FAULT TOLERANT COMPUTING

We say that a computer is fault tolerant if it fulfills its intended function despite the presence or the occurrence of faults. Fault tolerance is achieved through the introduction and the management of redundancy. A fault-tolerant computer may contain several forms of redundancy, depending on the types of faults it is designed to tolerate. For example, structural redundancy can be used to provide continued system operation even if some components have failed; information redundancy in the form of error control codes can allow the detection or correction of data errors; timing redundancy can be used to tolerate transient faults, and so on.

Redundancy techniques have been employed since the inception of the computer era. In those early days, computer

J. Webster (ed.), *Wiley Encyclopedia of Electrical and Electronics Engineering*. Copyright © 1999 John Wiley & Sons, Inc.

components were so unreliable that redundancy techniques were almost essential for the computer to successfully complete a lengthy computation. Indeed, extensive parity checking and duplicated arithmetic logic units were used in the very first commercial computer, the UNIVAC 1 (c. 1951) (1). The term *fault-tolerance* itself can be traced back to early work on onboard computers for unmanned spacecraft (2), which employed large numbers of spare subsystems to be able to survive missions of 10 years or more in deep space.

Of course, the reliability of hardware components has vastly improved since those early days. However, since computing technology now permeates almost every aspect of modern society, there is a growing potential for computer failures to cause us great harm, leading to loss of life or money, or damage to our health or to our environment. Consequently, fault-tolerance techniques are an essential means to ensure that we can depend on computers used in critical applications. Currently, fault-tolerance techniques are being employed as a means to protect critical computing systems not only from physical component failures, but also from faults in hardware and software design, from operator errors during human–machine interaction, and even from malicious faults perpetrated by felons.

In this article, we describe the essential principles of fault-tolerant computer system design. We first establish the basic concepts and terminology of dependable computing. Two sections then detail the various techniques that can be used for error detection and error recovery, with illustrative examples drawn from fault-tolerant systems currently in operation. A further section is devoted to fault-tolerance viewed in the context of distributed computing systems. The following two sections discuss the fault-tolerant system development process and present a case study of fault-tolerance techniques employed in the Ariane 5 space launcher. In the final section, we describe some future directions for fault-tolerant computing. We review the fault classes that are currently the subject of fault-tolerance research and discuss some of the current economic challenges.

BASIC CONCEPTS AND TERMINOLOGY

This section, based on Ref. 3, introduces the concept of dependability within which the fault tolerance approach plays a major role. It first presents some condensed definitions on dependability. These basic definitions are then commented on and supplemented in the three subsequent sections.

Basic Definitions

Dependability is that property of a computer system such that reliance can justifiably be placed on the service it delivers. The *service* delivered by a system is its behavior as it is perceived by its user(s); a *user* is another system (physical, human) which interacts with the former.

Depending on the application(s) intended for the system, different emphasis may be put on different facets of dependability, i.e., dependability may be viewed according to different, but complementary, properties, which enable the attributes of dependability to be defined:

- The readiness for usage leads to *availability*
- The continuity of service leads to *reliability*

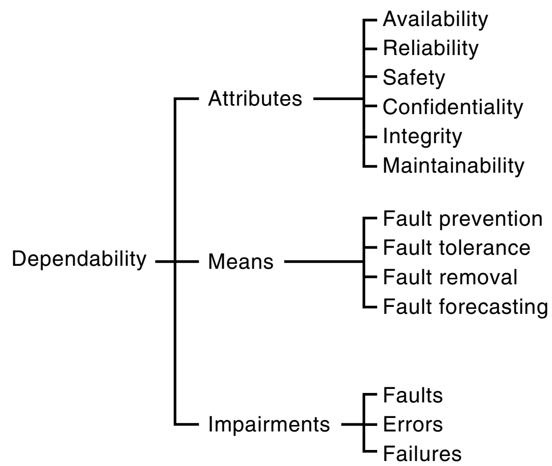

Figure 1. A classification of the main concepts of dependability.

- The nonoccurrence of catastrophic consequences on the environment leads to *safety*
- The nonoccurrence of unauthorized disclosure of information leads to *confidentiality*
- The nonoccurrence of improper alterations of information leads to *integrity*
- The aptitude to undergo repair and evolution leads to *maintainability*

Associating integrity and availability with respect to authorized actions, together with confidentiality, leads to *security*.

A system *failure* occurs when the delivered service stops fulfilling the system *function,* the latter being what the system is intended for. An *error* is that part of the system state that is liable to lead to subsequent failure: an error affecting the service is an indication that a failure occurs or has occurred. The adjudged or hypothesized cause of an error is a *fault*.

The development of a dependable computing system calls for the combined utilization of a set of methods that can be classed into:

- Fault prevention: how to prevent fault occurrence or introduction
- Fault tolerance: how to provide a service that fulfills the system function in spite of faults
- Fault removal: how to reduce the presence (number, seriousness) of faults
- Fault forecasting: how to estimate the present number, the future incidence, and the consequences of faults

The notions introduced up to now can be grouped into three classes (Fig. 1):

1. The attributes of dependability: availability, reliability, safety, confidentiality, integrity, maintainability; these (a) enable the properties which are expected from the system to be expressed, and (b) allow the system quality resulting from the impairments and the means opposing to them to be assessed;

2. The means for dependability: fault prevention, fault tolerance, fault removal, fault forecasting; these are the methods and techniques enabling one (a) to provide the ability to deliver a service on which reliance can be placed, and (b) to reach confidence in this ability.
3. The impairments to dependability: faults, errors, failures; they are undesired—but not in principle unexpected—circumstances causing or resulting from the lack of dependability.

A major strength of the dependability concept as formulated here is its integrative nature:

- It allows for the classical notions of reliability, availability, and so on to be put into perspective.
- It provides a unified presentation allowing for the understanding and mastering of the various impairments, while preserving their specificities via the various failure modes and fault classes that can be defined.
- The model provided for the means for dependability is extremely useful, as those means are much more orthogonal to each other than the usual classification according to the attributes of dependability, with respect to which the design of any real system has to perform trade-offs due to the fact that these attributes tend to be in conflict with each other.

The following sections expand on the basic definitions concerning the dependability attributes and impairments and the means for dependability. Fault-tolerant computing is the focus of this article so we will concentrate essentially on this aspect. A more detailed treatment of these basic definitions, and in particular of the respective role of and dependencies between fault tolerance, fault removal, and fault forecasting, can be found in Refs. 3 and 4.

The Attributes of Dependability

The attributes of dependability have been defined according to different properties, which may be emphasized more or less depending on the intended application of the computer system considered:

- Availability is always required, although to a varying degree depending on the application.
- Reliability, safety, and confidentiality may or may not be required according to the application.

Integrity is a prerequisite for availability, reliability, and safety, but may not be so for confidentiality (for instance when considering attacks via covert channels or passive listening).

Whether a system holds the properties which have enabled the attributes of dependability to be defined should be interpreted in a relative, probabilistic sense, and not in an absolute, deterministic sense: due to the unavoidable presence or occurrence of faults, systems are never totally available, reliable, safe, or secure.

The definition given for maintainability goes deliberately beyond *corrective maintenance,* aimed at preserving or improving the system's ability to deliver a service fulfilling its function (relating to repairability only), and encompasses via evolvability the other forms of maintenance: *adaptive maintenance,* which adjusts the system to environmental changes (e.g., change of operating systems or system data-bases), and *perfective maintenance,* which improves the system's function by responding to customer—and designer—defined changes, which may involve removal of specification faults (5).

Security has not been introduced as a single attribute of dependability, in agreement with the usual definitions of security, which view it as a composite notion, namely "the combination of confidentiality, the prevention of the unauthorized disclosure of information, integrity, the prevention of the unauthorized amendment or deletion of information, and availability, the prevention of the unauthorized withholding of information" (6).

The variations in the emphasis to be put on the attributes of dependability have a direct influence on the appropriate balance of the means to be employed to make the resulting system dependable. This is an all the more difficult problem as some of the attributes are antagonistic (e.g., availability and safety, availability and security), and therefore imply design trade-offs. The problem is further exacerbated by the fact that the dependability dimension of the computer design space is less understood than the cost and performance dimensions (7).

The Impairments to Dependability

In this section, after examining the failure modes, we describe the various fault classes to be considered. Finally, we address the fault pathology issue by discussing further the notions of fault, error, and failure and identifying their respective manifestations and relationships.

Failures and Failure Modes. A system may not, and generally does not, always fail in the same way. The ways a system can fail are its *failure modes,* which may be characterized according to three viewpoints: domain, perception by the system users, and consequences on the environment.

The failure domain viewpoint leads one to distinguish:

- Value failures: the value of the delivered service does not fulfill the system function.
- Timing failures: the timing of the service delivery does not fulfill the system function.

A class of failures relating to both value and timing are the *halting failures:* system activity, if any, is no longer perceptible to the users. According to how the system interacts with its user(s), such an absence of activity may take the form of (a) frozen outputs (a constant value service is delivered; the constant value delivered may vary according to the application, e.g., last correct value, some predetermined value, etc.), or of (b) a silence (no message sent in a distributed system). A system whose failures can be—or more generally are to an acceptable extent—only halting failures, is a *fail-halt system*; the situations of frozen outputs and of silence lead respectively to *fail-passive* systems and to *fail-silent* systems (8).

The failure perception viewpoint leads one to distinguish, when a system has several users:

- Consistent failures: all system users have the same perception of the failures.

- Inconsistent failures: the system users may have different perceptions of a given failure; inconsistent failures are usually termed, after Ref. 9, *Byzantine failures.*

Grading the consequences of the failures upon the system environment enables the failure *severities* to be defined. The failure modes are ordered into severity levels, to which are generally associated maximum admissible probabilities of occurrence. Two extreme levels can be defined according to the relation between the benefit provided by the service delivered in the absence of failure and the consequences of failures:

- Benign failures, where the consequences are of the same order of magnitude as the benefit provided by service delivery in the absence of failure
- Catastrophic failures, where the consequences are incommensurably greater than the benefit provided by service delivery in the absence of failure

A system whose failures can only be—or more generally are to an acceptable extent—benign failures is a *fail-safe system.* The notion of failure severity enables the notion of criticality to be defined: the *criticality* of a system is the highest severity of its (possible) failure modes. The relation between failure modes and failure severities is highly application-dependent. However, there exists a broad class of applications where inoperation is considered as being a naturally safe position (e.g., ground transportation, energy production), whence the direct correspondence that is often made between fail-halt and fail-safe (10,11). Fail-halt systems (either fail-passive or fail-silent) and fail-safe systems are however examples of *fail-controlled systems,* i.e., systems which are designed and realized in order that they may only fail—or may only fail to an acceptable extent—according to restrictive modes of failure, e.g., frozen output as opposed to delivering erratic values, silence as opposed to babbling, consistent failures as opposed to inconsistent ones; fail-controlled systems may in addition be defined by imposing some internal state condition or accessibility, as in the so-called *fail-stop* systems (12).

Errors. An error was defined as being liable to lead to subsequent failure. Whether or not an error will actually lead to a failure depends on three major factors:

1. The system composition, and especially the nature of the existing redundancy:
   a. Intentional redundancy (introduced to provide fault tolerance) which is explicitly intended to prevent an error from leading to failure,
   b. Unintentional redundancy (it is practically difficult if not impossible to build a system without any form of redundancy) which may have the same—unexpected—result as intentional redundancy.
2. The system activity: an error may be overwritten before creating damage.
3. The definition of a failure from the user's viewpoint: what is a failure for a given user may be a bearable nuisance for another one. Examples are (a) accounting for the user's time granularity: an error which "passes through" the system-user(s) interface may or may not be viewed as a failure depending on the user's time granularity, (b) the notion of "acceptable error rate"—implicitly before considering that a failure has occurred—in data transmission.

This discussion explains why it is often desirable to explicitly mention in the specification such conditions as the maximum outage time (related to the user time granularity).

Faults and Fault Classes. Faults and their sources are extremely diverse. They can be classified according to five main viewpoints, their phenomenological cause, their nature, their phase of creation or of occurrence, their situation with respect to the system boundaries, and their persistence.

The phenomenological causes leads one to distinguish (13):

- Physical faults, which are due to adverse physical phenomena
- Human-made faults, which result from human imperfections

The nature of faults leads one to distinguish:

- Accidental faults, which appear or are created fortuitously
- Intentional faults, which are created deliberately, with or without a malicious intention

The phase of creation with respect to the system's life leads one to distinguish:

- Development faults, which result from imperfections arising either (1) during the development of the system (from requirement specification through to implementation) or during subsequent modifications, or (2) during the establishment of the procedures for operating or maintaining the system
- Operational faults, which appear during the system's exploitation

The system boundaries leads one to distinguish:

- Internal faults, which are those parts of the state of a system which, when invoked by the computation activity, will produce an error
- External faults, which result from interference or from interaction with its physical (electromagnetic perturbations, radiation, temperature, vibration, etc.) or human environment

The temporal persistence leads one to distinguish:

- Permanent faults, whose presence is not related to pointwise conditions whether they be internal (computation activity) or external (environment)
- Temporary faults, whose presence is related to such conditions, and are thus present for a limited amount of time

The notion of temporary fault deserves the following comments:

- Temporary external faults originating from the physical environment are often termed transient faults.

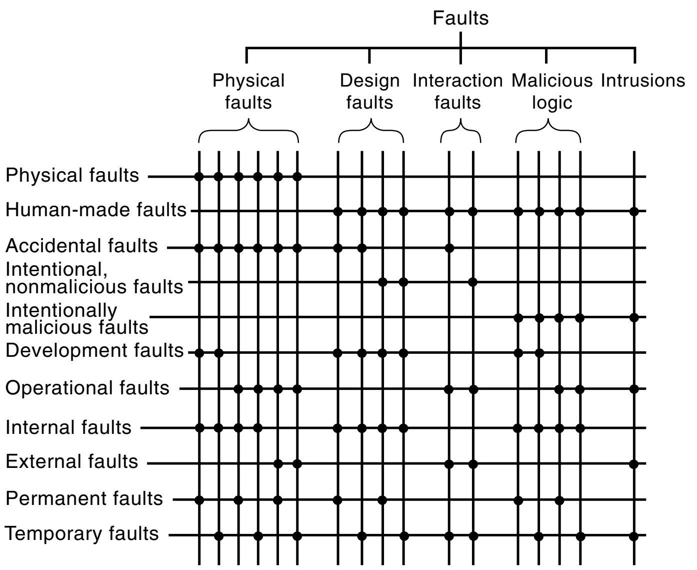

Figure 2. Fault classes resulting from pertinent combinations of the basic classification viewpoints.

- Temporary internal faults are often termed intermittent faults; these faults result from the presence of rarely occurring combinations of conditions.

In practice, the number of likely combinations is more restricted than the 48 different fault classes that could be obtained from the 5 viewpoints: only 17 combinations are indicated in Fig. 2, which also gives the usual labeling of these combined classes of faults.

These labels are commonly used to designate one or several combined fault classes in a condensed manner. In particular, the label *physical faults* relates to the various combinations that precisely share physical faults as elementary faults. Two comments are in order regarding the human-made fault classes:

1. Intentional, nonmalicious, design faults result generally from tradeoffs, either (a) aimed at preserving acceptable performance or at facilitating system utilization, or (b) induced by economic considerations. Such faults can be sources of security breaches, under the form of covert channels. Intentional, nonmalicious interaction faults may result from the action of an operator either aimed at overcoming an unforeseen situation, or deliberately violating an operating procedure without having developed the consciousness of the possibly damaging consequences of his or her action. These classes of intentional nonmalicious faults share the property that, often, it is realized that they were faults only after an unacceptable system behavior, thus a failure, has ensued.
2. Malicious logic encompasses development faults such as Trojan horses, logic or timing bombs, trapdoors, as well as operational faults (for the considered system) such as viruses or worms (14).

Fault Pathology. The creation and manifestation mechanisms of faults, errors, and failures may be summarized as follows:

1. A fault is *active* when it produces an error. An active fault is either (a) an internal fault that was previously *dormant* and which has been activated by the computation process, or (b) an external fault. Most internal faults cycle between their dormant and active states. Physical faults can directly affect the hardware components only, whereas human-made faults may affect any component.
2. An error may be latent or detected. An error is *latent* when it has not been recognized as such; an error is *detected* by a detection algorithm or mechanism. An error may disappear before being detected. An error may, and in general does, propagate; by propagating, an error creates other—new—error(s). During operation, the presence of active faults is determined only by the detection of errors.
3. A failure occurs when an error "passes through" the system-user interface and affects the service delivered by the system. The consequence of a component failure is a fault (a) for the system that contains the component, and (b) as viewed by the other component(s) with which it interacts; the failure modes of the failed component then become fault types for the components interacting with it.

These mechanisms enable the "fundamental chain" to be completed:

$$\cdots \rightarrow \text{failure} \rightarrow \text{fault} \rightarrow \text{error} \rightarrow \text{failure} \rightarrow \text{fault} \rightarrow \cdots$$

The arrows in this chain express a causality relationship between faults, errors, and failures. They should not be interpreted restrictively: by propagation, several errors can be generated before a failure occurs.

Situations involving multiple faults and/or failures are frequently encountered. Consideration of their causes leads one to distinguish:

- Independent faults, which are attributed to different causes
- Related faults, which are attributed to a common cause

Related faults generally manifest themselves by *similar* errors, whereas independent faults usually cause *distinct* errors, although it may happen that independent faults lead to similar errors (15). Similar errors cause common-mode failures.

Two final comments, relative to the words, or labels, fault, error, and failure:

1. Their exclusive use in this document does not preclude the use in special situations of words which designate, briefly and unambiguously, a specific class of impairment; this is especially applicable to faults (e.g., bug, defect, deficiency) and to failures (e.g., breakdown, malfunction, denial-of-service).
2. The assignment made of the particular terms fault, error, and failure simply takes into account current us-

age: (a) fault prevention, tolerance, and diagnosis, (b) error detection and correction, (c) failure rate.

Techniques for Fault Tolerance

Fault tolerance is carried out by two main forms of activities: error processing and fault treatment. *Error processing* is aimed at removing errors from the computational state, if possible before failure occurrence; *fault treatment* is aimed at preventing faults from being activated—again. We first define each of these two activities and then express some additional comments.

Error Processing. Error processing can be carried out via three primitives:

1. Error detection, which enables an erroneous state to be identified as such
2. Error diagnosis, which enables the assessment of the damage caused by the detected error, or by errors propagated before detection
3. Error recovery, where an error-free state is substituted for the erroneous state; this substitution may take on three forms:
   - Backward recovery, where the erroneous state transformation consists of bringing the system back to a state already occupied prior to error occurrence; this involves the establishment of recovery points, which are points in time during the execution of a process for which the then current state may subsequently need to be restored
   - Forward recovery, where the erroneous state transformation consists of finding a new state, from which the system can operate (frequently in a degraded mode)
   - Compensation, where the erroneous state contains enough redundancy to enable its transformation into an error-free state

When backward or forward recovery are used, it is necessary that error detection precedes error recovery. Backward and forward recovery are not exclusive: backward recovery may be attempted first, then, if the error persists, forward recovery may be attempted. In forward recovery, it is necessary to perform error diagnosis, which can—in principle—be ignored in the case of backward recovery, provided that the mechanisms enabling the transformation of the erroneous state into an error-free state have not been affected (16).

The association into a component of its functional processing capability together with error detection mechanisms leads to the notion of a *self-checking component,* either in hardware (11,17,18) or in software (19,20); one of the important benefits of the self-checking component approach is the ability to give a clear definition of error confinement areas (7). When error compensation is performed in a system made up of self-checking components partitioned into classes executing the same tasks, then state transformation is nothing else than switching within a class from a failed component to a nonfailed one. On the other hand, compensation may be applied systematically, even in the absence of errors, thus providing *fault masking* (e.g., through a majority vote). However, this can at the same time correspond to a redundancy decrease that is not known. So, practical implementations of masking generally involve error detection, which may then be performed after the state transformation. As opposed to fault-masking, the implementation of error processing by error recovery after error detection has taken place is generally referred to as *error detection and recovery.*

The operational time overhead necessary for error processing is radically different according to the adopted error recovery form:

- In backward or forward recovery, the time overhead is longer upon error occurrence than before; also, in backward recovery, there may be a considerable overhead even in the absence of errors due to the need to create recovery points.
- In error compensation, the time overhead required by compensation is the same, or almost the same, whether errors are present or not (in either case, the time for updating the system status records adds to the time overhead).

In addition, the duration of error compensation is much shorter than the duration of backward or forward error recovery, due to the larger amount of (structural) redundancy. This remark is of high practical importance in that it often conditions the choice of the adopted fault tolerance strategy with respect to the user time granularity. It also introduces a relation between operational time overhead and structural redundancy. More generally, a redundant system always provides redundant behavior, incurring at least some operational time overhead; the time overhead may be small enough not to be perceived by the user, which means only that the service is not redundant; an extreme opposite form is “time redundancy” (redundant behavior obtained by repetition) which needs to be at least initialized by some structural redundancy. Roughly speaking, the more the structural redundancy, the less the time overhead incurred.

Fault Treatment. The first step in fault treatment is *fault diagnosis,* which consists of determining the cause(s) of error(s), in terms of both location and nature. Then come the *fault passivation* actions aimed at fulfilling the main purpose of fault treatment: preventing the fault(s) from being activated again. This is carried out by preventing the component(s) identified as being faulty from being invoked in further executions. If the system is no longer capable of delivering the same service as before, then *reconfiguration* may take place, which consists in modifying the system structure so that the nonfailed components can deliver an acceptable, but possibly degraded, service. Reconfiguration may involve some tasks being abandoned, or reassigning tasks among nonfailed components.

If it is estimated that error processing could directly remove the fault, or if its likelihood of recurring is low enough, then fault passivation need not be undertaken. As long as fault passivation is not undertaken, the fault is regarded as a *soft* fault; undertaking it implies that the fault is considered as *hard,* or *solid.* At first sight, the notions of soft and hard faults may seem to be respectively synonymous to the previously introduced notions of temporary and permanent faults. Indeed, tolerance of temporary faults does not necessitate fault treatment, since error recovery should in this case directly remove the effects of the fault, which has itself vanished, provided that a permanent fault has not been created in the propagation process. In fact, the notions of soft and hard faults are useful due to the following reasons:

- Distinguishing a permanent fault from a temporary fault is a difficult and complex task, since (1) a temporary fault vanishes after a certain amount of time, usually before fault diagnosis is undertaken, and (2) faults from different classes may lead to very similar errors; so, the notion of soft or hard fault in fact incorporates the subjectivity associated with these difficulties, including the fact that a fault may be declared as a soft fault when fault diagnosis is unsuccessful.
- The ability of those notions to incorporate subtleties of the modes of action of some transient faults; for instance, can it be said that the dormant internal fault resulting from the action of alpha particles (due to the residual ionization of circuit packages), or of heavy ions in space, on memory elements (in the broad sense of the term, including flip-flops) is a temporary fault? Such a dormant fault is however a soft fault.

Comments. Before turning to the detailed presentation of the methods and techniques implementing the various primitives of error processing (see the three subsequent sections of this article), we will provide additional definitions and general comments that we find useful for (a) a better understanding of these developments, and (b) eliciting the appropriate methods and techniques developed therein.

The classes of faults (physical, design, etc.) that can actually be tolerated depend on the fault hypotheses that are considered in the design process, and in particular, on the independence of redundancies with respect to the process of fault creation and activation. An example is provided by considering tolerance of physical faults and tolerance of design faults. A (widely used) method to attain fault tolerance is to perform multiple computations through multiple channels. When tolerance of physical faults is foreseen, the channels may be identical, based on the assumption that hardware components fail independently; such an approach is not suitable for the tolerance of design faults where the channels have to provide identical services through separate designs and implementations (13,21,22), that is, through *design diversity* (15).

An important aspect in the coordination of the activity of multiple components is that of preventing error propagation from affecting the operation of nonfailed components. This aspect becomes particularly important when a given component needs to communicate some private information to other components. Typical examples of such single-source information are local sensor data, the value of a local clock, the local view of the status of other components, and so on. The consequence of this need to communicate single-source information from one component to other components is that nonfailed components must reach an agreement as to how the information they obtain should be employed in a mutually consistent way. Specific attention has been devoted to this problem in the field of distributed systems.

Fault tolerance is a recursive concept: it is essential that the mechanisms aimed at implementing fault tolerance be protected against the faults that can affect them. Examples are voter replication, self-checking checkers (17), and "stable" memory for recovery programs and data (23).

Validation of Fault Tolerance

The validation process incorporates both fault removal and fault forecasting activities as identified in Fig. 1. The validation of fault-tolerant computing systems calls for the same activities as the validation of nonfault-tolerant systems, and central to these activities are verification and evaluation. A major difference, however, is that in addition to the functional inputs, the validation has to be carried out with respect to the specific inputs, that is, the faults that such systems are intended to handle. From the verification viewpoint, a particular form of testing can be identified, and that is *fault injection.* From the evaluation viewpoint, the main issue is the efficiency of the fault tolerance mechanisms, that is generally called *coverage* (24). The possibility of assessing the fault tolerance coverage by modeling alone is very limited due to the complexity of the mechanisms involved, encompassing error detection, error processing, and fault treatment. Thus, in this case also, fault injection is necessary.

Accordingly, fault injection is a central technique for the validation of fault-tolerant systems. Its significance has long been recognized, but only recently has it been the subject of work aimed at overcoming the ad hoc perception that was usually associated with it. This work can be classified according to the method of fault injection (25):

- Physical fault injection, where the faults are injected directly on the hardware components by means of physical or electrical alterations
- Informational fault injection, where the faults are injected by altering Boolean variables or memory contents

Several fault injection techniques and supporting tools have been developed (26). Most work on, or using, physical fault injection is based on injecting at the level of integrated circuit (IC) pins. This technique constitutes a simulation of the faults, or, more exactly, of the errors provoked by faults that can occur during system operation. It is, however, possible to be closer to reality for a specific class of faults that are of particular interest for the space environment: heavy-ion radiation. Sources of particles similar to heavy ions exist, although they are only able to inject into a single IC. Besides the representativity of the faults injected on the pins of the ICs, another important issue is accessibility. Clearly, this problem will not improve in the future when considering the current assembly techniques such as surface-mounted components. Accessibility problems can be solved by injecting at the level of the information being processed or stored, although at the expense of a greater deviation from real faults, and thus intensifying the error simulation aspect. This approach is also known as software-implemented fault injection (SWIFI).

These fault injection approaches are targeted at the system being validated, after it has been implemented, possibly as a prototype. A natural move is to be able to carry out the fault injection during the design of the system, using a simulation model of the system being designed (27,28).

As noted at the beginning of this subsection, testing of the fault tolerance mechanisms has long been the primary objective of ad hoc fault injection approaches (29). Meanwhile, most of the fault-injection tools previously cited aim rather at evaluating the efficiency of the fault-tolerance mechanisms.

Fault removal is obtained only as a by-product, when the evaluations reveal some deficiency in the fault-tolerance mechanisms. Although it has been shown that such a by-product is nevertheless of real interest (25), the problem of fault-injection testing specifically aimed at removing potential fault-tolerance deficiencies is still an open issue, in spite of some recent advances (30,31).

ERROR DETECTION

Error detection is based on component, time, information, or algorithmic redundancy (or a combination thereof). The most sophisticated way of performing error detection is to build error detection mechanisms into a component alongside its functional processing capabilities, thus leading to the notion of a self-checking component (16,19). The most usual forms of error detection are the following:

- Error detecting codes
- Duplexing and comparison
- Timing and execution checks
- Reasonableness checks
- Structural checks

The last three forms of error detection can be implemented by executable assertions in software. An assertion is a logical expression that performs a reasonableness check on the objects of a program and is evaluated on-line. The logical expression is considered as true if the state is judged to be correct; if not, an exception is raised.

Error Detecting Codes

Error detecting codes (18) are directed essentially toward errors induced by physical faults. Detection is based on redundancy in the information representation, either by adding control bits to the data, or by a representation of the data in a new form containing the redundancy. The first form of redundancy constitutes the so-called separable code class and the second corresponds to the nonseparable code class.

A fundamental concept of error detecting codes is the so-called Hamming distance. This distance between two binary words corresponds to the number of bits for which the two words differ. The *distanc*e of a code is the minimum Hamming distance between two valid code words. For the code to be able to detect e errors, the code distance must be greater than or equal to $e + 1$.

The level of redundancy used depends on the error assumption: single errors, unidirectional errors, or multiple errors.

Parity is the most common form of coding that allows single errors to be detected. Errors affecting a slice of b bits can be detected using b-adjacent codes. When the encoded data must be processed arithmetically (addition, multiplication), it may be convenient to use arithmetic codes. Such codes are preserved during arithmetic operations (a code C is preserved by an operation o if $A, B \in C$ implies that $A \; o \; B \in C$). Arithmetic codes can be classified as either nonseparable or separable codes.

Codes detecting unidirectional errors can also be classified as either separable codes (e.g., Berger's code) or nonseparable codes (e.g., K-out-of-N codes). Detection of multiple errors requires the use of a 1-out-of-2 code (a so-called two-rail code) that can be thought of as a form of duplication.

Duplexing and Comparison

Duplexing and comparison, despite its high redundancy overhead, is a widely used detection mechanism due to its simplicity.

Since few assumptions need to be made about the cause of an error, duplexing is a general technique. There are few other checks that can provide equivalent detection power. The basic assumption concerns the independence of redundancies with respect to the process of fault creation and activation. It is thus mandatory to ensure that:

- Either the faults are created and activated independently in the duplexed units
- Or, if the same fault provokes an error in both units, these errors are distinct

Thus, when tolerance of internal physical faults is foreseen, the channels may be identical, based on the assumption that hardware components fail independently. However, if external physical faults are to be accounted for, then common mode failures should be avoided by physically separating the units and/or by having them execute at different times.

However, when these assumptions no longer hold (which is the case when design faults in either hardware or software are accounted for), it is necessary that the units provide identical services through dissimilar designs and implementations, that is, through design diversity.

Timing and Execution Checks

Due to its very limited cost, timing checks by means of watchdog timers are the most widely used concurrent error detection mechanism. Although this technique covers a wide spectrum of faults, it is not easy to evaluate the coverage achieved. Watchdogs can be used in many situations, ranging from the detection of the failure of a peripheral device whose response time should be less than a maximal value ("timeout") to the monitoring of the activity of central processing units (CPUs). In the latter case, the watchdog is periodically reset. If the behavior of the CPU is altered such that the watchdog is not reset before it expires, then an exception is raised. Such an approach can be used to allow the CPU to escape from a blocking state or from an infinite loop.

One possible improvement of the detection efficiency provided by a watchdog is to verify, in addition, the flow of control of the program being executed by the CPU (32). This method, also known as signature analysis, relies on a compression scheme that produces a signature (usually the checksum of a series of instructions). The flow of control can then be verified by generating the signature when executing the program and comparing it with a reference value obtained when applying the same compression function to the object code of the program. Signature analysis can be implemented efficiently by hardware monitors or watchdog processors.

Execution flow control can also be applied at higher levels of abstraction. For example, Ref. 33 presents the case of the control of a communication protocol described as a Petri-net model. In this case, the parallel execution of the program and of the abstract model enables the consistency of the successive states of the protocol be verified.

A practical implementation that is worth mentioning is the SACEM processor (34) that combines:

- An arithmetic code, aimed at detecting data storage, transfer and processing errors
- A signature scheme, for detecting errors in the sequencing of the program and in the addressing of the data to be processed

Reasonableness Checks

Reasonableness checks only induce a very low additional cost compared to the cost of functional elements of the system. Many such checks can be implemented to detect errors arising from a wide spectrum of faults, but their coverage is usually rather limited. Reasonableness checks can be implemented by means of:

- Specific hardware, to detect value errors (illegal instruction, unavailable memory address) and access protection violations
- Specific software, to verify the conformity of the inputs or outputs of the system with invariants

Software-based controls can be incorporated in the operating system to be applicable to any application program (e.g., dynamic type control, verification of array indices, etc.) or specific to the application program (e.g., range of possible values, maximum variation with respect to results of a previous iteration, etc.).

Structural Checks

Checks can be applied to complex data structures in a computer system. They can focus on either the semantic or structural integrity of the data.

Semantic integrity checks consist in the verification of the consistency of the information contained in the data structure using reasonableness checks as described in the previous paragraph.

Structural integrity checks are particularly applicable to data structures whose elements are linked by pointers. Redundancy in these structures can be of three main forms:

1. Counts of the number of elements contained in the structure
2. Use of redundant pointers (double linking)
3. Addition of indicators regarding the types of elements in the structure

A valid modification of the structure requires the atomic modification of all the redundant elements. Error detection relies on the fact that the modification will not be atomic if the system behaves in an erroneous fashion. The theory developed in Ref. 35 extends the properties of the classical coding theory to this application domain. In particular, it states that the greater the number of changes necessary for an update, the greater is the detection power.

ERROR RECOVERY

Error recovery consists in substituting an error-free system state for an erroneous state. Three forms of error recovery can be identified, depending on the way the error-free state can be built. These three forms are backward recovery, forward recovery, and error compensation.

Backward Error Recovery

Backward error recovery (also called rollback) is by far the most popular form of error recovery. It consists in periodically saving the system state so as to be able, following detection of an error, to return the system to a previous state.

The saved states, called checkpoints, must be stored by means of a stable storage mechanism to protect the data from the effects of faults (23). Checkpointing can be supported by hardware or software mechanisms that automatically save the data modified between two checkpoints. If the detection coverage is not perfect, checkpoints may be contaminated by an error before it is detected. In this case, recovery will not be successful unless an error-free state can be provided. This means that several successive checkpoints must be preserved or the application structure must allow nested checkpoints, as in the case of recovery blocks (36).

Generally, the data that is saved during the generation of a checkpoint is not a snapshot of the whole state of the system but only the state of part of the system, usually a process: a global checkpoint of the state of a system is, therefore, made up of a set of partial checkpoints. Restoration of an error-free state requires rolling back to their last checkpoint at least all the processes that may have been directly contaminated by the error (for example, those run on the unit on which the error has been detected). Even then, a consistent system state may not be obtained since these processes may have interacted with others since their last checkpoint. Not only must these other processes be rolled back but they may also have interacted with others. A domino effect can occur whereby the failure of one process leads to a cascade of rollbacks to make sure that the system is returned to a global consistent state (37). Three approaches to checkpointing can be identified:

1. Checkpoints are created independently by each process (this is called *uncoordinated* or *asynchronous checkpointing*). When a failure occurs, a set of checkpoints must be found that represents a global consistent state. This approach is a dynamic technique that aims to minimize timing overheads during normal operation (that is, without errors) at the expense of a potentially large overhead when a global state is sought dynamically to perform the recovery. However, it entails two major drawbacks: (a) the amount of information saved may be quite large and (b) it might be necessary to roll all processes back to the initial state if no other global consistent state can be found (i.e., domino effect). The dynamic search for a global consistent state and the related domino effect risk can be avoided in the case of deterministic processes by logging messages on stable storage so that they can be replayed to a recovering process.
2. The creation of checkpoints can be preprogrammed so as to generate a set of checkpoints corresponding to a

global consistent state. There exists a very simple but fairly costly technique: if, when a process sends a message, it takes the checkpoint atomically with the message transmission, the most recent checkpoints always constitute a global consistent state. Another approach is to structure process interactions in conversations (22). In a conversation, processes can communicate freely between themselves but not with other processes external to a conversation. If processes all take a checkpoint when entering or leaving a conversation, recovery of one process will only propagate to other processes in the same conversation. The transactional approach provides another elegant way of managing checkpoints. A transaction is the execution of a program that accesses a set of shared data items (38). The executed program is designed to transform the data from an initial state where data are mutually consistent into another consistent state. The transaction must sometimes be aborted, as would be the case, for example if a current account debiting request were rejected for lack of sufficient funds. In this case, the data must be restored to their initial state. The latter must, therefore, be saved, thereby constituting a checkpoint. The means for restoring the initial state can be utilized not only at the request of the program but also in case of conflicts of access to the shared data detected by a concurrency control algorithm (to authorize the execution of transactions in parallel as if their executions were carried out in series) or if a fault-induced error is detected.

3. The establishment of checkpoints is dynamically coordinated so that sets of checkpoints represent global consistent states (this is called *coordinated* or *synchronous checkpointing*). In this approach, the domino effect problem can be transparently avoided for the programmer even if the processes are not deterministic. Each process possesses one or two checkpoints at each instant: a permanent checkpoint (constituting a global consistent state) and another, temporary checkpoint, that may be undone or transformed into a permanent checkpoint. The transformation of temporary checkpoints into permanent ones is coordinated by a two-phase commit protocol to ensure that all permanent checkpoints effectively constitute a global consistent state (39). Another dynamic approach, called *communication-induced checkpointing,* relies on control information piggy-backed onto application messages to indicate when a process receiving a message needs to take a forced checkpoint to ensure that each local checkpoint belongs to at least one global consistent checkpoint (40,41).

Backward recovery techniques do have some drawbacks. First, rollback is usually incompatible with applications with hard real-time deadlines. Second, the size of the recovery points and the timing overhead needed for their establishment often impose structural constraints that must be taken into account during application development and require dedicated support from the operating system. Generally, this precludes the use of any general-purpose operating system such as UNIX and software packages that have not been developed specifically for the architecture considered. Note, however,

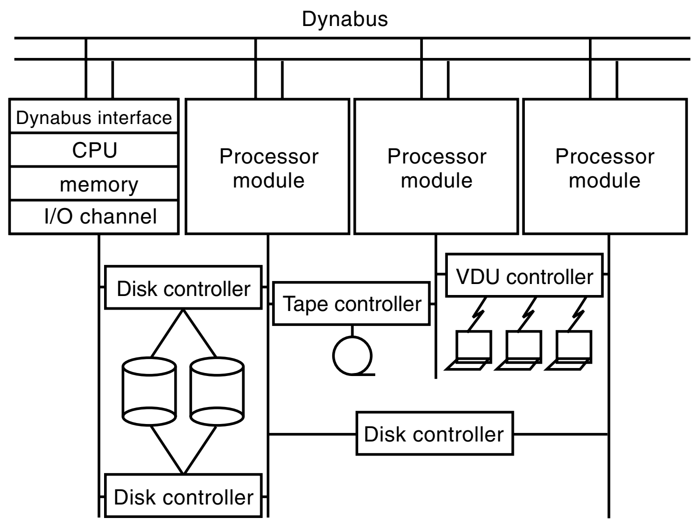

Figure 3. Architecture of the Tandem NonStop fault-tolerant computer.

that portable UNIX single process checkpointing systems have been developed for "well-behaved" programs, that is, programs that, among other restrictions, do not use interprocess communication and only access files sequentially (42).

Example: Tandem NonStop Computers. The NonStop systems produced by Tandem (43) are designed to tolerate a single hardware fault (Fig. 3). The CPUs and input/output controllers are *fail-fast* in that they are equipped with error detection mechanisms that block the unit in which an error is detected. Error detection is mainly based on parity checking, coding, and reasonableness tests in software and firmware. In certain cases, self-checking circuits are used. These units are also designed to limit error propagation: for example, the Dynabus and the I/O controllers are built in such a way that no hardware fault can block the two buses to which they are connected. There always exists a path to access a dual access peripheral even if a processor, bus, or controller has failed.

Disks are organized as mirror disks, that is, for each disk there exists an identical copy. Each write is transmitted to both disks, reading being done on one disk only to optimize access time. In case of a read error on a disk, the read order is repeated on the other disk. This standby concept is generalized: in case of failure of an operation on a processor, a bus, or a controller, there exists a standby unit capable of carrying out the same operation.

The same principle is applied to the software, which is organized around pairs of processes: to each active process corresponds a standby process running on another processor, and the active process regularly sends checkpoints to the standby process. These checkpoints are either copies of the active process state, or deviations relative to the previous state, or even a transform function of the state. In normal operation, the standby process only updates its state according to the checkpoints it receives. If the processor on which the active process is running fails, the other processors will detect it through an absence of "I'm alive" messages (transmitted every two seconds by any processor operating normally). The operating system of the processor on which the standby process is executed activates this process, which takes over from the last checkpoint received.

This organization into process pairs calls for a specific design of the operating system and of the application software, especially for the generation of checkpoints. Applications are facilitated by libraries of elementary functions but the incompatibility with standard products leads to a significant cost increase.

Another drawback of this architecture is linked to the fact that the error detection mechanisms have imperfect coverage, so an error may propagate prior to the blocking of the failing unit. However, based on the information published by Tandem, it appears that the global failures of their systems are mostly due to software design faults and interaction faults, most of them being Heisenbugs (44), that is, not diagnosed because difficult to reproduce.

Forward Error Recovery

Forward error recovery constitutes an alternative or complementary approach to rollback—following detection of an error, and a possible attempt at backward recovery, forward recovery consists in searching for a new state acceptable for the system from which it will be able to resume operation (possibly in a degraded mode).

One simple forward recovery approach is to reinitialize the system and acquire a new operating context from the environment (e.g., rereading the sensors in a real-time control system).

Another approach is that of *exception processing,* for which primitives exist in many modern programming languages. In this case, the application programs are designed to take into account error signals (from error detection mechanisms) and switch from normal processing to exceptional, generally degraded, processing. In fail-safe systems, the exceptional processing is reduced to the most vital tasks. In extreme cases, these vital tasks bring the system to a stable safe state and then halt the processor. An example is stopping a train: an immobile train is in a safe state if passengers can leave the train (e.g., in case of fire), and if the stopping of the train is signaled early enough to other trains on the same track.

Note that the implementation of forward recovery is always specific to a given application. Unlike backward recovery or compensation-based recovery techniques, forward recovery cannot be used as a basic mechanism of a general-purpose fault-tolerant architecture.

Compensation-Based Error Recovery

Error compensation requires sufficient redundancy in the system state so that, despite errors, it can be transformed into an error-free state. A typical example is given by the error-correcting codes presented later.

Error compensation can be launched following error detection (detection and compensation), or can be systematic (masking). Even in the latter case, it is useful to report errors to initiate fault treatment. Indeed, if no fault treatment is done, redundancy may be degraded without the users being aware of it, thereby leading to a failure when another fault is activated.

Using compensation, it is no longer necessary to re-execute part of the application (backward recovery) or run a dedicated procedure (forward recovery) to continue operation. This type of recovery is, therefore, fairly transparent to the application: there is no need to restructure the application to account for error processing. This can allow the use of standard operating systems and software packages.

Error Detection and Compensation. A typical example of error detection and compensation consists in using self-checking components executing the same processing in active redundancy; in case of failure of one of them, it is disconnected and processing can go on without disturbing the others. In this case, compensation is limited to a possible switch from one component to another. This is, for instance, the basis of the architecture of the Stratus S/32 or IBM System/88.

Example: The Airbus 320 Flight Control System. Recent passenger aircraft, such as the Airbus 320/330/340 family and the Boeing 777, include computers in the main flight control loop to improve overall aircraft safety (through stability augmentation, flight envelope monitoring, windshear protection, etc.) and to reduce pilot fatigue. Of course, these increases in aircraft safety must not be annihilated by new risks introduced by the computing technology itself. For this reason, the flight control systems of these aircraft are designed to be fault tolerant.

Fault tolerance in the flight control system of the Airbus 320/330/340 family is based on the error detection and compensation technique (45). Each flight control computer is designed to be self-checking, with respect to both physical faults and design faults, to form a fail-safe subsystem. Each computer consists of two lanes supporting functionally-equivalent but diversely-designed programs (Fig. 4). Both lanes receive the same inputs, compute the corresponding outputs, and check that the other lane agrees. Only the control lane drives the physical outputs. Any divergence in the results of each lane causes the physical output to be isolated.

Each flight control axis of the aircraft can be controlled from several such self-checking computers. The complete set of computers for each axis processes sensor data and executes the control loop functions. However, at any given instant, only one computer in the set (the primary) is in charge of physically controlling the actuators. This computer sends periodic "I'm alive" messages to the other computers in the set so that they may detect when it fails. Should the primary fail, it will do so in a safe way (thanks to the built-in self-checking) without sending erroneous actuator orders. According to a predetermined order, one of the other computers in the set then becomes the new primary and can immediately close the control loop without any noticeable jerk on the controlled surface.

The design diversity principle is also applied at the system level. The set of computers controlling the pitch axis (Fig. 5) is composed of four self-checking computers: two Elevator and Aileron Computers (ELACs) and two Spoiler and Elevator Computers (SECs), which are based on different processors and built by different manufacturers. Given that each computer type supports two different programs, there are overall four different pitch control programs.

There is also considerable functional redundancy between the flight control surfaces themselves so it is possible to survive a complete loss of all computer control of some surfaces, as long as the failed computers fail safely. Furthermore, if all computers should fail, there is still a (limited) manual backup.

Fault Masking. Unlike the previous technique, *masking* is an error compensation technique in which compensation is

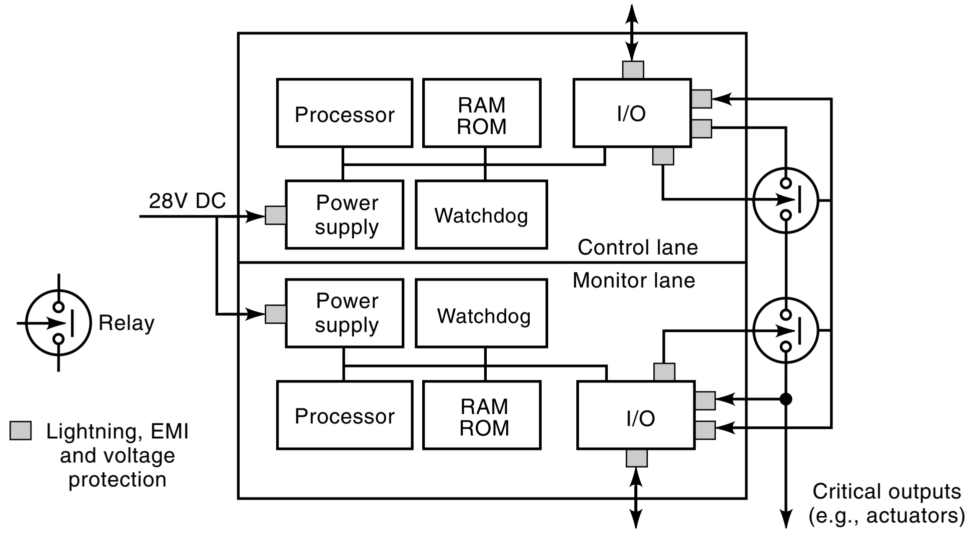

Figure 4. Self-checking Airbus 320 flight control computer based on diversely programmed control and monitor lanes.

carried out systematically, whether an error is detected or not. A typical example is that of *majority voting:* processing steps are run by three (or more) identical components whose outputs are voted. The majority results are transmitted, and minority results (supposedly erroneous) are discarded (Fig. 6).

As voting is systematically applied, the computation and, therefore, the execution time, are identical whether or not there exists an error. This is what differentiates masking from the detection and compensation techniques.

The voting algorithm can be simple if the copies are identical and synchronous and if computation is deterministic in the absence of errors. If these assumptions cannot be guaranteed, one has to consider that the copies are diverse and a more or less complex decisional algorithm will be needed, depending mainly on the type of information for which voting is needed (46).

Example: Tandem Integrity S2. Announced in 1989 by Tandem, the Integrity S2 system aims, like the NonStop system, to tolerate a single hardware fault with the additional requirement of supporting commercial off-the-shelf application software through the use of an operating system that is fully compatible with UNIX. The architecture is shown in Fig. 7.

This architecture features a triplex structure for the processors and their local memories and a duplex structure for the voters (which are self-checking), the global memories, and the input-output buses and processors. Local memories contain a copy of the UNIX kernel (in a protected memory), and program and data application zones. The global memories also contain application zones and control and buffer zones for input/output. Under normal operating conditions, the three local memories have identical contents. The same is true for the two global memories. Each processor has its own

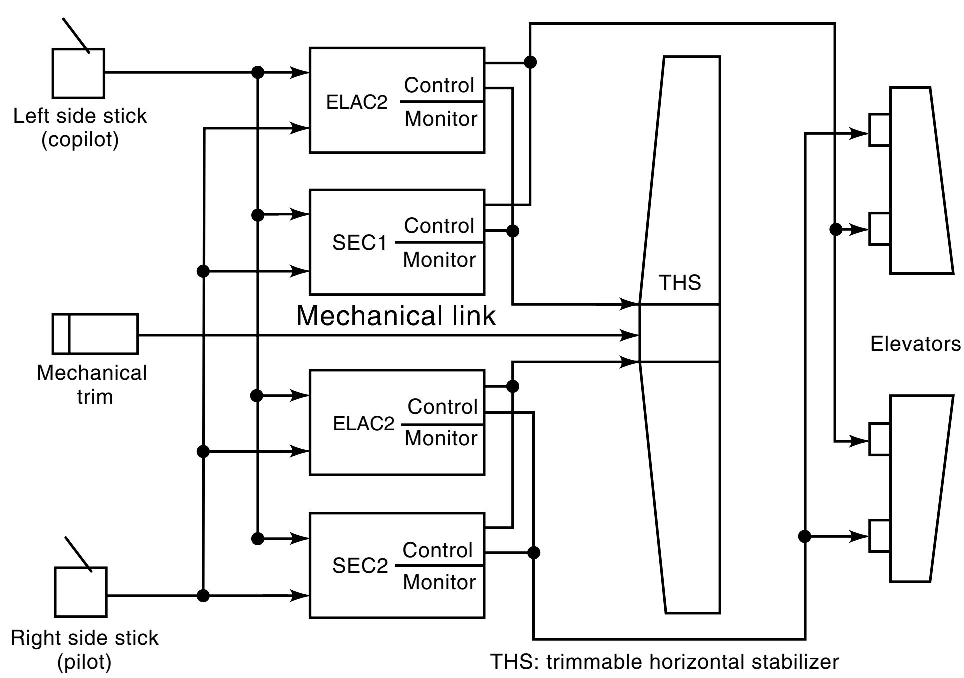

Figure 5. The Airbus 320 pitch control nominally uses two diverse pairs of diversely programmed self-checking computers.

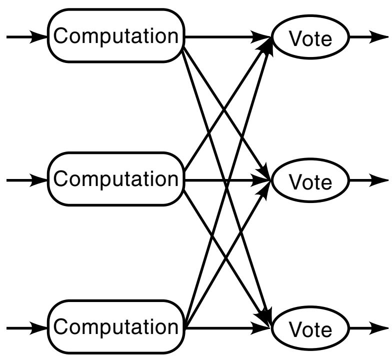

Figure 6. Principle of majority voting.

clock. Their computations are synchronized when accessing the global memory. Each such access gives rise to a majority vote.

In case of inequality, an error is reported and computation is continued without interruption on the majority processors. A self-checking program is then started on the minority processor to determine whether the error was created by a nonreproducible soft fault. If that is the case, the processor can be reinserted. Otherwise, it has to be replaced.

Input/output is based on the same technique as in the Tandem NonStop system: duplicated buses, self-checking input/output processors (IOPs), mirror disks. Nevertheless, a specific feature is worth noting, that is, bus interface modules (BIMs) serve to interface the dedicated duplicated buses with standard VME buses, so that peripherals and controllers from other suppliers can be used. If there is an IOP or associated bus failure, the BIM switches control of the VME bus over to the other IOP.

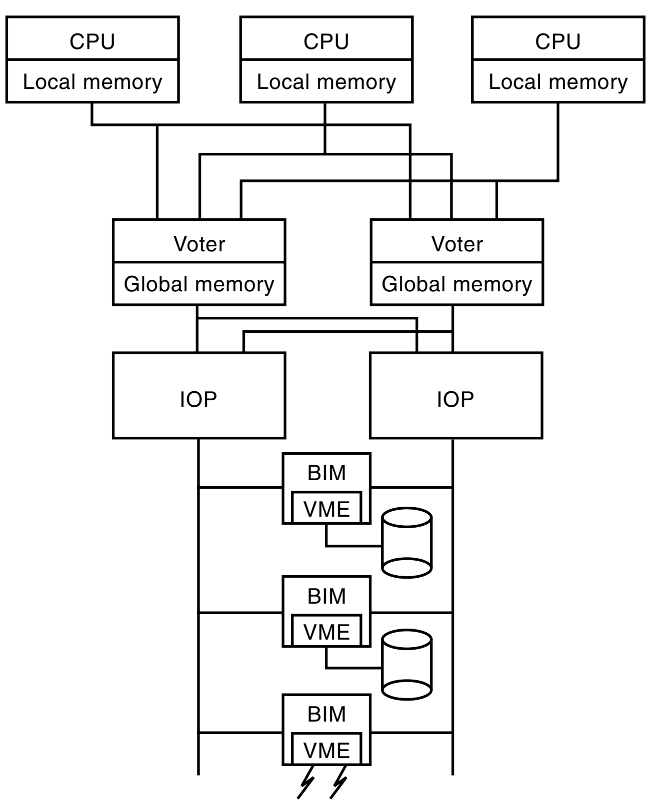

Figure 7. Architecture of the Tandem Integrity S2 fault-tolerant computer.

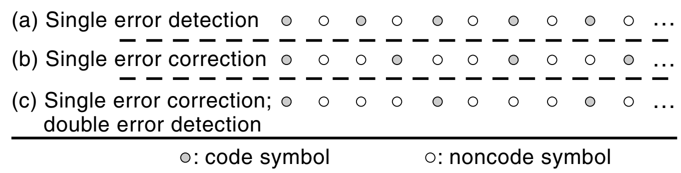

Figure 8. The distance between code symbols determines a code's ability to detect and correct errors.

Error Correcting Codes. The information encoding principle introduced in the section on error detection can also be employed to construct codes capable of correcting errors (47). Error correction requires a larger Hamming distance between the symbols (words) of the code. To correct a single error, the code distance must be greater than or equal to 3 (instead of 2 for detection). More generally, to correct e errors a code of distance $2 \times e + 1$ is required. Note that a code of distance d ($d \geq 3$) can be employed either to detect $d - 1$ errors, or to correct $\lfloor (d - 1)/2 \rfloor$ errors. Thus, a code of distance d can correct ec errors and detect ed additional errors if and only if: $d \geq 2 \times ec + ed + 1$.

The redundancy principle used to detect or correct an error is depicted in the diagram of Fig. 8.

The Hamming code is the most popular single error correction code. It is obtained by adding one or more bits to selectively control the parity of certain data bits. These control bits are used to build a syndrome that allows an unambiguous error diagnosis. For the correction to take place, the binary combinations of the syndrome must allow identification of the various combinations in which errors are absent or present on any bits of the word (including the control bits). If k stands for the number of information bits, c refers to the number of control bits (and equally of the syndrome) and n the total number of bits of the code word ($n = k + c$), then we must have: $2^c \geq n + 1$. For example, for a 16-bit encoded data word, 5 control bits are needed. The overhead in terms of the number of bits is about 30%. This cost becomes less than 15% for 64-bit data words.

The extension of the Hamming code to the systematic and simultaneous detection of double errors [see Fig. 8(c), for example] is simply obtained by adding a single parity bit covering the n bits.

Other more powerful correcting codes have been developed. Cyclic codes are particularly suited to serially transmitted data. These codes are interesting because the encoding and decoding operations can be performed easily and economically by using shift registers with loops. Additionally, these codes lend themselves well to the detection of error bursts (errors affecting several adjacent bits). The most popular class of binary cyclic codes are BCH codes (Bose, Chauduri, and Hocquenghem). These are a generalization of the Hamming code to multiple error correction. Among higher order codes (that is, covering nonbinary symbols), the most important class corresponds to the RS codes (Reed–Solomon). These are a direct extension of binary codes that allow correction of error bursts.

An efficient way to define a powerful code is to combine two (or more) codes. Such *product* codes allow interesting properties to be obtained at a low cost. For example, a single error correcting code can be obtained by using a bidimensional parity pattern. In addition to the conventional single parity bit associated with each word (row) of the matrix representing the memory space, a parity bit is associated with each column (including the row parity bit column). The matrix is thus extended by a horizontal parity bit word and a vertical parity bit word. An error affecting one bit can easily be detected and localized (and therefore, corrected) because it affects the parity in the corresponding row and column. This technique is both efficient and inexpensive but fails to correct multiple errors.

DISTRIBUTED SYSTEMS

A distributed system can be defined as a set of computing nodes, interconnected by a communication network, that cooperate to carry out some common work. The nodes can typically be considered as independent from the viewpoint of failures, so distribution can be a useful framework for providing fault tolerance. However, distribution of system state and other dependencies between cooperating tasks also mean that a distributed service can easily be disrupted if any of the nodes involved should fail. With the added potential complication of unreliable communication, distributed services often need to be made fault-tolerant if they are to be useful.

These two opposing facets of the relationship between fault-tolerance and distribution are strong motivations for distributed fault-tolerance techniques. An important characteristic of these techniques is that error processing and fault treatment are implemented mainly by software using distributed, message-passing algorithms.

Models and Assumptions

Fault-tolerant distributed algorithms have been devised according to several distributed system models that embody assumptions about faults and the timing of interprocess communication (48) (we use the term *process* in a very general sense, to designate any communicating entity or fault containment domain, be it a UNIX process, an object, a processor, etc.).

Fault Models. In distributed systems, a fault model is defined in terms of process and communication failures. It is common to admit that communication failures can only result in lost or delayed messages, since checksums can be used to detect and discard garbled messages. However, duplicated or disordered messages are also included in some models.

For processes, the most commonly assumed failures are (in increasing order of generality): stopping failures or crashes, omission failures, timing failures, and arbitrary failures. In the latter case, no restrictive assumption is made. An arbitrarily faulty process might even send contradictory messages to different destinations (a so-called *Byzantine* failure).

Some fault models also include assumptions about how a failed process may be restarted. In particular, a crash failure assumption is often accompanied by an assumption that some local storage is stable in that its contents can survive the failure.

Timing Models. The simplest timing model to reason about is the synchronous or *bounded time* model. In this model, any message sent from one nonfaulty process to another is received and processed at the destination process within a bounded time. In practice, to bound the time for message transmission and processing, it is necessary: (1) to use hard real-time scheduling and flow control techniques, and (2) to assume an upper bound on the number of failures that can occur per unit of time. This is a very powerful model since it is possible to use time-outs to unambiguously detect whether remote processes have crashed or are late. It is an appropriate model for critical applications that require guaranteed real-time progress, even in the presence of faults. However, the required assumptions must be justified through an appropriate design of the underlying networks and operating systems.

At the opposite extreme, the asynchronous or *time-free* model places no bounds at all on message transmission and processing delays. A message sent by a nonfaulty process to another, through a nonfaulty link, will eventually be received and processed, but with no guarantee when. Algorithms designed according to this model are attractive since they are independent of networks and operating systems, and are thus general and portable. Unfortunately, some very basic problems in fault-tolerance cannot be solved when this model is adopted (49).

In many practical systems, time-outs are used to empirically detect whether remote processes have crashed, even if the underlying assumptions of the synchronous model are not justified. It may just be the case that the distributed application is not very critical, so the occasional lack of fault-tolerance has no dire consequences. Alternatively, time-outs are over-dimensioned to the extent that the probability of false detection is considered negligible.

Much recent research has been devoted to defining models that are intermediate between the asynchronous and synchronous models. One promising approach is the timed asynchronous model, which assumes that noncrashed processes have local clocks with bounded drift. By using these clocks to timestamp messages, it is possible to compute worst-case bounds on the currently achieved message transfer delays. Periods of operation in which synchronous behavior cannot be guaranteed can thus be flagged as such. This allows distributed algorithms to be designed that carry out useful work whenever the system behaves synchronously, and that switch to a well-defined safe mode of operation whenever failures occur too frequently. This model is therefore particularly well suited for implementing fail-safe distributed systems (50).

Partitioning. A set of processes is partitioned if it is divided into subsets that cannot communicate with each other. Partitioning may occur due to normal operations, such as in mobile computing, or due to failures of processes or interprocess communication. Performance failures due to overload situations can cause ephemeral partitions that are difficult to distinguish from physical partitioning.

Partitioning is a very real concern and a common event in wide area networks (WANs). Certain distributed fault-tolerance techniques are aimed at allowing components of a partition to continue some form of degraded operation until the components can remerge. Note that partitioned operation is excluded by principle in the synchronous and asynchronous models. The former forbids partitioning, whereas the latter assumes that it will eventually disappear. Partitioning is however naturally included in the timed asynchronous model as periods of nonsynchronous operation.

Consistency

Programming distributed systems is notoriously difficult, even without faults. This is essentially because the "state" of the system is distributed across all its processes and, since communication cannot be instantaneous, this state cannot be viewed consistently by any single process. We consider here some useful consistency techniques that can greatly simplify the programmer's task.

Global Time. One of the characteristics of a distributed system is that processors do not have access to a common physical clock. This complicates the issues of coordination and event-ordering. Consequently, one of the most basic consistency abstractions is some notion of global time. At least two sorts of global time can be considered: physical time and logical time.

Global physical time can be approximated by synchronizing distributed physical clocks. Clock synchronization can be done mutually (internal synchronization) or with respect to some authoritative time reference (external synchronization). For internal synchronization, typically each clock periodically reads the values of remote clocks, computes a correction function (e.g., a fault-tolerant average) and applies it locally. External synchronization can be achieved by periodically polling a time server, perhaps itself implemented by a fault-tolerant set of internally synchronized clocks or using a global positioning system (GPS) receiver.

The precision to which clocks can be synchronized depends mainly on the uncertainty in the time it takes to read a remote clock. The synchronous timing model must be adhered to for there to be a deterministic bound on the offsets between correct clocks. The timed asynchronous model, while it cannot achieve a deterministic bound, does allow a very high precision to be achieved using probabilistic synchronization. An example of such an approach is the Internet network time protocol, that can achieve typical offsets of less than a few tens of milliseconds (51).

Since it is not possible to perfectly synchronize clocks, physical time cannot be used to order events that occur less than a clock offset apart. Logical clocks, however, can be used to causally order events according to a "happens-before" relationship even in a system that has no notion of physical time. Logical clocks are implemented by counters at each process. These are incremented whenever relevant local events occur or when messages are received from other processes, carrying piggy-backed values of remote logical clocks. The most elaborate logical clock system maintains a vector of counters at each process, with an element in the vector for every process in the system. Such vector timestamps have found several practical uses (52).

Consensus. The consensus problem is a fundamental issue in fault-tolerant distributed computing (53). In its most basic form, all processes in a set must make a binary decision. Each process has its own initial value (i.e., opinion on what the decision should be). The problem statement requires that all nonfaulty processes finally make the same decision. Furthermore, if all processes had the same initial value, then the final decision should be that value. An equivalent *agreement* problem can be coined for choosing a value among more than two possible values. Agreement in the presence of arbitrary process faults is called *Byzantine agreement.* Agreement on a vector of initial values is called *interactive consistency.*

Solving consensus is necessary if nonfaulty processes are to make consistent decisions. Unfortunately, it has proven impossible to achieve consensus deterministically when messages can be lost (unreliable communication) or when the time needed for them to reach their destination cannot be bounded in advance (asynchronous timing model). Consensus can however be achieved deterministically with a synchronous timing model. Two other important results are that, in the presence of k faulty processes, $k + 1$ rounds of information exchange are needed and that there must be a total of at least $3k + 1$ processes if arbitrary failures can occur.

Recent theoretical work is centered on the definition of models between the fully asynchronous and synchronous extremes and seeks to define the minimum amount of restrictive assumptions that need to be added to the asynchronous model for consensus to become achievable.

Group Communication. Group communication services facilitate communication with and among sets of processes and are thus a useful abstraction for implementing replicated, fault-tolerant services. Group communication is essentially concerned with three issues: (1) how to select which destinations should accept messages, (2) how to route messages to those destinations, and (3) how to provide guarantees about message acceptance and message ordering. There are many different protocols in the literature, with almost as many different terminologies as authors.

Protocols that send data to all possible destinations are called *broadcast* protocols; protocols that designate a subset of possible destinations are called *multicast* protocols. A *membership* service is used to dynamically manage multicast groups. Membership services typically allow processes to join and leave groups dynamically, either voluntarily, or due to failures or network partitioning.

The ability to route messages to different destinations has long been a feature of local area networks. With the multicast backbone (MBone), it is now a reality on the Internet. However, routing messages to multiple destinations only gives the latter the possibility to accept them, but no guarantees that they will do so consistently.

A broadcast protocol that guarantees that all destinations agree to accept the same messages is called a *reliable broadcast* (54). A reliable broadcast that also guarantees that destinations accept messages in the same order is called an *atomic broadcast.* Some protocols also satisfy other ordering constraints, such as FIFO (first in, first out) and causal ordering. In general, reliable and atomic broadcast require the same conditions for solvability as the consensus problem. In particular, neither is achievable with the totally asynchronous timing model.

Tolerance Techniques

As discussed in the introduction to this section, fault-tolerance can be either a necessary evil of distribution or one of its very purposes. In the first case, some form of fault-tolerance is required to minimize the negative impact of a failed process or node on the availability of a distributed service. In its simplest form, this can be just a local recovery of the failed node. However, continuity of service in the presence of failed nodes requires replication of processes and/or data on multiple nodes.

Here, we revisit in a distributed setting some of the techniques described in the section on error recovery.

Local Recovery. The failure of a node hosting an important server can have an important negative impact on numerous clients. It is important in such a setting to be able to restart the failed server as quickly as possible. Two features can be built into the design of the server to facilitate this. First, if server operations are idempotent, clients can simply repeat requests for which they received no reply. Second, if a server process is stateless, it can restart after failure and resume operation without needing to restore its state or that of its clients. Also, a stateless server is not affected by the failure of any of its clients. This strategy has been used with success in Sun's network file system (NFS).

If a process is "stateful" rather than stateless, stable storage is required to allow local checkpoints of the process state to survive failures. Stable storage can be implemented using local nonvolatile memory, for example, a disk. A process can recover autonomously from a local checkpoint only if it has not interacted with other processes since taking the checkpoint or if it can replay those interactions (e.g., from a log on stable storage). If that is not the case, distributed recovery is necessary.

Distributed Recovery. Distributed recovery occurs when the recovery of one process requires remote processes also to undergo recovery. Processes must rollback to a set of checkpoints that together constitute a consistent global state. A domino effect (cascading rollback) occurs if such a consistent set of checkpoints does not exist. It is therefore better to coordinate the taking of checkpoints to avoid this problem (see section titled "Backward Error Recovery").

Replicated Processes. A fault-tolerant service can be implemented by coordinating a group of processes replicated on different nodes. The idea is to manage the group of processes so as to mask failures of some members of the group. We consider three different strategies here: passive, active, and semi-active replication (55).

With *passive replication,* input messages are processed by one replica (the primary), which updates its internal state and sends output messages. The other replicas (the standby replicas) do not process input messages; however, their internal state must be regularly updated by checkpoints sent by the primary. If the primary should crash, one of the standby replicas is elected to take its place. Passive replication is particularly well suited to stateless processes, since the absence of internal state removes the very need for checkpointing. Note that this technique can be viewed as a distributed implementation of the local recovery technique discussed previously.

Active replication is a technique in which input messages are atomically multicasted to all replicas, which then process them and update their internal states. All replicas produce output messages. Effective output messages are chosen from these by a decision function that depends on the process fault assumption. For crash failures, the decision function could be to take the first available output. This technique is also capable of tolerating arbitrary failures, using a majority vote decision function.

Semiactive replication is similar to active replication in that all replicas receive and can process input messages. However, like passive replication, the processing of messages is asymmetric in that one replica (the leader) assumes responsibility for certain decisions (e.g., concerning message acceptance or process preemption). The leader can enforce its choice on the other replicas (the followers) without resorting to a consensus protocol. Optionally, the leader may take sole responsibility for sending output messages. Although primarily aimed at crash failures, this technique can, under certain conditions, be extended to arbitrary failures.

With both active and semiactive replication, recovery of failed group members (or creation of new ones) implies initialization of their internal state by copying it across from the current group members. This operation is basically the same as the checkpointing operation of passively replicated stateful processes.

Replicated Data. From a data-oriented viewpoint, replication serves to improve both availability of data items and performance of read operations. First, a replicated data item can be accessed even if some of its replicas are on failed or inaccessible nodes. Second, it is usually faster to read a local replica than a remote one. However, write operations on replicated data can be slow, since they ultimately involve all replicas.

Data replica management protocols are called pessimistic or optimistic according to whether or not they guarantee one-copy equivalence, that is, that users perceive the replicated data item as if only one copy existed (56).

A pessimistic protocol guarantees one-copy equivalence by ensuring mutual exclusion between write operations, and between write and read operations. The simplest such protocol is the read-one write-all protocol: a user (process) can read any replica, but must carry out writes on all of them. This technique gives excellent read performance, but very poor write performance. Moreover, writes are blocked if any replica should become inaccessible. Quorum protocols generalize this idea and allow improved write performance at the expense of always having to access more than one replica for read operations. Other pessimistic replica management protocols include the primary-copy and the virtual-partition protocols.

Optimistic protocols sacrifice consistency to improve availability. These protocols authorize write operations on replicas that are in different components of a partitioned network. The available-copies protocol is an optimistic variant of the read-one write-all protocol: writes are performed only on the copies that are currently accessible. When partitioning ceases, any conflicts resulting from write operations carried out in different components must be detected and resolved. Conflict resolution depends on the semantics of the data, so it is usually application-specific.

In distributed transaction systems, replica management is integrated with concurrency control, and the notion of one-copy equivalence is refined into that of one-copy serializability (38).

FAULT-TOLERANT SYSTEM DEVELOPMENT

In the field of safety-critical system development, a number of standards have been issued in the last decade that address the issue of fault-tolerant computing. Such standards are useful, but must be defined and applied with care. In particular, earlier standards were too directive in how development activities should be done. This led people to provide a scrupulous step-by-step compliance, while forgetting the actual objectives of the standards (to upgrade the overall system dependability).

Current standards, such as IEC 1508 (57), DO178B (58), and ECSS (59), now leave more freedom to the developers to choose their own methods and tools. They do not impose a particular lifecycle (i.e., how to build the system), but only give the objectives that must be satisfied (i.e., what must be achieved).

This section describes how activities related to dependability, and especially fault-tolerance, are integrated into the key phases of the development of critical systems. We first show how the four basic means for dependability permeate all development phases. Taking the opposite viewpoint, we then detail the system development phases and their contribution to the building of the overall system dependability.

Dependability Activities within the Lifecycle

The four basic means for dependability (see "Basic Definitions") are implicitly present as activities in every phase of system development, and are used iteratively throughout the whole lifecycle (60).

Fault-Prevention Activities. Fault-prevention activities are all those activities that enforce the system to be correctly developed, thus preventing faults from occurring. The concerned development activities are:

- Choice of methods, formalisms, and languages. These choices cover all system development activities, and some of them may be imposed by standards.
- Project management activities. A good organization of the whole project reduces the potential of creating accidental faults due to misunderstandings between people. Furthermore, risk-management activities allow some faults to be avoided by evaluating risks and then taking the appropriate risk-reduction actions.

Fault-Tolerance Activities. Fault-prevention and fault-removal activities do not have a perfect coverage. So there may be residual design or implementation faults. Also, of course, faults can occur during system operation. The very aim of fault-tolerance is to allow the system to provide satisfactory service despite faults. The following development activities can be identified:

- Study of system behavior in the presence of faults. This activity is aimed at articulating the fault hypotheses under which the system will be developed.
- System partitioning into fault independence regions and error containment regions. This activity uses as input the fault hypotheses.
- Choice of the overall fault-tolerance strategy. This activity defines the error-processing (detection, recovery) and fault-treatment schemes.

Fault-Removal Activities. Fault-removal activities are aimed at improving system dependability by removing the faults (accidentally) introduced during development. They include:

- Verification, aiming at revealing faults by detecting errors. The verification activities may involve very different approaches, from tests to reviews, inspections, or even formal verification.
- Diagnosis, which consists in effectively identifying the faults causing the errors detected by verification.
- Correction, leading to the actual removal of the faults. Since this correction modifies the system, nonregression testing must then be done (the whole fault removal process is recursive).

Fault-Forecasting Activities. Fault-forecasting activities allow the presence of faults and the severity of their consequences to be anticipated and estimated. They contribute to the following system development activities:

- Definition of the system requirements in terms of dependability attributes.
- Allocation of these requirements onto the building blocks of the system.
- Evaluation of the presence of faults and of their possible consequences. Different methods, like FMECA (failure modes, effects and criticality analysis), FTA (fault tree analysis) and Markov models, are available to demonstrate the product's ability to meet the apportioned dependability objectives (in terms of reliability and/or availability).

These forecasting activities must take into account various system characteristics, like detailed mission definition, operating and environmental conditions, system configuration and fault-tolerance mechanisms (optimized through FMECA and risk analyses), and the values of parameters of system dependability models (failure rates, . . .). For physical faults, the latter can be extracted from reference handbooks [e.g., MIL HDBK 217 (61)]. For design faults, however, there is no equivalent to the MIL handbook. In this case, parameter values need to be obtained by statistical testing, or by applying reliability growth models to collected failure data (62).

System Development Phases

Industrial projects aimed at developing fault-tolerant systems involve a client and a system supplier. The overall goal of the system supplier is to provide to the client in due time and cost a system that satisfies his needs. To this end, the system supplier carries out a number of activities, which may be distributed among three broad categories:

1. Project management, which includes all the activities related to the overall organization of the project (planning, identification of tasks, attribution of responsibilities, management of cost and schedules, risk management)

2. System development, which includes all the activities that participate directly in the creation of the system (requirements, design, production, integration, verification, validation)
3. Product assurance, which includes all the quality assurance activities of the project

The development process for a fault-tolerant system is not very different in nature from the development of a less demanding system. In fact, the main particularity is that the final product delivered to the client must demonstrate a very high level of dependability. This implies that procedures need to be more strictly defined and adhered to. In particular, the risk-management activity becomes an essential part of project management, and the system-development activities are carried out according to more stringent methods and rules.

The system-development activities have to be organized to manage the complexity of large industrial projects in a way that allows the dependability of the final product to match the client's needs. This organization, also known as the system lifecycle, may vary from one project to another, but generally includes the following phases:

Requirements definition and analysis
Design
Production and verification
Integration and validation

Depending on the project size, these phases may be performed recursively, at different levels of decomposition of the system. They collectively participate in the construction of the dependability of the final system.

Requirements Definition and Analysis. The requirements are defined by the system supplier according to the client's needs. They constitute the agreed basis on which the system is to be built, and hence are of particular importance.

The requirements are stated at the system level, and then iteratively refined by taking into account the progressive decomposition of the system. In particular, the ever-increasing complexity of components (both hardware and software) has an impact on the way the dependability requirements are stated. Indeed, it no longer possible to assume a fault-free design as it was previously, when safety-critical systems were implemented using simple hardware components and little or no software. In those systems, only physical faults were considered. Today, especially in systems designed to tolerate physical faults, the majority of observed errors are due to residual design faults.

So, in the field of fault-tolerant computing, the functional requirements are completed by requirements concerning the dependability attributes of the final system. These dependability-related requirements cover:

- The necessary trade-offs between availability objectives (provide a continuous service) and the safety objectives (put the system in a safe state). In particular, the maximum service interruption and/or the safe/unsafe system states must be defined.
- The number of faults to be tolerated, and their impact on the system service. This requirement is often stated in terms of FO/. . ./FS, meaning that the system must remain operational after the first fault(s) (fail operational), and then put into a safe state (fail safe).
- The definition of the possible degraded modes.
- The ability of the system to be verified and possibly certified.

Furthermore, each function must be analyzed regarding its possible failure modes. For each failure mode, the severity (linked to the consequences of the failure at system level) and the probability of occurrence must be evaluated, thus feeding the risk-management process (which is then able to trigger the appropriate risk-reduction actions if necessary).

The requirements concerning the use of particular methods and tools are also impacted when a fault-tolerant system is to be built. For example, a formal specification method may be imposed for the development of some parts of the system, possibly in conformance to some standards.

Finally, the early identification and specification of the verification and validation requirements are essential to master the dependability of the final system. These requirements must cover both static aspects (inspections, reviews, static analysis) and dynamic aspects (structural tests, functional test, simulations).

Design. The design activity consists in defining the system architecture, and its decomposition into interacting hardware and software components.

It is fundamental to clearly identify at each level of decomposition the fault hypotheses under which the fault tolerance mechanisms are built. Indeed, the weaker the fault hypotheses are, the more the necessary fault-tolerance mechanisms are complex. This is particularly true in the field of distributed computing systems. The identification of these fault hypotheses and of the possible error propagation paths may be supported by methods like FMECA or FTA.

One key aspect of the design activities of a fault tolerant system is to decompose and structure the system in independent parts allowing faults and/or errors to be confined:

- Fault independence regions (FIR) define the different parts of the system between which faults occur independently. In other words, faults affecting different FIRs are supposed to be noncorrelated. This is part of the fault hypotheses under which the system is built and against which the system will be verified.
- Error containment regions (ECR) define the different parts of the system between which errors cannot propagate. This nonpropagation is ensured not only by the system structure itself (in independent parts) but also by adequate barriers against error propagation (necessary as soon as two ECRs have to interact).

In some fault-tolerance approaches, FIRs and ECRs are grouped together in what is then called Fault containment regions (FCR). According to the number of FIRs, ECRs, or FCRs defined, and to their overall organization, several fault-tolerance strategies can be envisaged (e.g., backward recovery, forward recovery, or compensation). The choice of strategy is often guided by the requirements concerning the maximum duration of service interruption: if no such service interruption is allowed, or if its maximum duration is very short, then compensation may be the only possible choice for the error recovery scheme.

The design of a fault-tolerant system must facilitate as much as possible the verification activities. This design strategy is known as *design for verification.* The design drivers of such a strategy are simplicity, rigorous design, clearly-defined interfaces, and accessibility of any system variable that plays an important role with respect to dependability (e.g., critical output, error signal, . . .). If some components are reused from earlier projects (or if some of them are commercial off-the-shelf components), then their impact on the overall system testability must also be assessed.

Production and Verification, Integration and Validation. The production activities consist in effectively building the system components according to the design. They are closely linked to the verification activities, which are in charge of checking that the produced components actually fulfill their specifications. The verification activities must then be carefully defined and followed for fault-tolerant systems, and their coverage regarding the different components and errors considered must be evaluated.

The last activities performed during system development are integration and validation. During the integration all the system components are gathered to build the final global system. Then, the validation activities consist in checking that the system as a whole matches the client's needs, especially from the viewpoint of its expected dependability level. Specific fault-injection campaigns may be used to validate the fault-tolerant mechanisms built into the system.

CASE STUDY

As examples of a real-life implementation, we have chosen to present two complementary parts of the Ariane 5 data management system focusing on fault-tolerance issues: the on board computer system and the ground control center. It is not intended to describe in full these two very complex systems but rather to provide the reader with a broad view of the fault-tolerance techniques employed, and how and where they are actually implemented.

The Ariane 5 On Board Computer System

The Ariane 5 data handling system is responsible for power management (storage and distribution) and the operational functions, that is, guidance, navigation, and sequencing.

The design drivers were: reliability, cost, mass, volume, ease of verification, and thermal dissipation. It has to be emphasized that this kind of system has a very short operational lifetime, about one hour or less after lift-off. The acceptable duration of service interruption is less than a tenth of a second. There are two reasons for this: first, the natural instability of the launch vehicle could lead to a quick destruction through structural overloading and second, the accuracy of payload injection is extremely critical. A classical approach would have been to implement a triplicated actively redundant system with fault masking. Unfortunately, the already mentioned design drivers did not allow for such a solution, so a mixed scheme had to be chosen.

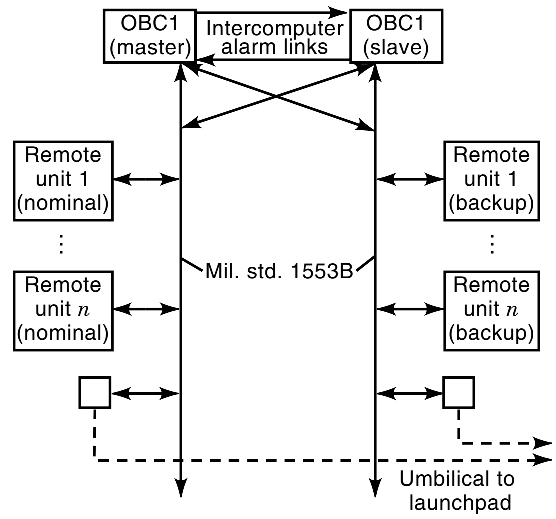

Figure 9. Architecture of the Ariane 5 fault-tolerant on-board data handling system.

In terms of fault handling, the on board computer system is centralized and can be seen as a pair of computers (OBC1 and OBC2) cross-linked through a redundant Mil. Std. 1553 bus (63) to two identical functional chains (sensors and actuators) (Fig. 9). These buses connect the on-board computer pool to all the Ariane 5 internal equipment, including interfaces to sensors and actuators. They operate in a nominal/standby configuration. The pool is organized as a master computer (OBC1) and a slave one (OBC2). The master computer controls the communications on the buses (nominal and standby) and executes the flight software. The slave passively monitors the communication buses to maintain a software context should it need to take over from OBC1. OBC2 does not check the behavior of OBC1; each computer has the capability of self detecting an abnormal local situation, to passivate itself and to inform the other that it has failed (details are given later).

Two phases can be defined for the system: from power-on to lift-off and from lift-off to payload delivery and mission termination. During the first phase, from a reliability standpoint, it has to be ensured that the hardware is properly functioning and that the flight software has been loaded correctly. There is also a system monitoring and control activity (under ground control) to check the readiness of the vehicle before launch, and to guarantee the safety of the launchpad and ground personnel.

During this phase, both OBCs act as slaves, with master control of the communication buses provided from the ground. Hardware checking is based on self-test, result monitoring by the ground, and the previously mentioned computer self-checking. Correct loading of the software is checked during the load operation by means of a proprietary secured packet protocol. This protocol checks that each packet has been correctly sent and received, and that the sequence of packets is in the right order. By allowing just a single faulty packet to be reloaded, the protocol can tolerate a defective communication medium without missing the launch window. A global cyclic redundancy checksum (CRC) ensures that the correct software has been loaded.

When the so-called synchronized sequence is entered, just before the effective launch, full control of the launcher is given over to the on-board computers. Both computers switch to the flight part of the software, OBC1 becomes master while OBC2 remains slave.

To support this description, mechanisms for error detection, error confinement, and error recovery have been implemented. Simply said, OBC1 executes the flight software, detects faulty units, passivates them by turning them off and switches on the redundant chain. If OBC1 self-detects itself in error it passivates itself and sends a signal to OBC2 through a dedicated link. OBC2 then switches to the master state and uses the context previously built up by monitoring of the communication buses to speed up software initialization, turn OBC1 off, and then control the launcher. When only one computer remains running, either OBC1 or OBC2, self-passivation is inhibited since there is no longer anything to be gained by attempting to recover from a computer failure.

At the level of remote units, error detection is ensured either by self monitoring for intelligent units (e.g., inertial reference system or engine actuator control electronics), or by the master OBC for dumb ones. The monitoring is based on reasonableness checks such as a range test on measurements, or a comparison between a model of the equipment and actual measures. Both local checks on individual items of equipment and global checks on the full launcher are carried out. For example, one global reasonableness check verifies that the launch vehicle trajectory remains in a predetermined flight corridor.

Since the system relies ultimately on the self-checking capability of each computer, let us now take a look at the internal architecture of an OBC. The computer is composed of three modules: power supply, processing unit, and input/output unit. The power supply is very classically built and electrical parameters such as output voltages are monitored. Should one of these parameters break some predefined nominal range, the power supply is turned off leading to a computer stop which is easily detected by the other OBC. The processing unit and the input/output unit are located on two separate boards and communicate through a shared memory. Each of these units contains error detecting and correcting (EDAC) memory, a watch dog, and an address violation detector. Any of these devices can trigger a computer stop with an associated context save operation for post mortem investigation. A computer is stopped by holding the CPU in the stop state until the power is turned off by the surviving computer. To avoid an erroneous interruption of OBC1 by OBC2, OBC2 checks that OBC1 has indeed passivated itself by verifying that there is no traffic on the bus. Furthermore, saturation of the buses by a permanently emitting device is avoided by defining a maximum message duration that is checked by every communication device.

Electrical isolation and electrical fault containment at the unit level are provided by transformer bus coupling, a dedicated power supply switching unit with electronic switches acting as power fuses, and optical couplers between computers.

At the 1553 bus level, the messages are checked for electrical correctness (e.g., fall and rise time and voltage level), and for protocol correctness (e.g., parity, response time, maximum emission duration, and word numbers associated to each subaddress). All parameters are statically defined to facilitate the detection of protocol violations.

The software is fully checked against the actual mission on a simulator. In flight, only the outputs of the software are checked against precomputed limits. There is no dedicated piece of software added to check it. As the mission is fully known before launch, this is a reasonable approach. As in any unique implementation, unrevealed specification faults or implementation faults can lead to a catastrophic failure.

To moderate this statement, it should be noted that the on board computer system has only a modest influence on the overall launcher reliability, as compared to the rate of mechanical or propulsive system failure.

The Ariane 5 Ground Control Center

The ground center represents the largest component in the Ariane 5 ground segment (64). It handles all interface management between the Ariane launcher and ground facilities during integration, testing, and launch preparation phases. It controls both electrical interfaces (main power supply, control, and data acquisition) and fluids. It ensures information exchange between on-board equipment and the ground, and controls the launch count-down during the five hours from tank filling until the synchronized sequence before launch and lift-off.

For operational considerations linked to the mission profile and other constraints, the control center has a fully decentralized architecture. It is a real-time system distributed over four sites more than 3 km apart and linked by an optical fiber network (Fig. 10).

A set of input/output (I/O) processors are in charge of interfacing with the controlled process and are located near the launcher. The control center manages and exploits more than 4000 wired inputs from and outputs to the process. These are managed by the electric power and housekeeping I/O processors. The fluid I/O processors are responsible for emptying and purging of launcher propellant gas. The 1553 I/O Processor manages the on-board 1553 data bus during prelaunch activities. Thirty-two workstations are in charge of the control operations in the Launch Center 3 control room. A further ten workstations, based in Evry (Metropolitan France), are used for real-time surveillance of the operations carried out 7000 km away in French Guyana.

The safety equipment and functional equipment of the control center are completely independent. The aim of the safety equipment is to enforce the fail-safe (FS) criterion in case of two failures. It includes the safety I/O processor (to acquire process data for safety monitoring), the safety operator workstation and the safety control panels (to interface with the safety automata executed by the safety I/O processor).

For availability reasons, the following subsystems are duplicated:

- The power supply subsystem
- The networking subsystem (control network, service network, archiving network, safety network)
- All the I/O processors (fluids I/O processors, 1553 I/O processors, electric power and housekeeping I/O processors, safety I/O processors)
- All the processing units except the evaluation unit, which is only used during the off-line launch debriefing

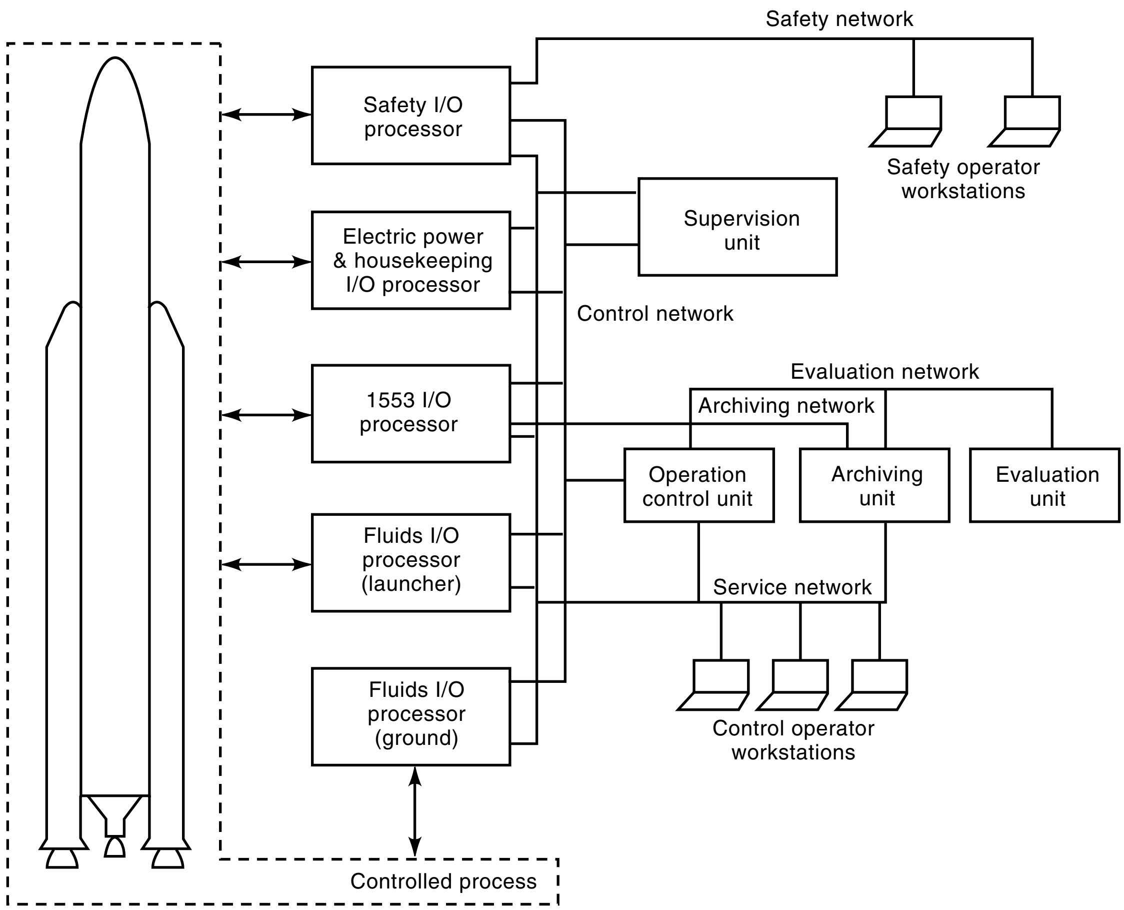

Figure 10. Architecture of the Ariane 5 fault-tolerant ground control center.

Dependability Requirements. Failure events are classified according to five levels:

1. Catastrophic event: loss of human life
2. Serious event: failure inducing a serious destruction of the ground-based facilities
3. Major event: failure inducing damage to the launcher or a postponement of the launch for more than one day
4. Significant event: failure inducing a postponement of the launch for less than one day
5. Minor event: failure during the off-line launch debriefing (after launch)

The control center must obey the FS/FS (fail safe/fail safe) rule for catastrophic events (i.e., safe with two consecutive faults). It must obey the FS criterion for serious or major events, and the FO criterion for significant events.

Depending on prelaunch phases, the control center must be FS for operations before count-down and FO/FS (fail operational/fail safe) for several operations during countdown. This implies:

- For the first failure: continued operation or stop in a safe state
- For the second failure: stop in a safe state

Fault-Tolerance Design. Two kinds of fault-tolerance techniques are used in the control center equipment due to the various operational or functional needs. Archiving units employ error detection and compensation (using "active" redundancy), whereas the operation control units and supervision unit use error detection and recovery (using "passive" redundancy). In the latter case, only a part of the software is executed within the standby unit, to continuously acquire dynamic context from the primary unit and to update the table of outstanding requests. With passive redundancy, the following states are defined for each unit of a redundant pair:

- Primary is the state of a unit able to control the process and to execute requests.
- Standby is the state of a unit ready to become active and replace the currently active unit, when the latter has been passivated.
- Operational is the state automatically reached after the correct execution of the first loop of unit self-test.
- Functional is the state automatically reached after loading the application software into the unit's memory.
- Frozen is the state of a unit after passivation; all interfaces are inhibited, but the unit's memory is not reset.
- Zero is the state of a unit after a reset.
- Off is the state of an equipment when no power is supplied.

Some of the data needed for a standby unit to be able to become primary cannot be acquired directly from the controlled process. This data forms the dynamic context that must be transmitted continuously by the primary unit to the standby unit.

A table of outstanding requests is used to determine whether or not a request has been executed by the primary unit (so that, should the current primary fail, the new primary can decide whether to re-execute the request).

Redundancy Management. A specific hardware board, called the reconfiguration board, is implemented in each redundant pair. This board carries out the following functions:

- Checking of unit state (primary, standby) and unit passivation
- Reception of heartbeats from each unit
- Switching of process control outputs
- Transmission of health status to the twin unit
- Transmission towards the supervision unit of the primary, standby, and health status bits

The primary and standby status bits are set to true when a unit is in the corresponding state. The health status of a unit is assessed by a set of hardware and software monitoring mechanisms that are chained together to form the monitor output synthesis chain (MOSC). The inputs and outputs of the reconfiguration board are directly wired independently of the equipment backplane bus.

The reconfiguration board is self-monitored by an internal watchdog that is rearmed periodically. The reliability of this board is maximized by the use of military standard components, preliminary burn-in, noise-protected inputs, and so on. A failure modes, effects and cause analysis concluded that no single fault could induce inadvertent redundancy switching.

Error Detection. Adequate means for error detection must be provided at both the unit level and system level. Three kinds of mechanisms are used: self-tests, self-checking, and functional checking.

For all detected errors, an alarm is generated. These alarms are classified according to three levels:

Level A: message for logbook
Level B: warning light turned on
Level C: unit passivation and redundancy switching

Self-tests are used only at system initialization. Successful execution of the self-tests is a prerequisite for the equipment to reach the operational state.

Self-checking is cyclic and carried out continuously while the control center is operational. It may be at board level, unit level, or system level.

- Self-checking is provided on all boards of every unit in the control center.
- For each duplicated subsystem, unit-level self-checking is supported by a dedicated processing board. A background task periodically resets a CPU watchdog and another background task periodically monitors the calling of all the cyclic tasks by checking their associated iteration counters. The execution of acyclic tasks is checked by input and output queue monitoring with generation of an alarm in case of saturation. For critical acyclic tasks, a periodic wake-up mechanism is also implemented. These mechanisms are relied upon to detect failures of the operating system and of the low level software.
- System level self-checking is carried out by the supervision unit that polls both the operator workstations and the networks.

Functional checking concerns hardware and low level software: power supply, memory parity, processing boards, network controllers, internal buses, wired interfaces. The results of these functional checks is reported to the local supervisor of the considered unit.

Fault Passivation. The passivation of a unit implies that all its interfaces with the network, the process and other equipment must be inhibited. The unit is put into the frozen state.

To avoid error propagation, a unit is automatically passivated by the reconfiguration board if the MOSC is open. The MOSC can be opened even due to a transient signal.

Redundancy Switching. When a redundant pair must be reconfigured, the primary unit is first put into the frozen state, and then into either the off state or the zero state. Then, the standby unit becomes active and switches over the outputs to the process.

This redundancy switching is carried out automatically by the reconfiguration boards of the primary and standby units.

Switching is initiated by the reconfiguration board of the primary unit when its MOSC opens (i.e., when an error has been locally detected). It passivates the faulty primary unit and puts its health status bit to bad.

When the reconfiguration board of the twin unit recognizes this bad health signal, it requests the local unit to GO-ACTIVE. Under software control, the unit that was previously on standby then checks that it is now both primary and not standby, and that its MOSC is closed. Analog outputs to the process are then switched without overlap, whereas switching of binary outputs must overlap to prevent glitches from being sent to the process. This is achieved by interlocking of the output switching relays.

SUMMARY AND FUTURE DIRECTIONS

This section summarizes the state-of-the-art in fault-tolerant computing and then provides some insights into the main challenges and the related potential solutions that should be tackled by the turn of the century.

Most work on fault-tolerant computing has been concerned with hardware defects, that is, accidental physical faults, resulting from internal or external (environmental) causes. These classes of faults are currently well mastered. Some of the most significant advances are:

- Error detecting and correcting codes, including also self-checking circuits for which they are a direct extension
- Error recovery procedures, either backward (retry) or forward, and their relationship with exception handling
- Distributed processing of errors and faults, and in particular the algorithms for reaching agreement in the presence of faults, including those leading to inconsistent behavior (Byzantine failures)

To build a dependable system, the use of suitable fault tolerance techniques should be complemented by a proper assessment strategy, encompassing both fault removal and fault forecasting. Here also, the most significant advances have concerned hardware failures, namely:

- The dependability evaluation of fault-tolerant systems based on probabilistic modeling, and in particular revelation of the influence of the efficiency—the coverage—of the fault tolerance mechanisms (24);
- The experimental evaluation of fault tolerance by means of fault injection, that corresponds to the testing of a fault-tolerant system regarding the specific inputs of such systems, that is, the faults (25).

As exemplified by several surveys of field data concerning hardware-fault tolerant systems, in practice, fault tolerance induces a significant increase in the mean time to failure, usually, from weeks to years. Referring such a result to the useful life of a computer system, a practical interpretation is that, on the average, a fault-tolerant system will not fail due to physical faults before it becomes obsolete. An important consequence of the ability to tolerate hardware (physical) faults is the logical modification of the ranking of the failure causes: design faults (especially, in software), are becoming the major source of failure followed by human–machine interactions (including both malicious faults and operator mistakes).

Thus, in spite of the progress made, fault-tolerant computing has still to cope with such fault classes. In the remainder of this section, we successively discuss these problematic fault classes and address the economic issues that are associated with a wider acceptance of fault-tolerant computing solutions.

Problematic Fault Classes

As already identified, three main classes of faults still pose problems: design faults, malicious faults, and interaction faults (see Fig. 2). The following subsections provide a brief discussion of these three sources of failures as well as the most promising solutions to cope with them.

Design Faults. Although the concern of software design faults has long since been identified and solutions have been put forward, it is worth noting that design faults remain a challenge for fault-tolerant computing. The problems encompass application software, executive software providing functional services, and software dedicated to fault tolerance. Indeed, the implementation of fault tolerance—even if restricted to physical faults—requires large volumes of code that may constitute 50% or more of the total volume of the software of a fault-tolerant system. In each case, the main issues result from the complexity of the functions to be computerized that poses new software engineering challenges and results in an inflation of the size of the codes to be developed, even in the case of embedded systems. More than 12 million of bytes were quoted for the Airbus A320; this size has risen to over 20 million for the A340. The severe problems affecting the design of the Advanced Automation System (AAS) for air traffic control and the deployment of the baggage-handling system of the Denver International Airport are illustrations of these difficulties.

The problem of design faults is not exclusive to software; it also affects hardware developments. The Intel Pentium microprocessor provides well-known examples: a circuit first marketed in May 1993, after being subjected to a significant series of fault-removal procedures, was found to exhibit a design fault in its divider hardware during the summer of 1994. Clearly, the development of modern microprocessors (more than 5.5 million transistors are quoted for the next Intel generation) is as difficult as the development of complex pieces of software. A detailed analysis of design faults in the Pentium II microprocessor has recently been reported in Ref. 65.

While tolerance of design faults (in hardware or software) has raised less attention, significant results have nevertheless been obtained. Two major types of techniques can be identified to cope with software design faults, depending on the considered objective: (1) either avoiding that the failure of a task provokes the failure of the whole system, or (2) ensuring service continuity. In the first case, the goal is to be able to detect rapidly an erroneous task and to abort it to avoid the propagation of the error(s); accordingly, such an approach is often termed fail-fast. In practice, error detection is achieved through defensive programming using executable assertions, and error processing is generally based on exception handling. Since the software faults that are found are often subtle faults whose activation is seldom reproducible, it has also been found that such a simple approach combined with error recovery techniques intended for hardware faults can prove to be very efficient for tolerating software faults.

The second alternative assumes that at least another component is available that is able to perform the same task and that was independently designed and implemented from the same specification, according to the design diversity principle. Three basic approaches can be identified (20): recovery blocks, N-version programming, and N-self-checking programming. Such approaches can be seen as resulting from the application to software of three classical hardware redundancy schemes (66): dynamic passive redundancy, static redundancy, and dynamic active redundancy.

This is still an open (research) domain, and thus somewhat prone to controversy; a recent development can be found in Ref. 67. Nevertheless, these results are already used in practical realizations, ranging from commercial systems (e.g., see the early Tandem Non-Stop system architecture) for the fail-fast fault tolerance approach, to highly critical applications such as civil avionics or railways, for the design diversity approach (see the Airbus example in the section "Error Compensation"). Similarly, design diversity is used to allow tolerance of hardware design faults and of compiler faults [see, for example, the diversified architecture of the Boeing 777 primary flight control computers (68)].

Malicious Faults. Malicious faults are having an increasing impact on a wide variety of "money-critical" application domains. In France, insurance company statistics about computer failures show that almost 62% of the incurred costs could be traced to malicious faults (1996 data); furthermore, this proportion has almost doubled during the last decade. It is likely that such figures apply comparatively in other industrial countries. Moreover, it was estimated by Dataquest in 1997 that industry would have to spend that year more than $6 billion worldwide for network security. It was further estimated that this spending would more than double by the end of the century, to reach almost $13 billion. It is worth noting that these amounts only account for services provided by external agencies and disregard the related in-house costs. Such a problem will be further exacerbated by the development of multimedia applications and the mutation of networks into the information freeways that will support them. Clearly, due to their lack of efficiency and the resulting high costs, fault-avoidance techniques alone can no longer cope with such classes of faults; they will have to be complemented by fault-tolerance techniques.

For instance, most security systems are developed around a trusted computing base (TCB), that is, that part (hardware and software) of the system that has to run securely for the whole system to be secure. Conversely, if the TCB fails (due to accidental or malicious faults), no security can be ensured. Fault tolerance can help to prevent such failures.

On the other hand, security relies in most cases on the correct behavior of some highly privileged persons: operators, administrators, security officers, and others. If any of them acts maliciously, he or she could violate most security measures. Consequently, security can be enhanced if fault-tolerance techniques are implemented to tolerate malevolence on the part of these persons.

When dealing with security, two kinds of faults need to be considered: *malicious logic* and *intrusions.* Malicious logic encompasses malevolent design faults, including trap-doors, logic bombs, Trojan horses, viruses, and worms. As for other design faults, tolerance of malicious logic has to be based on design diversity (69).

Intrusions are deliberate interaction faults that attempt to transgress the security policy of the system. The insertion of a virus or the execution of a worm are particular cases of intrusions. Intrusions can originate from external or internal intruders. External intruders are people not registered as users of the computing system. They thus have to deceive or by-pass the authentication and authorization mechanisms. Internal intruders are people who are registered as legitimate users, but who try to exceed or abuse their privileges. For instance, internal intruders could attempt to read confidential data or modify sensitive information to which they have no authorized access. To do so, they have to by-pass the authorization mechanisms. Abuse of privilege concerns some illegitimate (but authorized) actions. For instance, a security officer can (but should not) create dummy users, or an operator can (but should not) halt a computer at some inappropriate instant, causing a denial of service. Such intrusions are possible only because the least privilege principle is not perfectly implemented: otherwise, no illegitimate action would be authorized.

Intrusions and accidental faults may have the same effects, that is, that of modifying or destroying sensitive information or even disclosing confidential information. However, there are two main differences between tolerating accidental faults and tolerating intrusions. First, accidental faults are rare events, so there is a very low probability that two independent parts of the system be faulty at the same time. A single fault assumption is thus often justifiable and can be used to simplify the fault tolerance implementation. Conversely, several attacks by the same intruder can simultaneously affect different parts of the system and the single fault assumption may not be reasonable. Second, tolerance of accidental faults is not aimed at the preservation of the confidentiality of information. On the contrary, it introduces a redundancy that can be detrimental to confidentiality. For example, the mere replication of information leads to lower confidentiality since each copy can become the target for an intruder.

These specific requirements have led to the development of a particular fault-tolerance technique aimed at tolerating both accidental faults and intrusions, the fragmentation–redundancy–scattering (FRS) technique (70). The principle of FRS is to break information into fragments so that isolated fragments cannot provide significant information, to add redundancy to these insignificant fragments, and then to separate the fragments by scattering them in such a way that an intruder can only access isolated fragments.

Scattering can be topological (use of different sites or communication channels), temporal (transmission of fragments at random times or combined with other sources of fragments), or spectral (use of different frequencies in wideband communications). Another scattering technique is privilege scattering, which requires the cooperation of several entities to carry out an operation. Examples of such privilege scattering are the separation of duty proposed by Clark and Wilson (71) or the secret sharing proposed by Shamir (72).

The FRS technique has been successfully used to implement a secure distributed file storage, a distributed security server, and a fragmented data processing server.

The distributed file storage consists of several storage sites and user sites interconnected by a network. User sites are workstations that can be considered as secure during a user session since they can be easily configured to refuse any access from the network. Storage sites are dedicated to the storage of fragments. When a user file has to be stored, the file is fragmented on the user site. The file is first cut into fixed length pages so that all the fragments of every file have the same length. Each page is then ciphered, using cipher-block-chaining and a fragmentation key, and split into a fixed number of fragments. The fragments are given names by means of a one-way hash function taking as parameters the name of the file, the page number, the fragment number, and the fragmentation key. The fragments are then sent in a random order to the storage sites using multicast communication. A distributed algorithm guarantees that the requested number of copies is stored among the storage sites. Without knowing the fragmentation key, an intruder is not able to recognize from the fragment names how the ciphered page is to be rebuilt (due to the one-way function). Hence, even if he obtains the N fragments of a given page, he would have to attempt to rebuild about half the $N!$ possible fragment arrangements and carry out the same number of cryptanalyses to reconstitute the original page. In this case, the fragmentation technique multiplies the strength of the cipher by a coefficient of the order of the factorial of the number of fragments. Similar techniques have been proposed by Rabin (73) and the application of these techniques over the Internet has been proposed by Anderson (74).

The FRS technique has also been successfully applied to the management of system security functions, that is, user registration, authentication, authorization (control of access to objects or to servers), audit, key management. Certain pieces of information are confidential and must be fragmented (e.g., fragmentation keys), while others can simply be replicated (e.g., user identity). To tolerate intrusions, including intrusions by system administrators, these functions are implemented in a distributed security server composed of a set of sites, each administered by different people. This calls for the use of majority vote protocols and threshold algorithms to ensure that, as long as there exists a majority of nonfaulty sites (from the point of view of both accidental faults and intrusions), the security functions are properly carried out and no confidential information is disclosed. A similar approach has been proposed by Mike Reiter (75).

FRS can also be applied to the processing of confidential information by untrusted computers. In this case, the fragmentation relies on the structure of the information handled. By following an object-oriented approach, fragmentation consists in iterating the application design by decomposing the confidential objects until objects that do not handle confidential information are obtained. The confidential links between these objects are kept on the user site, the nonconfidential objects are made redundant and disseminated on the processing sites. To correct the modifications induced by accidental faults or intrusions, redundancy can be applied during the design by using the notion of inheritance or defined at a programming metalevel, using reflection (76).

Interaction Faults. The use of dependability concepts, and more precisely the use of fault-tolerance techniques, for the tolerance of hardware and software faults are now commonplace in critical systems. Because of this evolution, faults occurring during human–machine interaction are having an increasing impact on the dependability of critical systems that involve human operators (human–machine systems). Furthermore, technical progress has induced important changes in the operator involvement: the human operator is less implied in manual activity, but must increasingly carry out complex mental tasks. As a consequence, many accidents are judged to be caused by human error.

The statistics concerning the causes of accidents affecting commercial flights clearly illustrate the increasing impact of human faults: although the number of accidents has continuously decreased over the years, human faults have become the primary cause of accidents (77). In particular, the statistics published annually by Boeing concerning commercial flights in the United States rate these causes as high as 70% of the accidents for the years 1985 to 1995 (78). Such high proportions are also identified in all other application domains where operators are needed to interact with a computerized system. In Ref. 79, the author indicates that human faults are a primary cause of about 80% of all major accidents in aviation, power production, and process control.

Even if a significant proportion of interaction faults can be traced to design faults (poor design of the human–machine interface, lack of assistance by the system to the operators), human operator faults present a considerable threat. It is therefore necessary to take into account the role and characteristics of the human operator during the design of a human–machine system. This observation has led to various studies that consider the problems of human reliability during a complex system operation. Most work has aimed to reduce occurrences of human faults by methods that attempt to eliminate the conditions that can induce human faults. Harmonization of the allocation of tasks between the human and the machine, and the design of human–machine interfaces considering the user criteria are examples of potential methods for reducing human faults. These methods are important to increase the dependability of a human–machine system, but the complete elimination of human operator faults is not a realistic objective. Indeed, the human operator is frequently confronted with delicate and urgent situations requiring complex knowledge. Under stress, it is unreasonable to expect a human operator to act without any kind of error. It becomes therefore important to study means allowing the *tolerance* of human faults in the same way as for other classes of faults.

Current tolerance methods for operator faults are essentially based on the contribution of the human as a support for the tolerance, either by the operator himself, or through the pool of operators (both for masking erroneous commands and for analyzing troublesome situations). However, there is some recent work on how to use the technical system as a support for the tolerance of operator faults. In the case of systems possessing redundancy for tolerating physical and/or design faults, it may be interesting to see how this redundancy can be used to allow some tolerance of human faults.

Economic Challenges

Fault-tolerant solutions based on redundant architectures have been widely deployed in industry: first in specific domains such as space and telecommunications, and then, following the general trend of computerization, in all major industrial sectors.

In this current context, dependability requirements and economic challenges are increasingly mixed; accordingly, the massive solutions—especially the essentially proprietary hardware-based ones—are no longer acceptable. It follows that compromises must be found that encompass the development of low-cost fault tolerance solutions and the increasing role of software. Cost-effectiveness is indeed a major concern in the development of a fault-tolerant computer system. In particular, to cope with the high cost incurred by massive approaches, more cost-effective techniques such as control flow checking, or algorithmic-based fault tolerance techniques have been proposed.

In the sequel, we discuss three major aspects that are of concern: the provision of cost-effective solutions for temporary faults, the use of already-developed or commercial off-the-shelf components (COTS) in the design of fault-tolerant systems, and the incentive for developing COTS components featuring specific characteristics for supporting fault tolerance.

Tolerance of Temporary Faults. The vast majority of the faults observed in operation can be regarded as soft, that is, perceived as temporary faults (1). Accordingly, a cost-effective processing would require that the soft nature of the fault be explicitly accounted for before any unnecessary action (e.g., passivation) be undertaken. Indeed, such an action could be costly both in performance and resources. For example, commercial airlines report a rate of 50% of unjustified maintenance calls for on-board digital and electronic equipment. Simple threshold-based counter mechanisms (e.g., counting successive error occurrences) can significantly improve the balance between error processing and fault treatment decisions.

Similarly, due to the very soft nature of many of the software design faults activated in operation, it is very likely that such faults can be better tackled by using defensive programming techniques than through design diversity.

Commercial-Off-the-Shelf Components. The use of COTS components in fault-tolerant systems is not in itself a new problem. However, COTS components are now finding their way into very critical systems. Indeed, it is often no longer economically feasible to consider purpose-designed, non-COTS components, so designers of critical systems must find ways of accommodating them (e.g., see Ref. 80). From a software viewpoint, components of concern in safety-related applications include both packages that may form an integral part of the final application (e.g., operating systems—including the microkernel technology, databases, etc.) and tools used in the production of end-application software (e.g., compilers, code generators, etc.).

There are several issues at stake. For example, COTS components usually have limited self-checking capabilities, resulting in a rather restricted error-detection coverage. Another issue, concerning hardware COTS components, is that they may not be able to stand up to the severe constraints of some specific environments (e.g., radiation dose accumulation in space). However, the major issue with COTS components is undoubtedly that of residual design faults. Indeed, the salient characteristic of such components is the uncertainty that prevails about their origins and therefore their quality (81). Using components of unknown pedigree quite evidently introduces a formidable barrier to their acceptance for use in highly critical applications.

Various techniques can be deployed at the architectural level to help reduce the burden of validating COTS components, according to the criticality of the roles of the considered components:

- Critical COTS components, that is, COTS components playing roles on which critical services must depend
- Non-critical COTS components, that is, COTS components residing in an architecture supporting critical services, but not necessary for the provision of those services

For critical COTS components, at least three strategies can be considered for tolerating potential design faults:

1. Use diversified redundant COTS components to supply a service that is tolerant of design faults. This strategy is used in the Boeing 777 flight control system to provide protection against design faults in COTS hardware, Ada run-times and Ada compilers (68).
2. Diversify the usage patterns of identical redundant COTS components to decorrelate the activations of residual design faults. This diversification of usage can be used to argue the case for using identical COTS components in redundant channels. For example, the two redundant channels of the ELEKTRA system (82) are identically designed triple modular redundancy (TMR) systems using the same COTS processor type and the same COTS microkernel. However, the application codes executed by each channel are totally different so it can be argued that any design faults in the underlying COTS components will be activated in an uncorrelated fashion.
3. Use timing and execution checks in application software to provide an end-to-end verification of the correct execution of the underlying COTS hardware and software system. For example, this approach has been implemented using coding techniques. It relies on a precompilation of the application source code to augment it with instructions to calculate a signature for each operation as a separable arithmetic code. The signatures that are calculated at run-time are checked to verify that they respect the code. Any fault (design or otherwise) in the COTS software and hardware components used to generate and execute the run-time application code will, with a very high probability, alter or halt the stream of code-words generated at run-time and cause the checker to put the outputs of the system into a safe state (83,84).

For non-critical COTS components, whether or not they fulfill their intended role is secondary to ensuring that they do not detrimentally affect the execution of critical services. One fundamental mechanism for confining the effects of failures of noncritical COTS components is that of integrity level management. This allows COTS components of the most recent generation to be used, for example, to provide a state-of-the-art graphics display or network service. However, such components must be placed at a low integrity level so that their interactions with more critical components at higher integrity levels are rigorously policed. Integrity level management implies the use of spatial and temporal firewalls to partition components of different levels of criticality. Communication between components of different levels of criticality can be authorized, as long as it is mediated by a strictly enforced integrity policy (85).

There must be an approach to validation of COTS components that is consistent with the criticality of the supported services. In this respect, the paradox with using COTS components is that, on the one hand, their large-scale usage increases the confidence that one may have in their general reliability but, on the other hand, this same large-scale usage argument may not constitute a sufficient safety case for using COTS in critical applications. Thus, one is faced with providing further validation of components over whose design one has had no control.

There is currently little assistance from standards and guidelines on justifying the quality of COTS components, including the IEC (57). However, a recent aviation standard, the DO-178B (58) offers some pointers on experience-based justification to objectively support the large-scale usage argument of COTS software. Some of the objective arguments that can be advanced to reduce the lack of information on the production process include (81): product service history (experience-based arguments), use of certified products (e.g., validated compilers), and intensive statistical testing.

Statistical testing is feasible in applications where automatic comparison with expected outputs is possible. For example, benchmarks have been developed to analyze and compare the behavior of commercial operating systems in the presence of erroneous service requests. Such analyses can be useful to tailor or to wrap the operating system in such a way that it can handle the benchmarks properly. Indeed, delimiting the way a COTS package is used positively impacts the feasibility of certification. In spite of their merits, these approaches are tedious and can be invalidated when upgrading to a new version of the product (that may frequently be necessary to preserve the supplier's support).

General-Purpose Components with Fault-Tolerance Features. The high overhead associated with the design of redundant architectures is another important economic challenge. One potential solution to this problem lies in the incorporation of built-in self-test facilities in the design of the components, possibly at the expense of some performance degradation. The needed features may encompass the processing of both physical and design faults and thus concern either hardware or software components.

For hardware components, so far, besides the case of the iAPX 432™ launched in the early 1980s—and maybe because of the associated commercial flop—the microprocessor industry has been quite reluctant to firmly engage itself in such a direction (e.g., see Ref. 80). Nevertheless, the significant rate of improvement in clock speed achieved by new commercial microprocessors (more than 30% per year) should make this approach more practical and thus allow a real market to develop. Often, undocumented machine-specific registers exist in modern microprocessors [e.g., the Intel Pentium (86) or in the IBM POWER2 (87)] that can provide high-precision counting and/or accurate performance monitoring; by using those embedded software-accessible registers, one could easily derive enhanced observability for the purpose of error detection.

Similar features would be highly desirable for software components as well; these would consist, for example, in the incorporation of encapsulation mechanisms supporting defensive programming and interface error detection (by elaborating, for example, on the notion of wrappers elicited from the security arena as identified by Voas in Refs. 88 and 89). The availability of specific programming languages features can also significantly help in supporting fault-tolerant computing [e.g., the exception handling facilities in Ada or the reflection properties of certain object-oriented languages that can be used to implement user-transparent fault-tolerance mechanisms (90)].

GENERAL CONCLUSIONS

The ubiquity of computer systems, the trend toward the development of more open and interconnected systems, the increase in their complexity, their distribution, and widely varying size (constellation of satellites, air traffic networks, high-speed communication networks, multimedia applications, electronic trade, human–machine interactions, computer-assisted medicine, microsystems, etc.) are some of the new challenging targets for fault-tolerant computing. Two major issues have to be accounted for when addressing these challenges:

1. Fault tolerance is not just redundancy: although redundancy is the basic dimension, the proper management of the redundancies is essential to the success or failure of a fault-tolerant system, and such management relies heavily on the fault and error assumptions considered.
2. Fault tolerance is not merely common sense: it constitutes an engineering activity that has to follow precise rules; the still widespread misunderstanding that confines fault tolerance to common sense might explain the failures of several systems and naive entrepreneurs that have engaged themselves in this field.

To conclude, the following quotation from Ref. 80 seems particularly fitting: "After 30 years of study and practice in fault tolerance, high-confidence computing still remains a costly privilege of several critical applications. It is time to explore ways to deliver high-confidence computing to *all* users. . . . Fault tolerance is our best guarantee that high-confidence systems will not betray the intentions of their builders and the trust of their users by succumbing to physical, design, or human–machine interaction faults, or by allowing viruses and malicious acts to disrupt essential services."

BIBLIOGRAPHY

1. D. P. Siewiorek and R. S. Swarz, *Reliable Computer Systems—Design and Evaluation,* Burlington, MA: Digital Press, 1992.
2. A. Avizienis, Design of fault-tolerant computers, *AFIPS Conf. Proc.,* **31**: 1967, pp. 733–743.
3. J.-C. Laprie, Dependable computing: concepts, limits, challenges, *Spec. Issue, 25th Int. Symp. Fault-Tolerance Comput. FTCS-25,* Pasadena, CA, 1995, pp. 42–54.
4. J.-C. Laprie, Software-based critical systems, *Proc. 15th Conf. Comput. Saf., Reliab. Security SAFECOMP'96,* Vienna, Austria, 1996, pp. 157–170.
5. C. V. Ramamoorthy et al., Software engineering: problems and perspectives, *IEEE Comput.,* **17** (10): 191–209, 1984.
6. *Information Technology Security Evaluation Criteria,* Harmonized Criteria of France, Germany, the Netherlands, and the United Kingdom: Commission of the European Communities, 1991.
7. D. P. Siewiorek and D. Johnson, A design methodology for high reliability systems: The Intel 432, in D. P. Siewiorek and R. S. Swarz (eds.), *The Theory and Practice of Reliable System Design,* Burlington, MA: Digital Press, 1982, pp. 621–636.
8. D. Powell et al., The Delta-4 approach to dependability in open distributed computing systems, *18th Int. Symp. Fault-Tolerant Comput. Syst. FTCS-18,* Tokyo, 1988, pp. 246–251.
9. L. Lamport, R. Shostak, and M. Pease, The Byzantine generals problem, *ACM Trans. Prog. Lang. Syst.,* **4** (3): 382–401, 1982.
10. H. Mine and Y. Koga, Basic properties and a construction model for fail-safe logical systems, *IEEE Trans. Electron. Comput.,* **EC-16**: 282–289, 1967.
11. M. Nicolaïdis, S. Noraz, and B. Courtois, A generalized theory of fail-safe systems, *19th Int. Symp. Fault Tolerant Comput. FTCS-19,* Chicago, 1989, pp. 398–406.
12. R. D. Schlichting and F. B. Schneider, Fail-stop processors: an approach to designing fault-tolerant computing systems, *ACM Trans. Comput. Syst.,* **1** (3): 222–238, 1983.
13. A. Avizienis, Fault tolerance, the survival attribute of digital systems, *Proc. IEEE,* **66**: 1109–1125, 1978.
14. C. E. Landwher et al., A taxonomy of computer program security flaws, *ACM Comput. Surv.,* **26** (3): 211–254, 1994.
15. A. Avizienis and J. P. J. Kelly, Fault-tolerance by design diversity: concepts and experiments, *Computer,* **17** (8): 67–80, 1984.
16. T. A. Anderson and P. A. Lee, *Fault Tolerance—Principles and Practice,* Englewood Cliffs, NJ: Prentice-Hall, 1981; see also P. A. Lee and T. Anderson, *Fault Tolerance—Principles and Practice,* Vienna: Springer-Verlag, 1990.
17. W. C. Carter and P. R. Schneider, Design of dynamically checked computers, *IFIP'68 Congr.,* Amsterdam, The Netherlands, 1968, pp. 878–883.
18. J. Wakerly, *Error Detecting Codes, Self-Checking Circuits and Applications,* New York: Elsevier/North-Holland, 1978.
19. S. S. Yau and R. C. Cheung, Design of self-checking software, *1st Int. Conf. Reliab. Softw.,* Los Angeles, 1975, pp. 450–457.

20. J.-C. Laprie et al., *Definition and analysis of hardware-and-software fault-tolerance architectures, Computer,* **23** (7): 39–51, 1990.
21. W. R. Elmendorf, fault-tolerant programming, *2nd Int. Symp. Fault Tolerant Comput. FTCS-2,* Newton, MA, 1972, pp. 79–83.
22. B. Randell, System structure for software fault tolerance, *IEEE Trans. Softw. Eng.,* **SE-1**: 220–232, 1975.
23. B. W. Lampson, Atomic transactions, in B. W. Lampson (ed.), *Distributed Systems—Architecture and Implementation,* Berlin: Springer-Verlag, 1981, Lect. Notes *Comput. Sci.,* No. 105, pp. 246–265.
24. W. G. Bouricius et al., Reliability modeling for fault-tolerant computers, *IEEE Trans. Comput.,* **C-20**: 1306–1311, 1971.
25. J. Arlat et al., Fault injection for dependability validation—a methodology and some applications, *IEEE Trans. Softw. Eng.,* **16**: 166–182, 1990.
26. M.-C. Hsueh, T. K. Tsai, and R. K. Iyer, Fault injection techniques and tools, *IEEE Comput.,* **40** (4): 75–82, 1997.
27. E. Jenn et al., Fault injection into VHDL models: the MEFISTO tool, in B. Randell et al. (eds.), *Predictably Dependable Computing Systems,* Berlin: Springer-Verlag, 1995, pp. 329–346.
28. K. K. Goswami, R. K. Iyer, and L. Young, DEEND: A simulation-based environment for system level dependability analysis, *IEEE Trans. Comput.,* **46**: 60–74, 1997.
29. W. N. Toy, Fault-tolerant design of local ESS processors, *Proc. IEEE,* **66**: 1126–1145, 1978.
30. D. Avresky et al., Fault injection for the formal testing of fault tolerance, *IEEE Trans. Reliab.,* **45**: 443–455, 1996.
31. J. Christmansson and P. Santhaman, Error injection aimed at fault removal in fault tolerance mechanisms—criteria for error selection using field data on software faults, *Proc. 7th Int. Symp. Softw. Reliab. Eng. ISSRE'96,* White Plains, NY, 1996, pp. 175–184.
32. A. Mahmood and E. J. McKluskey, Concurrent error detection using watchdog processors—a survey, *IEEE Trans. Comput.,* **37**: 160–174, 1988.
33. J.-M. Ayache, P. Azéma, and M. Diaz, Observer: A concept for detection of control errors in concurrent systems, *9th Int. Symp. Fault-Tolerant Comput. FTCS-9,* Madison, WI, 1979, pp. 79–85.
34. C. Hennebert and G. Guiho, SACEM: A fault-tolerant system for train speed control, *23rd Int. Conf. Fault-Tolerant Comput. FTCS-23,* Toulouse, France, 1993, pp. 624–628.
35. D. J. Taylor, D. E. Morgan, and J. P. Black, Redundancy in data structures: Improving software fault tolerance, *IEEE Trans. Softw. Eng.,* **SE-6**: 383–394, 1980.
36. J. J. Horning et al., A program structure for error detection and recovery, in G. Goos and J. Hartmanis (eds.), *Operating Systems,* Berlin: Springer-Verlag, 1974, pp. 172–187.
37. K. M. Chandy and L. Lamport, Distributed snapshots: Determining global states of distributed systems, *ACM Trans. Comput. Syst.,* **3** (1): 63–75, 1985.
38. P. A. Bernstein, V. Hadzilacos, and N. Goodman, *Concurrency Control and Recovery in Database Systems,* Reading, MA: Addison-Wesley, 1987.
39. R. Koo and S. Toueg, Checkpointing and rollback recovery for distributed systems, *IEEE Trans. Softw. Eng.,* **SE-13**: 23–31, 1987.
40. D. Manivannan, R. H. B. Netzer, and M. Singhal, Finding consistent global checkpoints in a distributed computation, *IEEE Trans. Parallel Dist. Syst.,* **8**: 623–627, 1997.
41. J.-M. Hélary, A. Motefaoui, and M. Raynal, Communication-induced determination of consistent snapshots, *28th Int. Symp. Fault-Tolerant Comput. FTCS-28,* Munich, Germany, 1998, pp. 208–217.
42. M. J. Litzkow, M. Livny, and M. W. Mutka, Condor—A hunter of idle workstations, *8th Int. Conf. Distributed Comput. Syst. ICDCS-8,* San Jose, CA, 1988, pp. 104–111.
43. J. Bartlett, J. Gray, and B. Horst, Fault tolerance in tandem computer systems, in A. Avizienis, H. Kopetz, and J.-C. Laprie (eds.), *The Evolution of Fault-Tolerant Systems,* Vienna: Springer-Verlag, 1987, pp. 55–76.
44. J. Gray, Why do computers stop and what can be done about it? *5th Symp. Reliab. Distrib. Softw. Database Syst.,* Los Angeles, 1986, pp. 3–12.
45. D. Brière and P. Traverse, AIRBUS A320/A330/A340 electrical flight controls—a family of fault-tolerant systems, *23rd Int. Conf. Fault-Tolerant Comput. FTCS-23,* Toulouse, France, 1993, pp. 616–623.
46. A. Avizienis et al., The UCLA DeDiX system: A distributed testbed for multiple-version software, *15th Int. Symp. Fault-Tolerant Comput. FTCS-15,* Ann Arbor, MI, 1985, pp. 126–134.
47. W. W. Peterson and E. J. Weldon, *Error-Correcting Codes,* Cambridge, MA: MIT Press, 1972.
48. L. Lamport and N. Lynch, Distributed computing: models and methods, in J. van Leeuwen (ed.), *Handbook of Theoretical Computer Science,* Amsterdam: Elsevier, 1990, ser. B, pp. 1159–1199.
49. M. J. Fischer, N. A. Lynch, and M. S. Paterson, Impossibility of distributed consensus with one faulty process, *J. Assoc. Comput. Mach.,* **32** (2): 374–382, 1985, originally published as MIT Tech. Rep. MIT/LCS/TR-282, 1982.
50. C. Fetzer and F. Cristian, Fail-awareness: An approach to construct fail-safe applications, *27th Int. Symp. Fault-Tolerant Comput. FTCS-27,* Seattle, WA, 1997, pp. 282–291.
51. D. L. Mills, Internet time synchronization: The network time protocol, *IEEE Trans. Commun.,* **39**: 1482–1493, 1991.
52. G. Coulouris, J. Dollmore, and T. Kidberg, *Distributed Systems: Concepts and Design,* Reading, MA: Addison-Wesley, 1994.
53. N. A. Lynch, *Distributed Algorithms,* San Francisco: Morgan Kaufmann, 1996.
54. V. Hadzilacos and S. Toueg, Fault-tolerant broadcast and related problems, in S. Mullender (ed.), *Distributed Systems,* New York: ACM Press, 1993, pp. 97–145.
55. D. Powell, Distributed fault-tolerance—lessons from delta-4, *IEEE Micro,* **14** (1): 36–47, 1994.
56. S. B. Davidson, H. Garcia-Molina, and D. Skeen, Consistency in partitioned networks, *ACM Comput. Surv.,* **17** (3): 341–370, 1985.
57. *Functional safety: Safety-related systems,* Draft International Standard IEC 1508, Int. Electrotech. Commission, Geneva, Switzerland, IEC Document N°65A/179/CDV, June 1995.
58. *Software considerations in airborne systems and equipment certification,* RTCA, Inc., Washington D.C., Advisory Circular N°D0-178B, January 1992.
59. W. Kriedte, *ECSS—A single set of European space standards,* European Space Research & Technology Centre (ESTEC), Noordwijk, The Netherlands, 1996.
60. J.-C. Laprie et al., *Dependability Guidebook,* Toulouse: Cépaduès-Editions, 1995, in French.
61. Military Handbook N°217F, *Reliability Prediction of Electronic Equipment,* Department of Defense, USA.
62. J.-C. Laprie and K. Kanoun, Software reliability and system reliability, in M. R. Lyu (ed.), *Handbook of Software Reliability Engineering,* New York: McGraw-Hill, 1996, pp. 27–69.
63. Military Standard N°1553B, *Interface standard for digital time division command/response multiplex data bus,* Department of Defense, USA.
64. J.-L. Dega, The redundancy mechanisms of the Ariane 5 operational control center, *26th Int. Symp. Fault-Tolerant Comput. (FTCS-26),* Sendai, Japan, 1996, pp. 382–386.

65. A. Avizienis and Y. He, The taxonomy of design faults in COTS microprocessors, *Dig. FastAbstracts 28th Int. Symp. Fault-Tolerant Comput. FTCS-28,* Munich, Germany, 1998, pp. 52–53.

66. W. C. Carter, Hardware fault tolerance, in T. Anderson (ed.), *Resilient Computing Systems,* London: Collins, 1985, pp. 11–63.

67. L. Hatton, N-version design versus one good version, *IEEE Software,* November/December, pp. 71–76, 1997.

68. Y. C. B. Yeh, Dependability of the 777 primary flight control system, *5th IFIP 10.4 Work. Conf. Depend. Comput. Crit. Appl. DCCA-5,* Urbana-Champaign, IL, 1995, pp. 3–17.

69. M. K. Joseph and A. Avizienis, A fault tolerance approach to computer viruses, *1988 Symp. Security Privacy,* Oakland, CA, 1988, pp. 52–58.

70. Y. Deswarte, L. Blain, and J.-C. Fabre, Intrusion tolerance in distributed systems, *Symp. Res. Security Privacy,* Oakland, CA, 1991, pp. 110–121.

71. D. D. Clark and D. R. Wilson, A comparison of commercial and military computer security policies, *Symp. Security Privacy,* Oakland, CA, 1987, pp. 184–194.

72. A. Shamir, How to share a secret, *Commun. Assoc. Comput. Mach.,* **22** (11): 612–631, 1979.

73. M. O. Rabin, Efficient dispersal of information for security, load balancing and fault tolerance, *J. Assoc. Comput. Mach.,* **36** (2): 335–348, 1989.

74. R. J. Anderson, The eternity service, *Int. Conf. Theory Appl. Cryptol. PRGOCRYPT'96,* Prague, 1996.

75. M. K. Reiter, Secure agreement protocols: Reliable and atomic group multicast in rampart, *ACM Conf. Comput. Commun. Security,* 1994, pp. 68–80.

76. J.-C. Fabre et al., Implementing fault-tolerant applications using reflective object-oriented programming, *25th Int. Conf. Fault-Tolerant Comput. FTCS-25,* Pasadena, CA, 1995, pp. 489–498.

77. B. Ruegger, *Human Error in the Cockpit,* Swiss Reinsurance Company, 1990.

78. *Statistical Summary of Commercial Jet Aircraft Accidents,* Seattle, WA: Boeing Commercial Aircraft Group, 1996.

79. E. Hollnagel, *Human Reliability Analysis: Context and Control,* Computers and People Series, London: Academic Press, 1993.

80. A. Avizienis, Towards systematic design of fault-tolerant systems, *Computer,* **30** (4): 51–58, 1997.

81. I. J. Sinclair, *The Use of Commercial Off-The-Shelf COTS Software in Safety-Related Applications,* Glasgow: Real-Time Engineering Ltd., 1995, HSE Contract Res. Rep. No. 80/1995.

82. H. Kantz and C. Koza, The ELEKTRA railway signalling system: Field experience with an actively replicated system with diversity, *25th Int. Symp. Fault-Tolerance Comput. FTCS-25,* Pasadena, CA, 1995, pp. 453–458.

83. P. Forin, Vital coded microprocessor principles and application for various transit systems, *Proc. IFAC Conf. Control, Comput., Commun. Transp. CCCT'89,* Paris, 1989, pp. 137–142.

84. J. A. Profeta et al., Safety-critical systems built with COTS, *IEEE Comput.,* **29** (11): 54–60, 1996.

85. E. Totel et al., Supporting multiple levels of criticality, *28th Int. Symp. Fault-Tolerant Comput. FTCS-28,* Munich, Germany, 1998, pp. 70–79.

86. T. Mathisen, Pentium secrets, *Byte,* **19** (7): 191–192, 1994.

87. E. H. Welbon et al., The POWER2 performance monitor, *IBM J. Res. Develop.,* **38** (5): 545–554, 1994.

88. *Colloquium on COTS and Safety Critical Systems,* London: Institute of Electrical Engineers, 1997, Dig. No. 97/013.

89. F. Salles, J. Arlat, and J.-C. Fabre, Can we rely on COTS microkernels for building fault-tolerant systems, *6th Workshop Future Trends Distrib. Comput. Syst.,* Tunis, Tunisia, 1997, pp. 189–194.

90. J.-C. Fabre and B. Randell, An object-oriented view of fragmented data processing for fault and intrusion tolerance in distributed systems, in Y. Deswarte, G. Eizenberg, and J.-J. Quisquater (eds.), *2nd Eur. Symp. Res. Comput. Security ESORICS 92,* (Toulouse, France), Berlin: Springer-Verlag, 1992, pp. 193–208.

J. Arlat
Y. Crouzet
Y. Deswarte
J.-C. Laprie
D. Powell
LAAS-CNRS

P. David
J. L. Dega
C. Rabéjac
H. Schindler
J.-F. Soucailles
Matra Marconi Space France

FAULT-TOLERANT SYSTEMS ANALYSIS. See Reliability of redundant and fault-tolerant-systems.

FAX. See Facsimile equipment.

FDDI. See Metropolitan area networks.