up to ten years and more were being considered, and on-board computing was a prerequisite for their success. Design of computers that would survive a journey of several years and then deliver their peak performance at a distant planet was an entirely unexplored discipline. Existing theoretical studies of the long-life problem indicated that

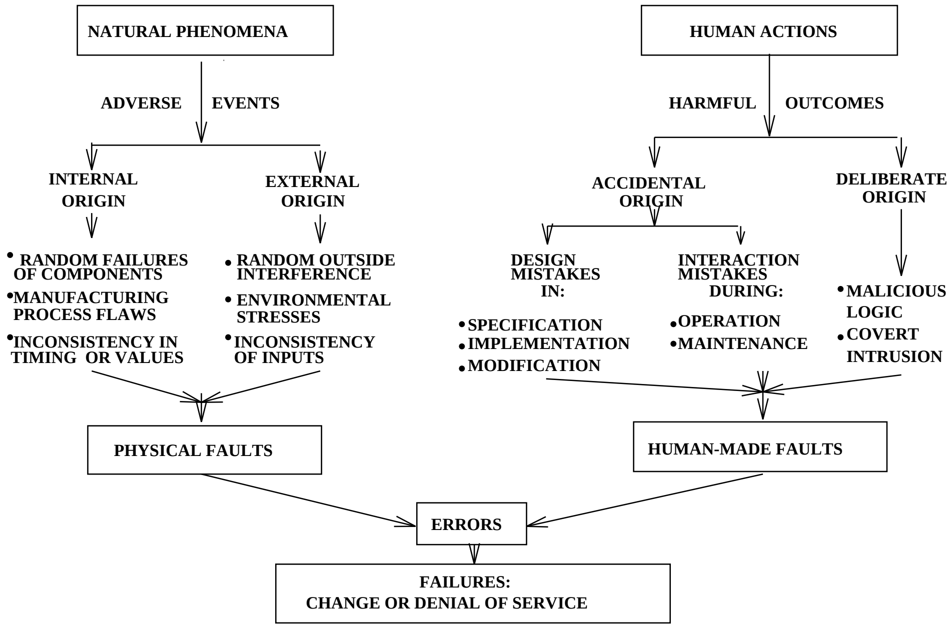


Figure 1: Why Systems Fail

large numbers of spare subsystems offered a promise of longevity, given that all spares could be successfully employed in sequence. The JPL problem was to translate the idealized "spare replacement" system model into a flightworthy implementation of a spacecraft guidance and control computer.

A proposal to design such a computer, called "A Self-Testing-And-Repairing System for Spacecraft Guidance and Control," and designated by the acronym "STAR" was presented in October, 1961 [2, pp. 181 - 191]. It was supported by JPL and NASA management, and the effort continued for over ten years, culminating with the demonstration of the laboratory model of the JPL-STAR computer [6]. Originality of the concept was affirmed by U.S. Patent No. 3,517,671 ("Self Testing and Repairing Computer"), granted on June 23, 1970 to A. Avižienis and assigned to NASA. A flight model of the JPL-STAR was designed for a 10-15 year space mission [6], but its building was halted when NASA discontinued the Grand Tour mission.

The longevity requirement led to the study of all accessible engineering solutions and theoretical investigations of reliability enhancement. The confusing variety of existing theories and techniques [2,3,4,5] motivated the unifying concept of fault tolerance [1] that merged diverse approaches into a cohesive view of all system survival attributes, and greatly facilitated the design of the JPL-STAR computer. It also was a logical step for JPL, the birthplace of the fault tolerance concept, to be the co-sponsor with IEEE-CS and to take the initiative to organize the first International Symposium on Fault-Tolerant Computing that was held in Pasadena, CA on March 1-3, 1971, and will take place for the twenty eighth time in Munich, Germany, this year.

During the following twenty five years the set of faults to be tolerated by fault-tolerant systems has grown extensively. The original concept dealt with transient and permanent logic faults of physical origin. Faults due to human mistakes in design were added when the growing complexity of software and of logic on VLSI chips made the removal of all *design faults* prior to operational use not certain. Experience also led to the addition of *interaction faults*, inadvertently introduced by humans during the operation or maintenance of a computer.

Finally, consequences of malicious actions intended to alter or to stop the service being delivered by a system were recognized as being *deliberate design faults*. This concept establishes a common ground for the unified treatment of security and fault tolerance concerns in system design. The assurance of full compatibility and integration of security and fault tolerance techniques is a major challenge for contemporary designers. An overview of the current set of faults to be tolerated is presented in Figure 1.

In retrospect, it may be said that the concept of fault tolerance has served well during the past thirty years by facilitating the appearance of successively more dependable systems for the control and support of various essential functions of contemporary society: computing and communications, transportation, nuclear power, financial transactions, health care delivery, etc.

## A DESIGN PARADIGM FOR FAULT-TOLERANT SYSTEMS

The definition of the concept of fault tolerance initiated the evolution of principles for the systematic design of fault-tolerant systems. The building of the JPL-STAR computer involved much improvisation and experimentation with design alternatives. It became apparent that the lessons learned could serve as the foundation for a more orderly approach that would guide the designer to treat fault tolerance as a systematic issue during design process.

The 1967 paper [1] initiated a 25-year sequence of publications [2, pp. 141-191] that formulated an evolving view of how to accomplish the systematic introduction of fault tolerance in hardware, software, communication, and man/machine interface subsystems and how to integrate them during design. Strong motivation came from the appearance of new, successful fault-tolerant systems that offered new design insights and more operational experience.

The most mature version of the guidelines presented here is called a design paradigm ("pattern, example, model", according to Webster). It is an abstraction of observed design processes, in which the various steps often overlap. Its objective is to minimize the probability of oversights, mistakes, and inconsistencies that may occur during the implementation of fault tolerance. The paradigm is summarized in Figure 2. The first three steps: specification, implementation, and evaluation deal with the building of a new system. The last ("modification") step addresses the repair or augmentation of an existing design. The arrows between the implementation and evaluation steps emphasize the concurrency of both activities.

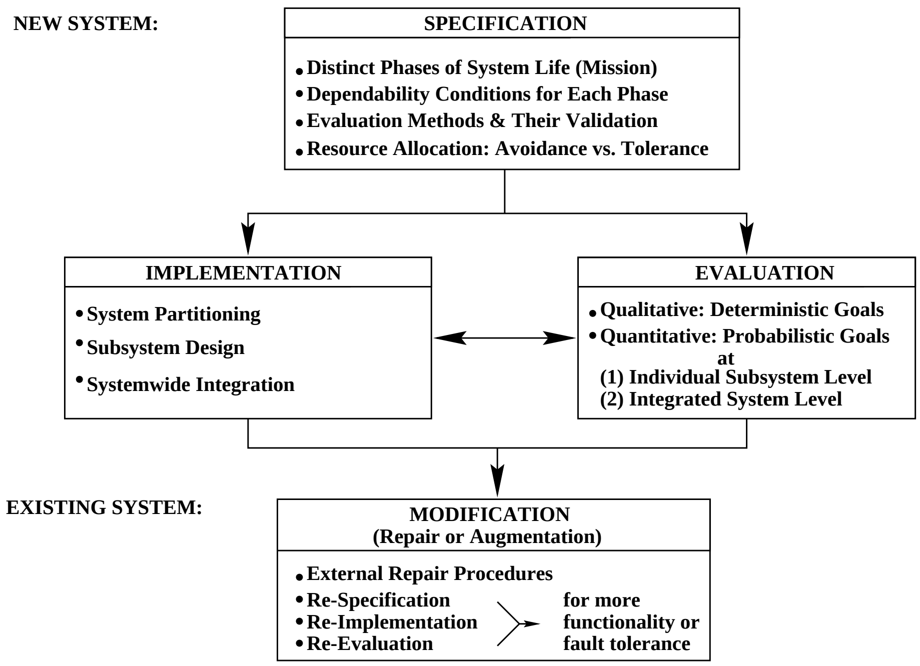


Figure 2: A Design Paradigm

## SPECIFICATION

*System requirements* that describe the services to be delivered in terms of functionality and performance are the starting point of specification.

### Phases of System Life (Mission)

The first step is to identify different *mission phases*, characterized by different environments and conditions of operation during the lifetime ("mission") of the system.

### Dependability Conditions

The following *dependability conditions* must be identified for each phase of system mission:

**Fault Classes.** Potential fault classes to be tolerated are summarized in Figure 1. Expected fault classes may differ because of differences in the technologies being used, environmental conditions, and operating interfaces. There are two parts of this problem: (a) internal faults that can arise within the system during each mission phase, and (b) external faults that can introduce errors into the system via the interfaces (I/O links, external fault tolerance support, human/machine interaction) during each mission phase.

**External Support.** Here we determine the availability and forms of repair by external agents (on-demand or periodic) and of remote support for fault tolerance functions: external diagnosis, reloading of software, etc..

**Modes Of Service.** The acceptability of possible different *modes of service* (full, reduced, degraded, emergency, safe shutdown, etc.) is determined for each phase. The required service level needs to be established for each mode.

**Dependability Goals.** They include system *availability, reliability, maintainability* and *safety*, as well as potentially varying *criticality* of service delivery and system *security* goals during different mission phases.

### Evaluation Methods

The *third* aspect of specification is to choose the *evaluation methods* that will be applied to assess the likelihood that the design will meet the specified dependability goals: availability, reliability, maintainability, safety, security. The independent means of validating the evaluation also need to be agreed upon and specified at this time. Fault and error occurrence scenarios need to be defined for the purpose of evaluation, including two (or more) independent near-coincident faults with overlapping effects, and the existence of latent errors.

### Resource Allocation

Finally, since the resources to make the system dependable are limited, the decision of how much to spend on fault tolerance vs. fault avoidance during implementation must be made before implementation begins.

## IMPLEMENTATION

### System Partitioning

The first stage of implementation is the definition of system structure, expressed by the interconnection of its building blocks and communications links. This process is called *partitioning.* Functionality, performance, and fault tolerance requirements, available technologies, and past experience all affect the choice of the hardware, software, communication and interface (I/O) subsystems that comprise the system being designed.

**Subsystem Definition.** The partitioning step of the paradigm requires collaboration between two groups of system architects: those responsible for performance and those responsible for dependability and fault tolerance. The partitioning that is accomplished should be a balanced solution that accommodates both sets of requirements: performance and dependable operation. For fault tolerance, the partitioning defines a set of fault and error *containment boundaries.* The purpose of the boundaries is to assure the independence of failure and absence of erroneous outputs for individual subsystems.

Three fundamental decisions on the implementation of fault tolerance need to be made at this time. Fault tolerance is accomplished by the successful execution of two functions: error and/or fault detection, followed by system recovery.

**Fault Tolerance Hierarchy.** The *first* decision is to define a *system-wide strategy* for the coordinated execution of these functions at both local (subsystem) and global levels. Intermediate levels (clusters, etc.) may be introduced in large systems. Specialized support subsystems for fault tolerance (e.g., service processors, software monitors, dedicated communication links) are also defined at this time and added to the system structure. The final part of this decision is which fault tolerance (i.e., detection and recovery) functions are to be (a) localized within individual subsystems; (b) shared by a set of functionally identical subsystems; (c) extended over a group of functionally diverse subsystems; or (d) made a global function of all subsystems. A hierarchical arrangement of two, three, or all four of these approaches offers other solutions of the detection and/or recovery problem.

**Redundancy Methods.** *Second*, the choice is made of the methods to introduce *redundant hardware* and *software resources*. Major options are: (a) multiple channel computation: duplex, triplex, etc., (b) spare subsystems, (c) degradation by exclusion of failed subsystems. Combinations of these techniques (e.g. hybrid redundancy, etc.) may be selected as well. The choice of time redundancy (repetition of computations) is also an option.

**Design Diversity.** The *third* major decision is whether *design diversity* [7] of redundant resources is to be implemented in order to tolerate design faults in some or all software and/or hardware subsystems. Design diversity is an especially effective means to assure safety in critical applications and to protect the most vital subsystems of complex systems and networks[9,10].

**Dependability Goals.** After the preceding decisions have been made, every subsystem is characterized by its own error and fault containment requirements, as well as the participation requirements in set, group, and/or global detection and recovery functions. At this time the system-level goals for availability, reliability, maintainability, safety, and security are apportioned among the sets of subsystems that compose the entire system.

## Subsystem Design

At the completion of partitioning each subsystem is characterized by a set of requirements for functionality, performance, and dependability. Dependability requirements include classes of faults to be tolerated, boundaries for error and fault containment, diversity objectives, modes of service, availability in terms of limits on the frequency and duration of service mode reduction and/or complete outage, and the bounds on recovery time. Furthermore there are the goals for subsystem reliability, maintainability, safety and security that are derived from the system-wide goals. Finally, the subsystem may be required to take part in *external* (i.e., group, set, or global) error detection, fault diagnosis, and/or recovery. The dependability goals are supported by either diverse or "cloned" replication at subsystem level and by built-in (*internal*) error detection, fault diagnosis (location), and recovery algorithms.

**Error Detection.** Errors are undesired states that are caused by active faults and may lead to subsystem failure. *Error detection* methods fall into two categories: (a) *concurrent* with delivery of service; and (b) *preemptive*, requiring that service delivery should be suspended during the application of detection procedures. Error detecting codes are an example of the former, and memory "scrubbing" an example of the latter method.

The choice of specific error detection methods strongly depends on the function of the subsystem. For example, error control codes are very often used in memory, while duplication and comparison is common for processors. The choice and location of error detection mechanisms must minimize the probability of undetected error propagation across subsystem boundaries. Additional detection techniques may be incorporated to support externally implemented (group, set, global) fault tolerance functions.

A special error detection requirement is that it is necessary to detect errors produced by faults in the fault tolerance mechanisms themselves. Furthermore, provisions are needed to check spare subsystems for dormant faults and to detect latent errors in stored information that is not subject to other forms of error detection.

**Recovery.** Every error signal that originates in a subsystem must invoke an *error recovery* procedure that attempts to restore the system to a valid state without known errors. Error recovery can be: (a) *backward*, returning the system to a previous, error-free state; or (b) *forward*, constructing a valid, error-free new state from existing (usually redundant) information.

After detection, it is necessary to record the detected erroneous state for analysis that will facilitate *fault diagnosis*. The main methods of diagnosis are: (a) analysis of a detected erroneous state; (b) repetition of previous operation(s); (c) application of test patterns; (d) use of special circuitry: BIST, micro-diagnostics, scan-in/scan-out, etc.; (e) use of independent (monitoring) subsystems.

A persistent fault may prevent successful error recovery. In this case, diagnosis identifies a faulty subsystem, and *fault removal* is performed by: (a) substituting a good spare subsystem; or (b) reconfiguring the system to continue functioning in spite of or without the faulty subsystem. Finally, an *independent validation* that the recovery was successful is a desirable attribute for every subsystem.

Recovery algorithms are usually hierarchical: a fast, local recovery is attempted first. If it does not succeed, a more extensive and time-consuming recovery procedure is invoked, and so on, until either recovery is accomplished, or the system is safely shut down by the last procedure of the recovery hierarchy. The time that is available to complete a recovery strongly depends on the application of the system. For example, a complete repetition that is acceptable in financial transaction handling would be too slow in real-time flight control. In addition to its internal recovery procedures a subsystem may also incorporate features that support external (group, set, global) cooperative recoveries.

The adequacy of the chosen internal error detection, fault diagnosis and recovery methods is assessed by an *evaluation* that takes place during subsystem design and may lead to modifications of initial choices.

## Systemwide Integration Of Fault Tolerance

The desired result of the system partitioning and subsystem design processes described above is an integrated set of local, intermediate, and global fault tolerance (detection and recovery) functions that are implemented to serve as a *protective infrastructure* that assures the timely and correct delivery of system services to its users in the presence of faults.

**The Infrastructure.** A systematically designed infrastructure has the following properties:

(a) it is *autonomous*, i.e., does not depend on other parts of the system (operating system, applications, personnel, etc.) for support in order to execute its functions;

(b) it is *distributed* and *fault-tolerant* itself;

(c) it has *dedicated communication* links, and also is able to use the main links;

(d) it is managed by highly protected "*hard core*" subsystem (or a hierarchy of such subsystems) that executes global decisions to assure system recovery.

It is interesting to note that the above properties are analogous to the protective mechanisms of the human body, in which the autonomous, distributed infrastructure consists of the immune system, pain sensors, and healing processes. This infrastructure is supported by the brain's cognitive function that invokes the use of medication, therapy, surgery, etc., as global recovery functions. The analogy will be further developed in the last section of this paper.

**Challenges Of Integration.** In practice, large and complex fault-tolerant systems have experienced unanticipated, failure-inducing improper interactions (e.g., the January 15, 1990 failure of AT&T long distance service) of otherwise well-designed fault-tolerant subsystems that have not been perfectly integrated into a distributed, fault-tolerant system. Especially difficult to integrate are the various hierarchical fault diagnosis and recovery sequences that support the localized fault tolerance of subsystems and those that support the fault tolerance of functional services ("threads") that are provided by two or more subsystems.

A second major integration problem is presented by two or more *nearly concurrent fault manifestations*. The large size and distributed nature of new systems lead to the possibility of two or more independent fault manifestations occurring close in time, most likely because of the previously undetected existence of latent errors and dormant faults. This, in turn, will require two or more recovery algorithms to be concurrently active, with the resulting risks of mutual interference, deadlocks, and behavior that is very difficult to anticipate.

A successful integration of several fault-tolerant subsystems into a cohesive infrastructure requires in-depth analysis as well as extensive experimentation, including *incremental demonstrations* of the capabilities, protection, completeness, and consistency of all fault tolerance functions and their specific implementation not only under single, but also under *multiple overlapping* fault conditions. The growing complexity of fault-tolerant systems in critical applications requires extensive use of the most advanced analytic and experimental evaluation techniques in order to eliminate inadequacies of global (systemwide) detection and recovery algorithms.

## EVALUATION

The evaluation of the adequacy of the chosen fault tolerance techniques is a continuous process during the steps of system partitioning, subsystem design, and system integration. At each step analytic and experimental evaluation is an important design tool that facilitates the choices between fault tolerance techniques and assesses the likelihood of meeting the dependability goals.

Successful completion of the design of a fault- tolerant system requires a convincing verification of two properties: the *completeness* of the design and its *potential to meet the stated goals* of availability, reliability, maintainability, safety and in some cases, security. The verification therefore must consist of two distinct evaluations: first *qualitative*, then *quantitative*.

### Qualitative Evaluation: Deterministic Goals

Qualitative evaluation generates deterministic predictions. It must be satisfied prior to quantitative evaluation, which generates probabilistic predictions. Otherwise, the evaluation may generate unreasonably optimistic predictions of availability and reliability because of unjustified simplifications that remain unnoticed.

The outcome of the evaluation is a "Yes/No" conclusion with respect to the four goals:

**Fault Tolerance Completeness And Consistency.** This evaluation is part of the systemwide integration. Here we use checklists of questions derived from the design paradigm, and experimental fault injection using worst-case scenarios.

**Absolute Tolerance.** This evaluates whether the system can survive one (or more than one) fault from a specified set and then execute a safe shutdown, usually stated as fail-operational/fail safe. A detailed design analysis is needed to prove this property.

**Absence Of Design Faults.** Here formal and heuristic methods such as proof of correctness, testing, and experimental methods are applicable.

**System Security Goals.** To evaluate deterministic security requirements, we use the same tools used to evaluate the absence of design faults.

### Quantitative Evaluation: Probabilistic Goals

**System Modeling.** We describe the design using a system evaluation model that is characterized by sets of physical, structural, repair, fault tolerance, and performance parameters for every subsystem.

**Fault Tolerance Parameter Measurement.** We obtain coverage and execution time parameters for all local and global detection and recovery functions. Fault injection experiments are an essential tool for this task.

**Forecasting of Dependability.** We use the model to predict system reliability, availability, maintainability, and safety. The existence of multiple service modes necessitates the prediction of *mean time between mode reductions* and *duration of mode reduction* in suitable measures (mean, 99th percentile, etc.) instead of a single availability prediction.

## MODIFICATION

Two major reasons for modification of an existing system are (1) *repair*, i.e., removal of newly discovered faults; and (2) *augmentation* of functionality, performance, and/or fault tolerance in an existing system. Subsystems are modified or new subsystems are added.

In both cases it is essential to modify the specification first and then to re-implement the subsystems with a complete reexamination and re-evaluation of the fault tolerance functions. Failure to return to the specification may cause gaps in the protection offered by fault tolerance. A very prominent example of such a "gap" was the omission of a timely synchronization check after an alternate reentry program was added that led to the aborting of the first flight of the U.S. Space Shuttle program 19 minutes before launch on April 10, 1981.

## THE "OFF-THE-SHELF" APPROACH

The preceding section deals with the bottom-up design paradigm for fault-tolerant systems that are composed of an integrated set of fault-tolerant subsystems. However, cost and development time constraints often lead to the use of pre-existing or "off-the-shelf" (OTS) subsystems, such as microprocessors, displays, sensors, operating systems, application software packages, etc., as building blocks for the design of systems that are expected to be highly dependable. The OTS items usually have few fault tolerance (f. t.) functions, and sometimes none at all.

To illustrate the nature of f. t. functions in OTS microprocessors, we consider the Intel Pentium Pro microprocessor [8] that appears to have the most complete set of f. t. functions among contemporary microprocessors. Also considered were Sun ultraSparc II, MIPS 10000, HP PA-8000, DEC Alpha 21164 and IBM/Apple/Motorola PowerPC 620 microprocessors.

### Defenses of the Pentium Pro

The Pentium Pro's ancestor Intel 486 provided parity checking for data bytes; the predecessor Pentium added address parity, introduced parity checks for cache, translation lookaside buffer, and microcode storage arrays, as well as a Machine Check exception with Address and Type registers. The Pentium also reintroduced the master/checker duplexing ("Functional Redundancy Checking") option that Intel had pioneered in the i432 processor chips.

The Pentium Pro integrates five components used by Pentium into a single component. It retains all Pentium f. t. techniques, replacing data byte parity with 8 ECC bits for SEC/DED operations. It uses two parity bits and provides a retry for the address bus, and also includes parity bits for two groups of control signals. The Machine Check Exception is generalized into a Machine Check Architecture, with 3 Global Control registers and 5 banks of 4 Error Reporting registers each [8].

We see that as complexity of the chips increases, more f. t. functions are added; however, major "OTS" drawbacks remain in the Pentium Pro. They are:

(1) Protection by parity and ECC is limited to storage arrays and communication links which are the "easy" parts to check; the more complex data and instruction processing logic, for example, remains unchecked.

(2) The extensive (over 1400 pages) system developer's documentation [8] commingles error handling with all other information. There is no comprehensive top-down view of the f. t. techniques and their interrelationships. The management of most error conditions is relegated to a "central agent" that remains entirely unexplained. There is a significant risk that the developer can overlook or misinterpret the details of error handling.

(3) The use of the Machine Check Architecture is optional and must be enabled by software. This leaves open the possibilities of its accidental or malicious disabling during operation. The Machine Check Exception Handler software only logs machine status and error information and then shuts down the system, since there are no on-chip recovery procedures that can be invoked.

(4) The master/checker duplexing ("FRC") is a "throwaway" solution at twice the cost of one microprocessor. Error detection is delayed until the error reaches the component's output; at that time shutdown, BIST, and restart is the only recovery option left, even if the cause was only a "soft" parity error that a local retry could eliminate.

**Retrofit Solutions**

The problem of "retrofitting" systems built of OTS subsystems for fault tolerance is very difficult. The lack of OTS hardware support for fault tolerance leaves the "top-down" solution: to build a software "monitor" subsystem (such as the Pentium Pro's Machine Check Exception Handler) that resides and executes on the OTS hardware elements. The monitor attempts to check all subsystems for indications of failure, records abnormal symptoms, and initiates shutdowns, BIST, and restarts when needed. The weakness of this approach is the lack of protection for the monitor software itself, since it resides and executes on an OTS processor, and the on/off handling of recovery.

A costly and effective method to build highly dependable systems that use OTS subsystems is to employ multiple-channel computation with diverse hardware and software in each channel [7]. Variations of this approach have been successfully used in safety-critical flight control [9], rail transportation [10] and similar applications that employ well defined, cyclic control algorithms.

However, cost and the complexity of applications preclude the use of the above solution in most distributed, heterogeneous systems with OTS components. A potential "retrofit" solution here is to implement a small, highly fault-tolerant hardware subsystem that monitors its operation, assures the protection of data integrity, and manages recovery when part of the system fails by switching in spare resources or reconfiguring the system. The Test-and-Repair Processor of the JPL-STAR computer [6] is the first prototype of such a monitor. An excellent contemporary example of a hardware monitor is the Processor

Controller of the IBM ES/9000 Type 9021 computer [11]. The effectiveness of the hardware monitor is entirely limited by the nature of OTS elements being monitored; for example, a Machine Check Exception(MCE) message from a Pentium Pro means that the processor must be shut down, even if the MCE indicates only a "soft" parity error due to an SEU. It is apparent that more survivable OTS-based systems can only be built when the OTS subsystems themselves are fault-tolerant. One way to attain this goal is discussed in the final section.

### The AAS: a Disastrous Experience

The difficulty of the off-the-shelf approach is illustrated by the AAS system for air traffic control in the U.S. [12]. Its system level specification stated that "availability shall be the most important consideration in the design of AAS," and fault tolerance was explicitly required as the means to assure very high availability. The allowed time of service denial is about 3 seconds per year for critical functions (below emergency mode), 32 seconds per year for conflict detection functions (below reduced service mode) and about 2.5 minutes per year for planning and display data recording functions (below full service mode). Two 42-month Design Competition contracts of $200 million each were awarded in August, 1984 to Hughes Aircraft Co. and IBM Federal Systems Division, and a $4.8 billion contract was awarded to the winner - IBM - in August, 1988.

The IBM design employed IBM RISC System/6000 (RS/6000) processors as the basic subsystem interconnected by a redundant token ring local communication network. Fault tolerance is implemented for groups of processors by software-implemented Group Service Availability Management, while the highest level supervision is exercised by the Global Availability Management Service, residing on a group of up to four servers [13]. The classes of faults to be tolerated were defined as those that cause processor crash (fail-silent), omission (non-response), or performance (lateness) failures. Faults that cause timely, but incorrect outputs are excluded by definition.

Fault tolerance in the RS/6000 exists for the memory (ECC, bit steering, scrubbing) and data buses (parity). The chip set of the CPU is limited to data path parity and a built-in self-test sequence after powerup. The "fail-silent" assumption was made in the implementation of AAS fault tolerance [13], which implies a coverage of 1.0 within the processor with respect to all incorrect outputs, yet the off-the-shelf RS/6000 offers no such claim.

The largest facility of the AAS was the Area Control Computer Complex (ACCC) which contains up to 400 processors in a distributed configuration. Twenty three such facilities were needed for the entire U.S.A. The ACCC design clearly was a crucial test of the usability of off-the-shelf processors and the upward scalability of the Group Service Availability Management concept.

Regretably, on June 3, 1994 the FAA announced the cancellation of the ACCC and of the smaller Terminal AAS system procurement. The building of the less complex Tower CCC was to be continued, and the software for the Initial Sector Suite System that had already cost over $1 billion was to be "analyzed to determine whether it can be operated and maintained."

It is disappointing to note the failure of the world's most ambitious and costly attempt to build the highly available ACCC system using off-the-shelf processors and software-implemented fault tolerance attributes. We can only hope that sufficient information on the successes and failures of the fault tolerance implementation of the ACCC will be made available soon for the benefit of the builders of future systems.

## THE GOAL: HIGH-CONFIDENCE COMPUTING FOR ALL

The complexity of today's computing and communication systems is already comparable to the complexity of some living organisms. However, the evolution of life forms and of computing systems began at opposite ends. For living organisms, survival of individuals and species came first, and higher cognitive functions evolved gradually over billions of years, culminating with the emergence of homo sapiens. On the other hand, computers were built to emulate man's intellectual functions. "Survival" mechanisms, such as error detection and redundancy, had to be introduced in early computers because vacuum tubes, relays, and other components frequently malfunctioned [2]. A few years later, much more reliable transistors and magnetic cores were introduced, and second generation hardware (e.g., IBM 7090, CDC 6600, etc.) dropped all checking except some parity and error codes for storage tapes. This proved to be inadequate, and checking of general-purpose machines gradually came back with the third generation, culminating with some excellent contemporary designs, such as the IBM ES/9000 Type 9021 processor [11].

### Another Setback on the Way to High-confidence Computing

The preceding section shows that the advent of the microprocessor has again caused a setback to the confidence level of general-purpose computing. The Pentium Pro falls short of the checking provided thirty years ago by IBM System 360 and its contemporaries. It is evident that the explosive growth of complexity, speed, and performance of single-chip processors has not led to a proportional growth of on-chip error detection and recovery features.

Thirty years ago the concept of fault tolerance gathered diverse techniques that were used to cope with the consequences of faults into a cohesive set[1]. Since then, fault tolerance has evolved as the survival attribute of computing and communication systems, especially of high-confidence systems in applications in which human lives are at risk when a system fails [2, 7]. Many successes are evident: control of high-speed transportation, Tandem and other systems for dependable transaction processing, etc. Yet it is also clear that the on-chip checking of high-performance microprocessors of today is still in the pre-fault-tolerance age. At the Comdex in November 1996 Intel's CEO Andy Grove predicted a chip with a billion transistors running at ten gigahertz by the year 2011, but the issue of "more confidence" remained unmentioned.

Why do the most prevalent microprocessors remain low-confidence chips when high-confidence systems are flying our airliners and dispensing our cash? The answer has two parts:

(1) Reasonably high-confidence systems can be built from contemporary low-confidence chips when specific applications justify the high cost, but the presently small market does not motivate the production of fault-tolerant chips.

(2) Consumers and end users, who would benefit most from inexpensive high-confidence computing, generally are not aware of the advantages of making a commodity microprocessor as a self-contained fault-tolerant system, so manufacturers aren't motivated to provide such chips.

### Fault-tolerant Microprocessors for All

High-confidence computing will be affordable for all users when every microprocessor is a self-contained fault-tolerant system as well as a building block for a fault-tolerant multi-chip system. To secure fault-tolerant microprocessors, we need to determine

(1) What technical knowhow makes a microprocessor fault-tolerant, and

(2) What will motivate the major chip makers to build fault-tolerant chips?

The answer to the first question has already been presented in this paper: to treat a single chip as a self-contained system and to apply the design paradigm to the subsystems of the chip. At the same time, the chip is to be viewed as a subsystem of a multi-chip system and designed to provide and to receive support of global f. t. functions at the higher level.

For example, a fault-tolerant Pentium Pro would remain functionally equivalent to the present implementation, but the external Exception Handler would be invoked only in the rare cases of faults that permanently damage the chip or simultaneously cause a large number of errors. The common "soft" errors due to transient faults (such as SEUs) would be handled on the chip. In addition, the chip would have the ability to take part in exception handling for other chips of the system. As further extensions, on-chip redundancy can be introduced to handle permanent faults and design diversity to protect the chip's "hard core" against design faults.

The second question can also be stated as: "What would motivate Dr. Grove of Intel to predict at least a ten-fold (or maybe hundred-fold?) growth, compared to the Pentium Pro, in the confidence that we will be able to justifiably place in his billion-transistor, ten-gigahertz chips in the year 2011?"

To make it happen we must communicate the nature and the advantages of on-chip fault tolerance to the community of users who would benefit from high-confidence communications and computing. Customer demand is the force that will motivate the manufacturers of microprocessors to take a pause in their race for more function, speed, and storage capacity in order to introduce autonomous fault tolerance at chip level. These elements will serve to build affordable high-confidence systems that do not rely on software alone to assure the delivery of continuous and error-free service.

A means to get the message across to the users and to interest the builders is proposed next.

### A Conceptual Model: the Challenge

The human body with its defenses: the immune system, the sense of pain, and the healing processes, is proposed as a conceptual model ("example for imitation" according to Webster) for a high-confidence system in which fault tolerance is an integral attribute of every hardware element.

The key analogies are: the body for hardware, and the cognitive processes that are supported by the body for software, whose execution delivers the required high-confidence services for which the system is programmed.

Four fundamental attributes of the immune system [14] are particularily relevant: (1) it functions continuously and is autonomous, i.e., independent of cognition; (2) its elements (lymph nodes, other lymphoid organs, lymphocytes) are distributed throughout the body, serving all its organs; (3) it has its own communication links - the network of lymphatic vessels; and (4) its elements (organs and vessels) themselves are redundant and in some cases diverse.

All four attributes enumerated above also would characterize a systematically designed fault-tolerant microprocessor, or some other hardware element, as well as a network of such elements (see the section "Systemwide integration of fault tolerance").

The sense of pain in a body is analogous to the messages from the hardware f. t. infrastructure to the fault management software that may take further action, just as the mind may decide to invoke higher-level defenses: medication, physical therapy, etc. It is to be noted that most computing subsystems built today (microprocessors, disk drives, displays, etc.) lack sufficient local error detection and recovery attributes, and external software must be involved in managing most of their fault conditions as in the Pentium Pro. That is analogous to expecting cognition to compensate for the absence of immunity or of the sense of pain, which is a risky substitution in a computer just as it would be in a body.

The healing processes of wounds and fractures in a body are highly sophisticated and as yet lack a parallel in electronics. The rougher analogy here is with automatic repair by the use of spares or by switch-out of failed elements both on-chip and at complete chip levels.

In addition to in-body diversity, such as found in the immune system, it is worth noting that diversity is the attribute of a species that protects it against extinction due to genetic defects in the individual members, while the diversity among species has assured the continuity of life on Earth. The self-evident analogy here is the use of hardware and software design diversity[7] in fault-tolerant systems in order to tolerate design faults within one chip as well as at the whole-chip level.

### How Can the Model Help?

In conclusion let us ask: is the proposed model likely to succeed by raising an expectation for affordable high-confidence computing for all? There are three reasons that offer a strong hope for its success.

First, we are setting up an analogy with the most dependable information processing systems in existence - the various species of living creatures that have survived and evolved over millions of years on our continually changing planet.

Second, the analogy reaches out to appeal to a wide spectrum of intelligent people, since it does not require knowledge of computer science or technology, or an *a priori* belief in the benefits of fault tolerance.

Third, we need a model and a goal that will stimulate the imaginations of the coming generation of system builders and lead to solutions that will far surpass the best that we can do today.

While it may seem to some readers that taking the defenses of the human body as the model to imitate is too ambitious a goal, such a step forward is well worth taking if it will help to avoid a continuing procession of ever more complex and faster low-confidence chips in the future.

### A Pair of Concluding Predictions

I conclude with the confident prediction that while the speed of computing will ultimately be limited by the laws of physics, the demand for affordable high-confidence computing will not cease as long as humans will use computers to enhance the quality of their lives and to be entertained.

Eventually, one enterprising chip builder will deliver the first fault-tolerant microprocessor at a competitive price, and soon thereafter fault tolerance will be considered as indispensable in computers as immunity in the human body. The other chipmakers will follow suit or go the way of the dinosaurs. Once again, Darwin will be proven right!

## ACKNOWLEDGMENTS

The year 1997 has marked 42 years since the author's first professional work on system dependability at JPL. It has been my good fortune to work with and learn from many colleagues at the University of Illinois, Caltech's Jet Propulsion Laboratory and the UCLA Computer Science Department. University of Illinois professors James E. Robertson and David E. Muller were my mentors and role models for my academic career.

The work with my friends in the IEEE Computer Society's Technical Committee on Fault-Tolerant Computing and IFIP Working Group 10.4 on Dependable Computing and Fault Tolerance has been a rewarding experience and a source of inspiration for this paper. Mr. Yutao He has contributed valuable assistance in the survey of microprocessors and in the preparation of this text.

This presentation is an expanded version of a paper that appeared in *Computer*, Vol. 30, No. 4, April, 1997.

## REFERENCES


[1] Avižienis, A., "Design of Fault-Tolerant Computers," *AFIPS Conference Proceedings, 1967 Fall Joint Computer Conference*, Vol. 31, Washington D.C.: Thompson, 1967, pp. 733-743.

[2] Avižienis, A., Kopetz, H., Laprie, J.-C., editors; *The Evolution of Fault-Tolerant Computing*, Springer-Verlag: Vienna and New York, 1987.

[3] von Neumann, J., "Probabilistic Logics and the Synthesis of Reliable Organisms from Unreliable Components," Automata Studies, C.E. Shannon and J. McCarthy editors, *Annals of Math Studies* No 34, Princeton, NJ: Princeton University Press, 1956, pp. 43-98.

[4] Moore, E. F., Shannon, C. E., "Reliable Circuits Using Less Reliable Relays," *Journal of the Franklin Institute* 262, No. 9 and 10, Sept. Oct. 1956, pp. 191-208 and 281-297.

[5] Pierce, W. H., *Failure-Tolerant Computer Design*, Academic Press: New York and London, 1965.

[6] Avižienis, A., Gilley, G. C., Mathur, F. P., Rennels, D. A., Rohr, J. A., Rubin, D. K., "The STAR (Self-Testing-and-Repairing) Computer: An Investigation of the Theory and Practice of Fault-Tolerant Computer Design," *IEEE Trans. on Computers*, Vol. C-20, No. 11, November 1971, pp. 1312-1321.

[7] Avižienis, A., Laprie, J.-C., "Dependable Computing: From Concepts to Design Diversity," *Proceedings of the IEEE*, Vol. 74, No. 5, May 1986, pp. 629-638.

[8] *Pentium Pro Family Developer's Manual*, Vols 1, 2, 3; Intel Corporation: Mt. Prospect, IL; 1996.

[9] Yeh, Y. C., "Dependability of the 777 Primary Flight Control System", *Preprints of Fifth International Working Conference on Dependable Computing for Critical Applications*, Coordinated Science Lab, Univ. of Illinois at Urbana-Champaign; September 27-29, 1995, pp. 1-13.

[10] Profeta, J. A., et al., "Safety-Critical Systems Built with COTS", *Computer*, Vol. 29, No. 11, November 1996, pp. 54-60.

[11] Chen, C. L., et al., "Fault-Tolerance Design of the IBM Enterprise System/9000 Type 9021 Processor", *IBM Journal of Research and Development*, Vol. 36, No. 4, July 1992, pp. 765-779.

[12] Hunt, V. R., Kloster, G. V., guest editors, "The FAA's Advanced Automation Program", theme issue, *Computer*, Vol. 20, No. 2, February 1987.

[13] Cristian, F., Dancey, R. D., Dehn, J. D., "Fault Tolerance in the Advanced Automation System," *Digest of FTCS-20, the 20th International Symposium on Fault Tolerant Computing*, June 1990, pp. 6-17.

[14] Nossal, G. J. V., "Life, Death and the Immune System", *Scientific American*, Vol. 269, No. 3, September 1993, pp. 52-62.

## THE AUTHOR'S BIOGRAPHY

Algirdas Avižienis is an emeritus professor at UCLA and Rector Emeritus at Vytautas Magnus University in Kaunas, Lithuania. As a senior research engineer at Caltech's Jet Propulsion Laboratory, he initiated research on interplanetary spacecraft computers in 1961 and published the concept of fault tolerance in 1967. His teaching and research interests include computer design, digital arithmetic, and fault-tolerant computing. He received a BS, an MS, and a PhD, all in electrical engineering, from the University of Illinois, Urbana-Champaign. A fellow of the IEEE and a member of the Lithuanian Academy of Sciences, he received the NASA Apollo Achievement Award, the American Institute of Aeronautics and Astronautics Information Systems Award, the NASA Exceptional Service Medal, the IFIP Silver Core, and the degree Docteur Honoris Causa from the Institut National Polytechnique in Toulouse, France. He was the founding chairman of the IEEE CS Technical Committee on Fault-Tolerant Computing and of the IFIP WG 10.4 on Dependable Computing and Fault Tolerance.