A Conceptual Framework for System Fault Tolerance

Source http://hissa.ncsl.nist.gov/chissa/SEI_Framework/framework_1.html

---

A major problem in transitioning fault tolerance practices to the practitioner community is a lack of a common view of what fault tolerance is, and how it can help in the design of reliable computer systems. This document takes a step towards making fault tolerance more understandable by proposing a conceptual framework. The framework provides a consistent vocabulary for fault tolerance concepts, discusses how systems fail, describes commonly used mechanisms for making systems fault tolerant, and provides some rules for developing fault tolerant systems.

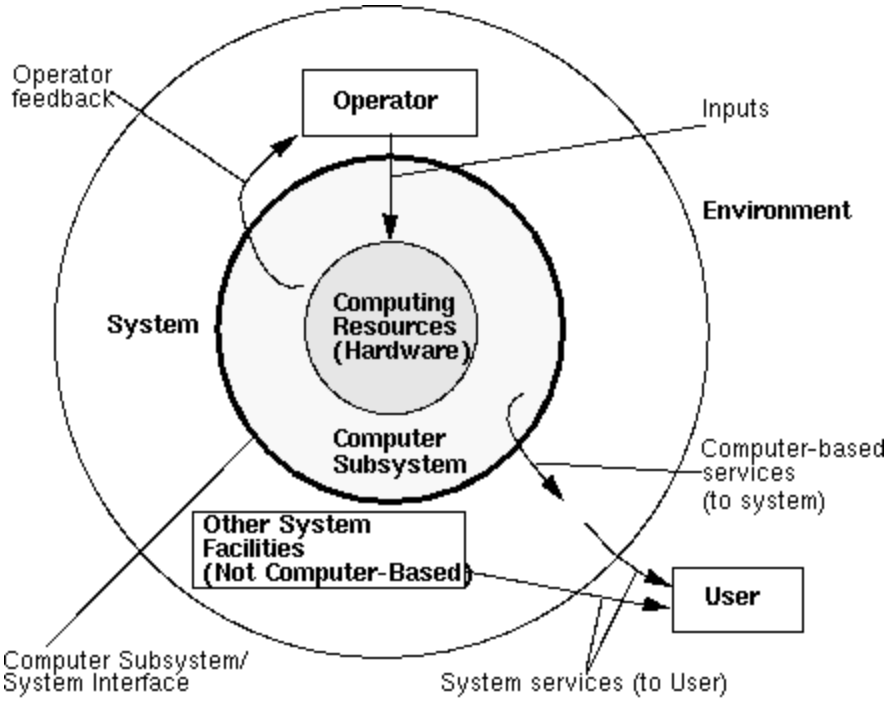
1 Introduction

One of the major problems in transitioning fault tolerance practices to the practitioner community is a lack of a common view of exactly what fault tolerance is, and how it can help in the design of reliable systems. One step towards making fault tolerance more understandable is to provide a conceptual framework. The purpose of this document is to propose such a framework.

This document begins with a discussion of what constitutes a system. From there a standard vocabulary of system fault tolerance is developed, using commonly accepted terminology (e.g., [Laprie 92]) wherever possible. Vocabulary terms are illustrated with computer system examples and an alternate set of examples from a radically different type of system, a bridge. Next, the document discusses how systems fail, including fault classes. This is followed by a summary of the existing approaches to implementing fault tolerance. The final section revisits the key concepts of the paper and proposes some rules for fault tolerant system design.

1.1 What is a System?

In the software engineering arena, a system is often equated with software, or perhaps with the combination of computer hardware and software. Here, we use the term system in its broader sense. As shown in Figure 1-1, a *system* is the entire set of components, both computer related, and non-computer related, that provides a service to a user.



System Relationships

For instance, an automobile is a system composed of many hundreds of components, some of which are likely to be computer subsystems running software.

A system exists in an environment (e.g., a space probe in deep space), and has operators and users (possibly the same). The system provides feedback to the operator and services to the user. Operators are shown inside the system because operator procedures are usually a part of the system design, and many system functions, including fault recovery, may involve operator action. Not shown in the figure, but of equal importance, are the system's designers and maintainers.

Systems are developed to satisfy a set of requirements that meet a need. A requirement that is important in some systems is that they be highly dependable. Fault tolerance is a means of achieving dependability.

There are three levels at which fault tolerance can be applied. Traditionally, fault tolerance has been used to compensate for faults in computing resources (hardware). By managing extra hardware resources, the computer subsystem increases its ability to continue operation. *Hardware fault tolerance* measures include redundant communications, replicated processors, additional memory, and redundant power/energy supplies. Hardware fault tolerance was particularly important in the early days of computing, when the time between machine failures was measured in minutes.

A second level of fault tolerance recognizes that a fault tolerant hardware platform does not, in itself, guarantee high availability to the system user. It is still important to structure the computer software to compensate for faults such as changes in program or

data structures due to transients or design errors. This is *software fault tolerance*. Mechanisms such as checkpoint/restart, recovery blocks and multiple-version programs are often used at this level.

At a third level, the computer subsystem may provide functions that compensate for failures in other system facilities that are not computer-based. This is *system fault tolerance*. For example, software can detect and compensate for failures in sensors. Measures at this level are usually application-specific. It is important that fault tolerance measures at all levels be compatible, hence the focus on system-level issues in this document.

---

2 Requirements

Many of the terms used in this section are defined in Section 3.

2.1 Dependable Systems

Hazards to systems are a fact of life. So are faults. Yet we want our systems to be dependable. A system is dependable when it is trustworthy enough that reliance can be placed on the service that it delivers [Carter 82]. For a system to be dependable, it must be available (e.g., ready for use when we need it), reliable (e.g., able to provide continuity of service while we are using it), safe (e.g., does not have a catastrophic consequence on the environment), and secure (e.g., able to preserve confidentiality) [Laprie 92].

Although these system attributes can be considered in isolation, in fact they are interdependent. For instance, a system that is not reliable is also not available (at least when it is not operating correctly). A secure system that doesn't allow an authorized access is also not available. An unreliable system to control nuclear reactors is probably not a safe one either.

2.2 Approaches to Achieving Dependability

Achieving the goal of dependability requires effort at all phases of a system's development. Steps must be taken at design time, implementation time, and execution time, as well as during maintenance and enhancement. At design time, we can increase the dependability of a system through *fault avoidance* techniques. At implementation time, we can increase the dependability of the system through *fault removal* techniques. At execution time, *fault tolerance* and *fault evasion* techniques are required.

2.2.1 Fault Avoidance

Fault avoidance uses various tools and techniques to design the system in such a manner that the introduction of faults is minimized. A fault avoided is one that does not have to be dealt with at a later time. Techniques used include design methodologies, verification and validation methodologies, modelling, and code inspections and walk-throughs.

2.2.2 Fault Removal

Fault removal uses verification and testing techniques to locate faults enabling the necessary changes to be made to the system. The range of techniques used for fault removal includes unit testing, integration testing, regression testing, and back-to-back testing. It is generally much more expensive to remove a fault than to avoid a fault.

2.2.3 Fault Tolerance

In spite of the best efforts to avoid or remove them, there are bound to be faults in any operational system. A system built with fault tolerance capabilities will manage to keep operating, perhaps at a degraded level, in the presence of these faults. For a system to be fault tolerant, it must be able to detect, diagnose, confine, mask, compensate and recover from faults. These concepts will be discussed thoroughly in Section 4 of this paper.

2.2.4 Fault Evasion

It is possible to observe the behavior of a system and use this information to take action to compensate for faults before they occur. Often, systems exhibit a characteristic or normal behavior. When a system deviates from its normal behavior, even if the behavior continues to meet system specifications, it may be appropriate to reconfigure the system to reduce the stress on a component with a high failure potential. We have coined the term *fault evasion* to describe this practice. For example, a bridge that sways as traffic crosses may not be exceeding specifications, but would warrant increased attention from a bridge inspector. Similarly, a computer system that suddenly begins to respond sluggishly often prompts a prudent user to backup any work in progress, even though overall system performance may be within specification.

2.3 Dependability Specifications

The degree of fault tolerance a system requires can be specified quantitatively or qualitatively.

2.3.1 Quantitative Goals

A quantitative reliability goal is usually expressed as the maximum allowed failure-rate. For example, the reliability figure usually stated as a goal for computer systems in commercial aircraft is less than 10^{-9} failures per hour. The problem with stating reliability requirements in this manner is that it is difficult to know when it has been achieved. Butler has pointed out that standard statistical methods cannot be used to show such reliability with either standard or fault tolerant software [Butler 91]. It is also clear

that there is no way to achieve confidence that a system meets such a reliability goal through random testing. Nevertheless, reliability goals are often expressed in this manner.

2.3.2 Qualitative Goals

An alternative method of specifying a system's reliability characteristics is to specify them qualitatively. Typical specifications would include:

Fail-safe

Design the system so that, when it sustains a specified number of faults, it fails in a safe mode. For instance, railway signalling systems are designed to fail so that all trains stop.

Fail-op

Design the system so that, when it sustains a specified number of faults, it still provides a subset of its specified behavior.

No single point of failure

Design the system so that the failure of any single component will not cause the system to fail. Such systems are often designed so that the failed component can be replaced or repaired before another failure occurs.

Consistency

Design the system so that all information delivered by the system is equivalent to the information that would be delivered by an instance of a non-faulty system.

---

3 Fault Tolerance Concepts With Examples

3.1 Introduction

A major purpose of this document is to define system fault tolerance concepts in an understandable manner. To help the reader understand the concepts, each concept is illustrated by a set of examples after it is defined. Where possible the same two examples are used throughout the document.

It is often easier to understand a concept using an analogy. This avoids the problems associated with the unintentional overloading of the meaning of words that often occurs in a familiar context. Thus one of the two examples that will appear throughout this document is a (simplified) highway bridge over a river. The other example will be in the probably more familiar world of computers.

3.2 Faults and Failures

3.2.1 Definitions

The terms failure and fault are key to any understanding of system reliability. Yet they are often misused. One describes the situation(s) to be avoided, while the other describes the problem(s) to be circumvented.

3.2.1.1 Concept Definition

Over time, failure has come to be defined in terms of specified service delivered by a system. This avoids circular definitions involving essentially synonymous terms such as defect, etc. This distinction appears to have been first proposed by Melliar-Smith [Melliar-Smith 75]. A system is said to have a *failure* if the service it delivers to the user deviates from compliance with the system specification for a specified period of time. While it may be difficult to arrive at an unambiguous specification of the service to be delivered by any system, the concept of an agreed-to specification is the most reasonable of the options for defining satisfactory service and the absence of satisfactory service, failure.

The definition of failure as the deviation of the service delivered by a system from the system specification essentially eliminates "specification" faults or errors. While this approach may appear to be avoiding the problem by defining it away, it is important to have some reference for the definition of failure, and the specification is a logical choice. The specification can be considered as a boundary to the system's region of concern, discussed later. It is important to recognize that every system has an explicit specification, which is written, and an implicit specification that the system should at least behave as well as a reasonable person could expect based on experience with similar systems and with the world in general. Clearly, it is important to make as much of the specification as explicit as possible.

It has become the practice to define faults in terms of failure(s). The concept closest to the common understanding of the word fault is one that defines a *fault* as the adjudged cause of a failure. This fits with a common application of the verb form of the word fault, which involves determining cause or affixing blame. However, this requires an understanding of how failures are caused. An alternate view of faults is to consider them failures in other systems that interact with the system under consideration--either a subsystem internal to the system under consideration, a component of the system under consideration, or an external system that interacts with the system under consideration

(the environment). In the first instance, the link between faults and failures is cause; in the second case it is level of abstraction or location.

The advantages of defining faults as failures of component/interacting systems are: (1) one can consider faults without the need to establish a direct connection with a failure, so we can discuss faults that do not cause failures, i.e., the system is naturally fault tolerant, (2) the definition of a fault is the same as the definition of a failure with only the boundary of the relevant system or subsystem being different. This means that we can consider an obvious internal defect to be a fault without having to establish a causal relationship between the defect and a failure at the system boundary.

In light of the proceeding discussion, a *fault* will be defined as the failure of (1) a component of the system, (2) a subsystem of the system, or (3) another system which has interacted or is interacting with the considered system. Every fault is a failure from some point of view. A fault can lead to other faults, or to a failure, or neither.

A system with faults may continue to provide its service, that is, not fail. Such a system is said to be *fault tolerant*. Thus, an important motivation for differentiating between faults and failures is the need to describe the fault tolerance of a system. An observer inspecting the internals of the system would say that the faulty component had failed, because the observer's viewpoint is now at a lower level of detail.

The observable effect of a fault at the system boundary is called a *symptom*. The most extreme symptom of a fault is a failure, but it might also be something as benign as a high reading on a temperature gauge. Symptoms are discussed in greater detail later.

A Digression on Errors

The term error often is used in addition to the terms fault and failure, as in the article by Melliar-Smith previously cited. Often, errors are defined to be the result of faults, leading to failures. Informally, errors seem to be a passive concept associated with incorrect values in the system state. However, it is extremely difficult to develop unambiguous criteria for differentiating between faults and errors. Many researchers refer to value faults, which are also clearly erroneous values. The connection between error and failure is even more difficult to describe.

As we have seen, differentiation between failures and faults is essential for fault tolerant systems. A third term, error, adds little to this distinction and can be a source of confusion. Consequently, we substitute the term fault for the common uses of the term error. Generally, references to the term "error" in the literature can be fitted to the context of this document by substituting the term "fault."

3.2.1.2 Bridge Example

To help understand these definitions, consider the example of a highway bridge over a river. Some time after developing this example, Alfred Spector has pointed out that a precedent for using this as an example exists in an article comparing practices in bridge design with practices in software design [Spector 86].

When designing the bridge the designer must consider a myriad of details regarding requirements, and the environment in which the bridge would operate. Suppose a 20 ton truck drives onto the bridge and the bridge collapses. From the truck's point of view, the bridge has failed. But what is the fault that led to the failure? There are lots of possible answers to this:

1. The designer of the bridge did not allow for appropriate bridge loading. This could be:
    1. A specification fault if the highway department did not anticipate that 20 ton trucks would need to use the bridge, or
    2. A design fault if the specification called for it being able to carry 20 ton trucks.
    3. An implementation fault if the fabricator didn't correctly follow the design.
2. The truck driver ignored a "Load Limit" sign. This would be a user fault.
3. A worker for the highway department posted an erroneous "Load Limit" sign. This would be an operator fault.
4. The people preparing the documentation for the bridge mistakenly indicated that the bridge would support 20 tons, when in fact it was only designed to support 10 tons. The highway department erected a 20 ton "Load Limit" sign. This would be a documentation fault, followed by an operator fault.
5. Previously a 30 ton truck crossed the bridge and sufficiently weakened the structure so that the subsequent 20 ton truck caused the bridge to fail. This, again, would be a user fault (the prior user).
6. Inadequate maintenance caused the bridge to develop structural flaws which led to it being unable to support a 20 ton truck. This would be another operator fault.
7. A barge on the river hit the bridge and knocked out a support. Or a 100 year flood came along and washed the bridge out, or a meteor crashed through the bridge. These would be environmental faults.

As an example of a fault which does not lead to a failure, consider the same bridge with a crack in its concrete roadbed. There is no failure involved if the bridge continues to carry the loads requested of it in spite of this fault. It may be the result of normal wear and tear on the roadbed. However, a thorough inspection of the bridge might discover that the crack in the roadbed was a symptom of a faulty strut, only observable by x-raying the strut. From the point of view of the bridge inspector, the strut would have failed. This component failure is an internal fault.

Scenarios like this can be generated ad infinitum. Note that a fault does not lead to a failure unless the result is observable by the user, and leads to the bridge becoming unable to deliver its specified service. This means that one person's fault is another person's failure. For instance, in example 4 above, from the point of view of the highway department the erroneous documentation was a fault that led to an operator failure. From the point of view of the user of the bridge the erroneous documentation was a documentation fault that led to an operator fault which led to a bridge failure.

3.2.1.3 Computer System Example

Consider a computer system running a program to control the temperature of a boiler by calculating the firing rate of the burner for the boiler. If a bit in memory becomes stuck at one, that is a fault. If the memory fault effects the operation of the program in such a way that the computer system outputs cause the boiler temperature to rise out of the normal zone, that is a computer system failure and a fault in the overall boiler system. If there is a gauge showing the temperature of the boiler, and its needle moves into the "yellow" zone (abnormal, but acceptable), that is a symptom of the system fault. On the other hand, if the boiler explodes because of the faulty firing calculation, that is a (catastrophic) system failure.

The reasons for the memory fault could be manifold. The chip used might not have been manufactured to specification (a manufacturing fault), the hardware design may have caused too much power to be applied to the chip (a system design fault), the chip design may be prone to such faults (a chip design fault), a field engineer may have inadvertently shorted two lines while performing preventive maintenance (a maintenance fault), etc.

3.3 Dependency Relations

3.3.1 Definitions

A major concern in fault tolerant system design and verification is the identification of dependencies. Dependencies may be static, remaining the same over the life of the system, or they may change, either by design or because of the effects of faults.

3.3.1.1 Concept Definition

A component of a system is said to *depend* on another component if the correctness of the first component's behavior requires the correct operation of the second component. Traditionally, the set of possible dependencies in a system are considered to form an acyclic graph. The term fault tree analysis seems to imply this, among other things. Indeed, many systems exhibit this behavior, in which one fault leads to another which leads to another until eventually a failure occurs. It is possible, however, for a dependency relationship to cycle back upon itself. A dependency relationship is said to be *acyclic* if it forms part of a tree. A *cyclic* dependency relationship is one that cannot be described as part of a tree, but rather must be described as part of a directed cyclic graph.

3.3.1.2 Bridge Example

In a bridge, the structural integrity of the roadbed depends, in part, on the structural integrity of the bridge piers. In a suspension bridge, the structural integrity of each of the suspension lines depends on each of the others.

A weakened strut may lead to another strut developing faults, which in turn could put more load on the original strut causing it to weaken further. This would be a cyclic fault trajectory. If the faults which developed in the second strut did not further trigger the fault in the first strut it would be an acyclic fault trajectory.

3.3.1.3 Computer System Example

In a computer system, consider two cooperating sequential processes using semaphores to synchronize. If either process fails to release the semaphore when it should, then the other process will fail as well. Thus they are mutually dependent.

A piece of software with a bad bit set in one of its instructions could cause a bad value to be calculated which could cause the program to take a different logical path. This different path might cause the original piece of software to be re-executed which could lead to still other unexpected behavior. This would be a cyclic fault trajectory. If the original fault did not ultimately result in the fault being triggered again it would be an acyclic fault trajectory.

3.3.2 Failure Regions

Defining a failure region limits the consideration of faults and failures to a portion of a system and its environment. This is necessary to insure that system specification, analysis and design efforts are concentrated on the portions of a system that can be observed and controlled by the designer and user. It helps to simplify an otherwise overwhelming task.

3.3.2.1 Concept Definition

A system is typically made up of lots of components parts. These components are, in turn, made up of sub-components. This continues arbitrarily until an *atomic* component (a component that is not divisible or that we choose not to divide into sub-components) is reached. Although all components are theoretically capable of having faults, for any system there is a level beyond which the faults are "not interesting". This level is called the *fault floor*. Atomic components lie at the fault floor. We are concerned with faults emerging from atomic components, but not faults that lie within these components.

Similarly, as components are aggregated into a system, eventually the system is complete. Everything else (e.g., the user, the environment, etc.) is not a part of the system. This is the *system boundary*. Failures occur when faults reach the system boundary.

As illustrated in Figure 3-1, the *span of concern* begins at the boundaries between the system and the user and between the system and the environment, and ends at the fault floor. Faults below the fault floor are indistinguishable, either because they are not fully

understood, or because they are too numerous. Informally, the span of concern is the area within which faults are of interest.

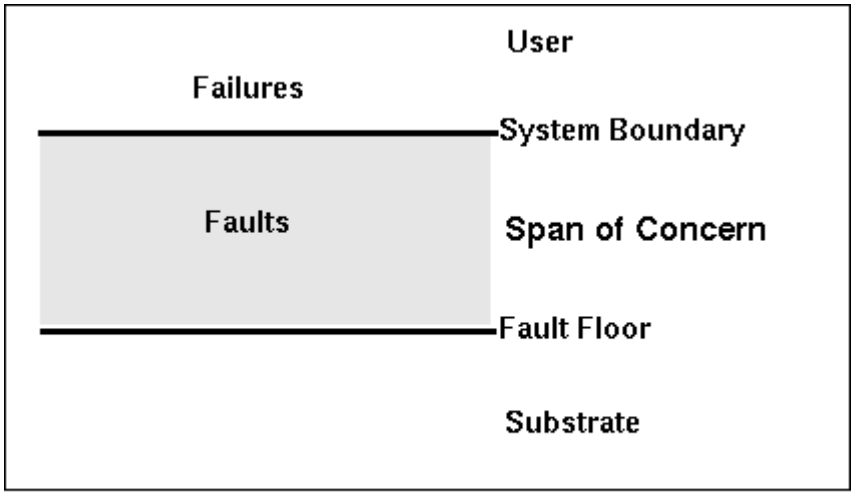


Figure 0-1 The Failure Region

3.3.2.2 Bridge Example

Bridges are designed with the assumption that the structural members used (beams, braces, fasteners) have known load bearing, deformation, and fracture characteristics, which are predicted from knowledge of the composition of the materials, the process used to produce the materials, and from statistical sampling of the materials. Thus the structural members form the fault floor for most bridges. Faults at the molecular level are generally below the level of consideration. The design process for a typical bridge design begins with specification of a certain grade of steel and employs standard structural shapes. The combination of known materials, known shapes, and standard procedures for summing loads and forces is used to predict the failure modes of the overall structure.

3.3.2.3 Computer System Example

In a computer example, a repair person may not care to localize a "problem" to the component level, but instead be satisfied to localize it to the circuit board level. The circuit board represents a fault floor for the repair person. This fault floor is often referred to as a Field Replacable Unit (FRU) or Line Replaceable Unit (LRU). The selection of FRUs and LRUs is an important part of the maintenance strategy for any computer system. The selection is based on considerations such as replacement cost, diagnosis facilities, and skill levels in the field and at repair depots. Notice, however, that when the board is shipped back to the repair depot, they may indeed care about localizing the "problem" down to the component level. In this case the fault floor has changed.

3.4 Fault Classes

No system can be made to tolerate all possible faults, so it is essential that the faults be considered throughout the requirements definition and system design process. However,

it is impractical to enumerate all of the faults to be tolerated; faults must be aggregated into manageable fault classes.

Faults may be classified based on *Locality* (atomic component, composite component, system, operator, environment), on *Effect* (timing, data), or on *Cause* (design, damage). Other possible classification criteria include *Duration* (transient, persistent) and *Effect on System State* (crash, amnesia, partial amnesia, etc.).

Since the location of a fault is so important, fault location is a logical starting point for classifying faults.

3.4.1 Locality

3.4.1.1 Atomic Component Faults

Concept Definition

A *atomic component fault* is a fault at the fault floor, that is, in a component that cannot be subdivided for analysis purposes.

Bridge Example

A fault in an individual structural member in a bridge may be considered a atomic component fault. If the bridge design properly distributes the load among the various structural members (resources) of the bridge, then the load is transferred to other structural members, no failure occurs, and the fault is masked. The fault may be detected by observation of cracks or deformation, or it may remain latent.

Computer System Example

In a computer system, substrate faults can appear in diverse forms. For instance, a fault in a memory bit is not an atomic component fault if the details of the memory are below the current span of concern. Such a fault may or may not appear as a memory fault, depending upon the memory's ability to mask bit faults.

3.4.1.2 Composite Component Faults

Concept Definition

A *composite component fault* is one that arises within an aggregation of atomic components rather than in an atomic component. It may be the result of one or more atomic component faults.

Bridge Example

A pier failure would be an example of a composite component failure for a bridge.

Computer System Example

A disk drive failure in a computer system is an example of a composite component failure. If the individual bits of memory are considered to be in the span of concern, a failure of one of those would be a component failure as well.

3.4.1.3 System Level Faults

Concept Definition

A *system level fault* is one that arises in the structure of a system rather than in the system's components. Such faults are usually interaction or integration faults, that is, they occur because of the way the system is assembled rather than because of the integrity of any individual component. Note that an inconsistency in the operating rules for a system may lead to a system level fault. System level faults also include *operator faults*, in which an operator does not correctly perform his or her role in system operation. Systems that distribute objects or information are prone to a special kind of system fault: *replication faults*. Replication faults occur when replicated information in a system becomes inconsistent, either because replicates that are supposed to provide identical results no longer do so, or because the aggregate of the data from the various replicates is no longer consistent with system specifications. Replication faults can be caused by *malicious* faults, in which components such as processors "lie" by providing conflicting versions of the same information to other components in the system. Malicious faults are sometimes called Byzantine faults after an early formulation of the problem in terms of Byzantine generals trying to reach a consensus on attacking when one of the generals is a traitor [Lamport 82].

Bridge Example

A bridge failure resulting from insufficient allowance for thermal expansion in the overall structure could be considered a system failure: individual structural members behave as specified, but faulty assembly causes failures when they interact. Operator faults have been discussed in the example in Section 3.2.1.

Computer System Example

Consider the computer systems in an automobile. Suppose the airbag deployment computer and the anti-lock brake computer are both known to work properly and yet fail

in operation because one computer interferes with the other when they are both present. This would be a system fault.

3.4.1.4 External Faults

External faults arise from outside the system boundary, the environment, or the user. *Environmental faults* include phenomena that directly affect the operation of the system, such as temperature, vibration, or nuclear or electromagnetic radiation or that affect the inputs provided to the system. *User faults* are created by the user in employing the system. Note that the roles of user and operator are considered separately; the user is considered to be external to the system while the operator is considered to be a part of the system.

3.4.2 Effects

Faults may also be classified according to their effect on the user of the system or service. Since computer system components interact by exchanging data values in a specified time and/or sequence, fault effects can be cleanly separated into timing faults and value faults. Timing faults occur when a value is delivered before or after the specified time. Value faults occur when the data differs in value from the specification.

3.4.2.1 Value Faults

Computer systems communicate by providing values. A *value fault* occurs when a computation returns a result that does not meet the system's specification. Value faults are usually detected using knowledge of the allowable values of the data, possibly determined at run time.

3.4.2.2 Timing Faults

A *timing fault* occurs when a process or service is not delivered or completed within the specified time interval. Timing faults cannot occur if there is no explicit or implicit specification of a deadline. Timing faults can be detected by observing the time at which a required interaction takes place; no knowledge of the data involved is usually needed.

Since time increases monotonically, it is possible to further classify timing faults into early, late, or "never" (omission) faults. Since it is practically impossible to determine if "never" occurs, omission faults are really late timing faults that exceed an arbitrary limit. Systems that never produce value faults, but only fail by omission are called *fail-silent* systems. If all failures require system restart, the system is a *fail-stop* system.

3.4.3 Duration

Persistent faults remain active for a significant period of time. These faults are sometimes termed hard faults. Persistent faults usually are the easiest to detect and diagnose, but may be difficult to contain and mask unless redundant hardware is available. Persistent faults can be effectively detected by test routines that are interleaved with normal processing. *Transient faults* remain active for a short period of time. A transient fault that becomes active periodically is a *periodic fault* (sometimes referred to as an intermittent

fault). Because of their short duration, transient faults are often detected through the faults that result from their propagation.

3.4.4 Immediate Cause

Faults can be classified according to the operational condition that causes them. These include resource depletion, logic faults, or physical faults.

Resource depletion faults occur when a portion of the system is unable to obtain the resources required to perform its task. Resources may include time on a processing or communications device, storage, power, logical structures such as a data structure, or a physical item such as a processor.

Logic faults occur when adequate resources are available, but the system does not behave according to specification. Logic faults may be the result of improper design or implementation, as discussed in the next section. Logic faults may occur in hardware or software.

Physical faults occur when hardware breaks or a mutation occurs in executable software. Most common fault tolerance mechanisms deal with hardware faults.

3.4.5 Ultimate Cause

Faults can also be classified as to their ultimate cause. Ultimate causes are the things that must be fixed to eliminate a fault. These faults occur during the development process and are most effectively dealt with using fault avoidance and fault removal techniques.

A common ultimate cause of a fault is an improper requirements specification which leads to a *specification fault*. Technically this is not a fault, since a fault is defined to be the failure of a component/interacting systems and a failure is the deviation of the system from specification. However, it can be the reason a system deviates from the behavior expected by the user. An especially insidious instance of this arises when the requirements ignore aspects of the environment in which the system operates. For instance, radiation causing a bit to flip in a memory location would be a value fault which would be considered an external fault (Section 3.4.1.4). However, if the fault propagates inside the system boundary the ultimate cause is a specification fault because the system specification did not foresee the problem.

Flowing down the waterfall, a *design fault* results when the system design does not correctly match the requirements, and an *implementation fault* arises when the system implementation does not adequately implement the design. The validation process is specifically designed to detect these faults. Finally, a *documentation fault* occurs when the documented system does not match the real system.

3.5 Other Fault Attributes

3.5.1 Observability

Faults originate in a system component or subsystem, in the system's environment, or in an interaction between the system and a user, operator, or another subsystem. A fault may ultimately have one of several effects:

1. It may disappear with no perceptible effect
2. It may remain in place with no perceptible effect
3. It may lead to a sequence of additional faults that result in a failure in the system's delivered service (propagation to failure)
4. It may lead to a sequence of additional faults with no perceptible effect on the system (undetected propagation)
5. It may lead to a sequence of additional faults that have a perceptible effect on the system but do not result in a failure in the system's delivered service (detected propagation without failure)

Fault detection is usually the first step in fault tolerance. Even if other elements of a system prevent a failure by compensating for a fault, it is important to detect and remove faults to avoid the exhaustion of a systems fault tolerance resources.

3.5.1.1 Concept Definition

A fault is *observable* if there is information about its existence available at the system interface. The information that indicates the existence of a fault is a *symptom*. A symptom may be a directly observed fault or failure, or it may be a change in system behavior such that the system still meets its specifications. A fault that a fault tolerance mechanism of a system has found is said to be *detected*. Otherwise it is *latent*, whether it is observable or not. The definition of detected is independent of whether or not the fault tolerance mechanism is able to successfully deal with the fault condition. For a fault to be detected, it is sufficient that it be known about.

3.5.1.2 Bridge Example

Fault detection in a bridge usually relies on the principle that stress in a structural member results in deformation of the member, which can usually be observed by looking for cracks in the surface or changes in the alignment of the bridge. Note that the fault is not observed directly; rather, its effects are observed. Other faults, such as metal fatigue, can only be predicted by knowing the history of the loads imposed on the member.

A flaw in a structural member of the bridge is a latent fault. If a bridge inspector x-rays the member and discovers the flaw, or observes a crack that is a logical consequence of the flaw, it is a detected fault.

3.5.1.3 Computer System Example

To provide failure-free outputs in a computer-based fault tolerant system, the system must detect faults, a process that requires redundant information (that is, information in addition to the minimum information needed to perform a prescribed function). Redundant information may be combined with a value or it may be stored separately. Such information may include attributes of a value, such as an abstract type; encoded information, such as error correcting code words; and independently calculated reference values. Attribute information is used to verify that the value is being used in the correct context. Codeword information is used to determine if one or a few of the bits in the value have been changed since the value was created. Independently calculated values may be static (for example, a predefined invariant or limit) or they may be dynamically calculated by a reference process. The reference process may be a redundant copy of the primary process, or it may be a diverse implementation that uses a different approach to produce the value being tested. Either time redundancy (retry) or space redundancy (a concurrently executing process) may be used. For instance, a flipped bit in a program is a latent fault. If a checksum is taken, and it does not match a previously computed value, the fault becomes detected, although, in this case, it may only be possible to tell that a fault exists, and not exactly where it is.

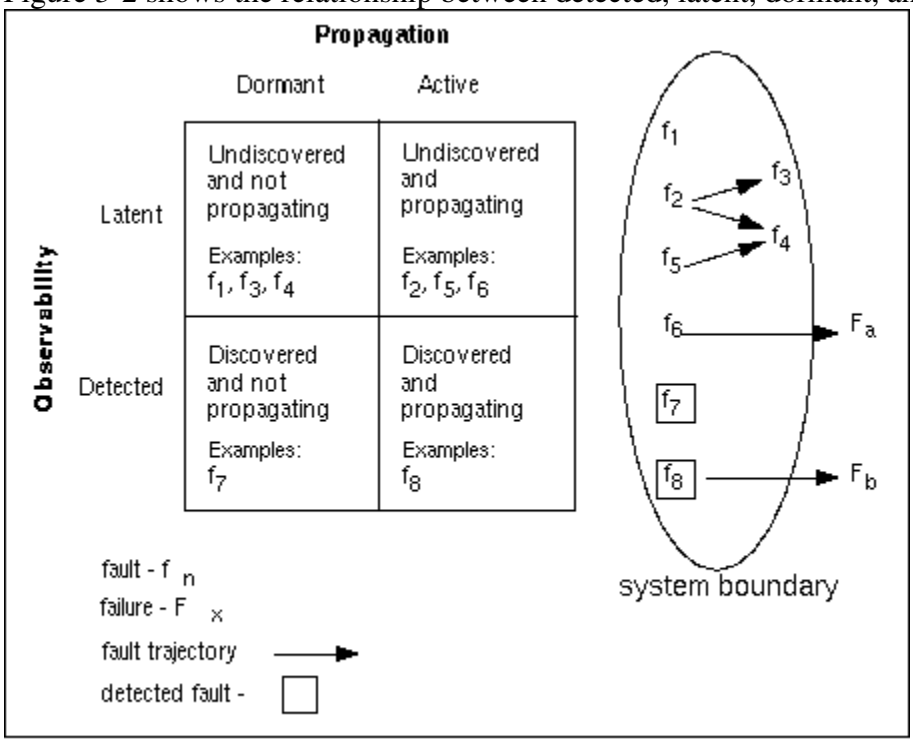
Timing faults may be detected by recognizing the passage of an allotted time interval or by serializing outputs to detect missing outputs. The passage of time may be monitored directly using values from hardware clocks or it may be inferred by noting the completion of one or more processes that complete within a known time interval under normal circumstances.

3.5.2 Propagation

3.5.2.1 Concept Definition

A fault that propagates to other faults or failures is said to be *active*. A non-propagating fault is said to be *dormant*. When a previously dormant fault becomes active it is said to be *triggered*. An active fault may again become dormant, awaiting a new trigger. The sequence of faults, each successive one triggered by the preceding one and possibly ending in a failure, is known as a *fault trajectory*. (Because of the ways faults trigger successive faults, a fault trajectory could be viewed as a chain reaction.)

Figure 3-2 shows the relationship between detected, latent, dormant, and active



Fault Attributes

faults.

3.5.2.2 Bridge Example

Suppose the example bridge was designed to carry 10 ton vehicles over it, but the highway department erects a "load limit 40 tons" sign on the approach. The sign is a dormant fault. It becomes active when a 38 ton truck triggers it by attempting to drive over the bridge and causes the bridge to fall (a failure) or perhaps a structural member to weaken (another fault). The original fault (the sign) becomes dormant again, until another over weight truck drives onto the bridge. The sequence "overweight truck drives over bridge", "structural member weakens" is the fault trajectory.

3.5.2.3 Computer System Example

As another example, consider a computer program loaded in memory, but with a bad bit in one of its instructions. Until that instruction is executed, the fault is dormant. Once it is executed it becomes active and perhaps results in a crash (failure) or a wrong value in a computation (fault). If the value computed was the altitude of an aircraft, and the resulting faulty information led to the plane flying into a mountain, that would be another fault in the fault trajectory (actually a failure in this case).

4 Fault Tolerance Mechanisms

4.1 Characteristics Unique to Digital Computer Systems

Digital computer systems have special characteristics that determine how these systems fail and what fault tolerance mechanisms are appropriate. First, digital systems are discrete systems. Unlike continuous systems, such as analog control systems, they operate in discontinuous steps. Second, digital systems encode information. Unlike continuous systems, values are represented by a series of encoded symbols. Third, digital systems can modify their behavior based on the information they process.

Since digital systems are discrete systems, results may be tested or compared before they are released to the outside world. While analog systems must continuously apply redundant or limiting values, a digital system may substitute an alternative result before sending an output value. While it is possible to build digital computers that operate asynchronously (without a master clock to sequence internal operations), in practice all digital computers are sequenced from a clock signal. This dependency on a clock makes an accurate clock source as important as a source of power, but it also means that identical sequences of instructions take essentially the same amount of time. One of the most common fault tolerance mechanisms, the time-out, uses this property to measure program activity (or lack of activity).

The fact that digital systems encode information is extremely important. The most important implication of information encoding is that digital systems can accurately store information for a long period of time, a capability not available in analog systems, which are subject to value drift. This also means that digital systems can store identical copies of information and expect the stored copies to still be identical after a substantial period of time. This makes the comparison techniques discussed in Section 4.4 possible.

Information encoding in digital systems may be redundant, with several codes representing the same value. Redundant encoding is the most powerful tool available to ensure that information in a digital system has not been changed during storage or transmission. Redundant encoding may be implemented at several levels in a digital system. At the lowest levels, carefully designed code patterns attached to blocks of digital information can allow special-purpose hardware to correct for a number of different communication or storage faults, including changes to single bits or changes to several adjacent bits. Parity for random access memory is a common example of this use of encoding. Since a single bit of information can have significant consequences at the

higher levels, a programmer may encode sensitive information, such as indicators for critical modes, as special symbols unlikely to be created by a single-bit error.

4.2 Redundancy Management

Fault tolerance is sometimes called redundancy management. For our purposes, *redundancy* is the provision of functional capabilities that would be unnecessary in a fault-free environment. Redundancy is necessary, but not sufficient for fault tolerance. For example, a computer system may provide redundant functions or outputs such that at least one result is correct in the presence of a fault, but if the user must somehow examine the results and select the correct one, then the only fault tolerance is being performed by the user. However, if the computer system correctly selects the correct redundant result for the user, then the computer system is not only redundant, but also fault tolerant. Redundancy management marshals the non-faulty resources to provide the correct result.

Redundancy management or fault tolerance involves the following actions:

Fault Detection
: The process of determining that a fault has occurred.

Fault Diagnosis
: The process of determining what caused the fault, or exactly which subsystem or component is faulty.

Fault Containment
: The process that prevents the propagation of faults from their origin at one point in a system to a point where it can have an effect on the service to the user.

Fault Masking
: The process of insuring that only correct values get passed to the system boundary in spite of a failed component.

Fault Compensation
: If a fault occurs and is confined to a subsystem, it may be necessary for the system to provide a response to compensate for output of the faulty subsystem.

Fault Repair
: The process in which faults are removed from a system. In well-designed fault tolerant systems, faults are contained before they propagate to the extent that the delivery of system service is affected. This leaves a portion of the system unusable because of residual faults. If subsequent faults occur, the system may be unable to cope because of this loss of resources, unless these resources are

reclaimed through a recovery process which insures that no faults remain in system resources or in the system state.

The measure of success of redundancy management or fault tolerance is *coverage*. Informally, coverage is the probability of a system failure given that a fault occurs. Simplistic estimates of coverage merely measure redundancy by accounting for the number of redundant success paths in a system. More sophisticated estimates of coverage account for the fact that each fault potentially alters a systems ability to resist further faults. The usual model is a Markov process in which each fault or repair action transitions the system into a new state, some of which are failure states. Because a distinct state is generated for each stage in each possible failure and repair process, Markov models for even simple systems can consist of thousands of states. Sophisticated analysis tools are available to analyze these models and to create the Markov models from more compact system descriptions such as Petri Nets.

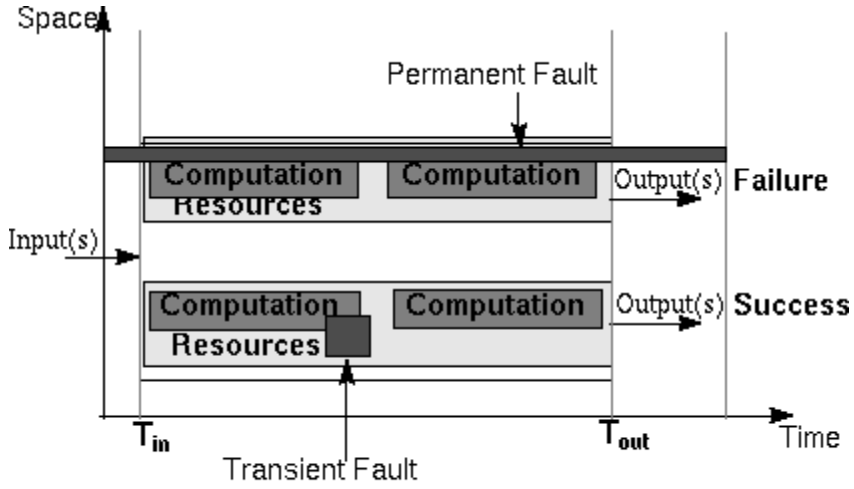
The implementation of the actions described above depend upon the form of redundancy employed such as space redundancy or time redundancy.

4.2.1 Space Redundancy

Space redundancy provides separate physical copies of a resource, function, or data item. Since it has been relatively easy to predict and detect faults in individual hardware units, such as processors, memories, and communications links, space redundancy is the approach most commonly associated with fault tolerance. It is effective when dealing with persistent faults, such as permanent component failures. Space redundancy is also the approach of choice when fault masking is required, since the redundant results are available simultaneously. The major concern in managing space redundancy is the elimination of failures caused by a fault to a function or resource that is common to all of the space-redundant units. This is discussed in more detail in Section 4.2.5.

4.2.2 Time Redundancy

As mentioned before, digital systems have two unique advantages over other types of systems, including analog electrical systems. First, they can shift functions in time by storing information and programs for manipulating information. This means that if the expected faults are transient, a function can be rerun with a stored copy of the input data at a time sufficiently removed from the first execution of the function that a transient fault would not affect both. Second, since digital systems encode information as symbols, they can include redundancy in the coding scheme for the symbols. This means that information shifted in time can be checked for unwanted changes, and in many cases, the information can be corrected to its original value. Figure 4-1 illustrates the relationship between time and space redundancy



Time and Space Redundancy

. The two sets of resources represent space redundancy and the sequential computations represent time redundancy. In the figure, time redundancy is not capable of tolerating the permanent fault in the top processing resource, but is adequate to tolerate the transient fault in the lower resource. In this simple example, there is still the problem of recognizing the correct output: this is discussed in more detail in Sections 4.3 and 4.4.

4.2.3 Clocks

Many fault tolerance mechanisms, employing either space redundancy or time redundancy, rely on an accurate source of time. Probably no hardware feature has a greater effect on fault tolerance mechanisms than a clock. An early decision in the development of a fault tolerant system should be the decision to provide a reliable time service throughout the system. Such a service can be used as a foundation for fault detection and repair protocols. If the time service is not fault tolerant, then additional interval timers must be added or complex asynchronous protocols must be implemented that rely on the progress of certain computations to provide an estimate of time. Multiple-processor system designers must decide to provide a fault tolerant global clock service that maintains a consistent source of time throughout the system, or to resolve time conflicts on an ad-hoc basis [Lamport 85].

4.2.4 Fault Containment Regions

Although it is possible to tailor fault containment policies to individual faults, it is common to divide a system into *fault containment regions* with few or no common dependencies between regions.

Fault containment regions attempt to prevent the propagation of data faults by limiting the amount of communication between regions to carefully monitored messages and the propagation of resource faults by eliminating shared resources. In some ultra-dependable designs, each fault containment region contains one or more physically and electrically isolated processors, memories, power supplies, clocks, and communication links. The only resources that are tightly coordinated in such architectures are clocks, and extensive precautions are taken to insure that clock synchronization mechanisms do not allow faults

to propagate between regions. Data fault propagation is inhibited by locating redundant copies of critical programs in different fault containment regions and by accepting data from other copies only if multiple copies independently produce the same result.

4.2.5 Common Mode Failures

System failures occur when faults propagate to the outer boundary of the system. The goal of fault tolerance is to intercept the propagation of faults so that failure does not occur, usually by substituting redundant functions for functions affected by a particular fault. Occasionally, a fault may affect enough redundant functions that it is not possible to reliably select a non-faulty result, and the system will sustain a *common-mode failure*. A common-mode failure results from a single fault (or fault set). Computer systems are vulnerable to common-mode resource failures if they rely on a single source of power, cooling, or I/O. A more insidious source of common-mode failures is a design fault that causes redundant copies of the same software process to fail under identical conditions.

4.2.6 Encoding

Encoding is the primary weapon in the fault tolerance arsenal. Low-level encoding decisions are made by memory and processor designers when they select the error detection and correction mechanisms for memories and data buses. Communications protocols provide a variety of detection and correction options, including the encoding of large blocks of data to withstand multiple contiguous faults and provisions for multiple retries in case error correcting facilities cannot cope with faults. Long-haul communication facilities even provide for a negotiated fall-back in transmission speed to cope with noisy environments. These facilities should be supplemented with high-level encoding techniques that record critical system values using unique patterns that are unlikely to be randomly created.

4.3 Acceptance Test Techniques

The fault detection mechanism used influences the remainder of the fault tolerance activities (diagnosis, containment, masking, compensation, and recovery). The two common mechanisms for fault detection are *acceptance tests* and *comparison*.

4.3.1 Fault Detection

Acceptance tests are the more general fault detection mechanism in that they can be used even if the system is composed of a single (non-redundant) processor. The program or sub-program is executed and the result is subjected to a test. If the result passes the test,

execution continues normally. A failed acceptance test is a symptom of a fault. An acceptance test is most effective if it is based on criteria that can be derived independently of the function being tested and can be calculated more simply that the function being tested (e.g., multiplication of a result by itself to verify the result of a square root function).

4.3.2 Fault Diagnosis

An acceptance test cannot generally be used to determine what has gone wrong. It can only tell that something has gone wrong.

4.3.3 Fault Containment

An acceptance test provides a barrier to the continued propagation of a fault. Further execution of the program being tested is not allowed until some form of retry successfully passes the acceptance test. If no alternatives pass the acceptance test, the subsystem fails, preferably silently. The silent failure of faulty components allows the rest of the system to continue in operation (where possible) without having to worry about erroneous output from the faulty component [Schlichting 83].

4.3.4 Fault Masking

An acceptance test successfully masks a bad value if a retry or alternate results in a new, correct result within the time limit set for declaring failure.

4.3.5 Fault Compensation

A program that fails an acceptance test can be replaced by an alternate, as described in the next section. If the alternate passes the acceptance test, its result may be used to compensate for the original result. Notice that the alternate program run during a retry may be a very simple one that just outputs a "safe" value to compensate for the faulty subsystem. A common approach in control systems is to "coast" the result by providing the value calculated from the last known good cycle.

4.3.6 Fault Repair

Acceptance tests are usually used in a construct known as a *recovery block*. A recovery block provides backward fault recovery by rolling program execution back to the state before the faulty function was executed. This repairs the faulty state and the result. When a result fails an acceptance test, the program can be executed again before leaving the recovery block. If the new result passes the acceptance test, it can be assumed that the fault originally detected was transient. If the software is suspect, an alternative can be executed in place of the original program fragment. If a single processor is used, the state of the processor must be reset to the beginning of the function in question. A mechanism called the *recovery cache* has been proposed to accomplish this [Anderson 76]. A recovery cache records the state of the processor at the entrance to each recovery block. Although a recovery cache is best implemented in hardware, implementations to date have been limited to experimental software. Where multiple processors are available, the retry may take the form of starting the program on a backup processor and shutting down

the failed processor. Recovery blocks can be cascaded so that multiple alternatives can be tried when an alternate result also fails the acceptance test.

4.4 Comparison Techniques

4.4.1 Fault Detection

Comparison is an alternative to acceptance tests for detecting faults. If the principal fault source is processor hardware, then multiple processors are used to execute the same program. As results are calculated, they are compared across processors. A mismatch indicates the presence of a fault. This comparison can be pair-wise, or it may involve three or more processors simultaneously. In the latter case the mechanism used is generally referred to as *voting*. If software design faults are a major consideration, then a comparison is made between the results from multiple versions of the software in question, a mechanism known as n-version programming [Chen 78]. This is discussed more in the Section 4.5.

4.4.2 Fault Diagnosis

Fault diagnosis with comparison is dependent upon whether pair-wise or voting comparison is used:

pair-wise
: When a mismatch occurs for a pair it is impossible to tell which of the processors has failed. The entire pair must be declared faulty.

voting
: When three or more processors are running the same program, the processor whose values do not match the others is easily diagnosed as the faulty one.

4.4.2.1 Voting Issues

Voting may be centralized or decentralized. Centralized voting is easy to mechanize, either in software or hardware, but results in a single point of failure, a violation of many qualitative requirements specifications. It is possible to compensate for total voter failure using a master-slave approach that replaces a silent voter with a standby voter, as in the pair and spare approach. Decentralized voting avoids the single point of failure, but requires a consensus among multiple voting agents, either hardware or software in order to avoid replication faults mentioned in Section 3.4.1.3. In order to reach consensus, the distributed voters must synchronize to exchange several rounds of messages. In the worst case, where up to f faulty processors are allowed to send misleading results to other processors participating in the consensus process, $3f+1$ distributed voters must be

provided to reach a state known as interactive consistency [Pease 80]. Interactive consistency requires that each non-faulty processor provides a value, that all non-faulty processors agree on the same set of values, and that the values are correct for each of the non-faulty processors. Similar processes are required to maintain a consensus as to the number of members remaining in a group of distributed processors [Cristian 88].

4.4.3 Fault Containment

When pair-wise comparison is used, containment is achieved by stopping all activity in the mismatching pair. Any other pairs in operation can continue executing the application, undisturbed. They detect the failure of the miscomparing pair through time-outs.

When voting is used, containment is achieved by ignoring the failed processor and reconfiguring it out of the system.

4.4.4 Fault Masking

In a comparison based system, fault masking is achievable in two ways. When voting is used the voter only allows the correct value to pass on. If hardware voters are used, this usually occurs quickly enough to meet any response deadlines. If the voting is done by software voters that must reach a consensus, adequate time may not be available.

Pair-wise comparison requires the existence of multiple pairs of processors to mask faults. In this case the faulty pair of processors is halted, and values are obtained from the functional, good pairs.

4.4.5 Fault Compensation

The value provided by a voter may be the majority value, the median value, a plurality value, or some other predetermined satisfactory value. While this choice is application dependent, the most common choice is the median value. This guarantees that the value selected was calculated by at least one of the participating processors and that it is not an extreme value.

4.4.6 Fault Repair

In a comparison-based system with a single pair of processors, there is no recovery from a fault. With multiple pairs of pairs, recovery consists of using the values from the "good" pair. Some systems provide mechanisms to restart the miscomparing pair with data from a "good" pair. If the miscomparing pair subsequently produces results that compare for an adequate period of time, it may be configured back into the system.

When voting is used, recovery from a failed processor is accomplished by utilizing the "good" values from the other processors. A processor that is outvoted may be allowed to continue execution and may be configured back into the system if it successfully matches in a specified number of subsequent votes.

4.5 Diversity

The only fault tolerance approach for combating common-mode design errors is design diversity--the implementation of more than one variant of the function to be performed [Avizienis 84]. For computer-based applications, it is generally accepted that it is more effective to vary a design at higher levels of abstraction (i.e., by varying the algorithm or physical principles used to obtain a result) than to vary implementation details of a design (i.e. by using different programming languages or low level coding techniques). Since diverse designs must implement a common system specification, the possibility for dependencies always arises in the process of refining the specification to reflect difficulties uncovered in the implementation process. Truly diverse designs would eliminate dependencies on common design teams, design philosophies, software tools and languages, and even test philosophies. Many approaches attempt to achieve the necessary independence through randomness, by creating separate design teams that remain physically separate throughout the design, implementation, and test process. Recently, some projects have attempted to create diversity by enforcing differing design rules for the multiple teams.

---

5 Putting It All Together

This document has attempted to present a conceptual framework of system fault tolerance. The previous discussion has been centered around definitions and examples. This section discusses how to use the information in the prior sections.

A system is said to have a *failure* if the service it delivers to the user deviates from compliance with the system *specification*. A *fault* is the adjudged cause of a failure. The significance of this is that, in the absence of precise requirements, it is impossible to tell whether a system has failed, and therefore whether a fault has occurred.

Rule 1: Know precisely what the system is supposed to do. Part of this process should be determining how long a system can be allowed to deviate from its specification before the deviation is declared a failure.
However, it is not sufficient to know what the system is supposed to do under normal circumstances. It is also necessary to know what abnormal conditions the system must accommodate. It is virtually impossible to enumerate the set of all possible faults that a system might encounter. It is much more manageable to deal with classes of faults.

Rule 2: Look at what can go wrong, and try to group the causes into classes for easier manageability. This involves defining a fault floor based on your ability to diagnose and repair faults.
The goal of fault tolerance is to prevent faults from propagating to the system boundary, where it becomes observable and, hence, a failure. In general, the further a fault has propagated, the harder it is to deal with. Since fault tolerance is redundancy management, however, it becomes a matter of the degree of redundancy desired. For instance, it is

almost certainly cheaper to deal with memory faults by using error correcting memory (that is, redundant bits in a memory location) than by providing a "shadow" memory. Note, however, that dealing with faults earlier rather than later may go counter to the advice given above regarding dealing with classes of faults rather than individual faults.

Rule 3: Study your application and determine appropriate fault containment regions and the earliest feasible time to deal with potential faults.
In general, the price paid for a fault tolerant system is additional resources, both in terms of time, and in terms of space. As with most things these two can be traded off against each other. In some applications (e.g., flight control), timing is everything, even at the cost of extra processors. In general, the comparison approach to fault detection works best in these situations. In other applications (e.g., a space probe), weight and power consumption is an overriding issue--arguing for a higher reliance on time redundancy and suggesting the use of acceptance tests.

Rule 4: Completely understand the requirements of your application and use them to make appropriate time/space trade-offs.
Protecting a system from every conceivable fault can exhaust another resource--money. This is true even if a rational set of fault classes is defined. The trade-off here is fault coverage versus the cost of that coverage. In all systems, it is possible to classify faults by the likelihood of occurrence.

Rule 5: Whenever possible, concentrate on the credible faults and ignore those less likely to occur unless they can be dealt with at little or no additional cost.
Time is an essential element in any digital computer system, even in systems that do not claim to be real-time. It is important to define the minimum period of time a system can fail to provide its defined service before a failure is declared. Unnecessarily short failure margins force the system designer to resort to expensive fault tolerance mechanisms, such as real-time fault masking.

Rule 6: Carefully determine application failure margins and use the information to balance the degree of fault tolerance needed with the cost of implementation.
