High Availability Computer Systems

Jim Gray
Digital Equipment Corporation
455 Market St., 7'th Floor
San Francisco, CA. 94105

Daniel P. Siewiorek
Department of Electrical Engineering
Carnegie Mellon University
Pittsburgh, PA. 15213

Abstract: *The key concepts and techniques used to build high availability computer systems are (1) modularity, (2) fail-fast modules, (3) independent failure modes, (4) redundancy, and (5) repair. These ideas apply to hardware, to design, and to software. They also apply to tolerating operations faults and environmental faults. This article explains these ideas and assesses high-availability system trends.*

Overview

It is paradoxical that the larger a system is, the more critical is its availability, and the more difficult it is to make it highly-available. It is possible to build small *ultra-available modules*, but building large systems involving thousands of modules and millions of lines of code is still an art. These large systems are a core technology of modern society, yet their availability are still poorly understood.

This article sketches the techniques used to build highly available computer systems. It points out that three decades ago, hardware components were the major source of faults and outages. Today, hardware faults are a minor source of system outages when compared to operations, environment, and software faults. Techniques and designs that tolerate this broader class of faults are in their infancy.

A Historical Perspective

Computers built in the late 1950's offered twelve-hour mean time to failure. A maintenance staff of a dozen full-time customer engineers could repair the machine in about eight hours. This failure-repair cycle provided 60% availability. The vacuum tube and relay components of these computers were the major source of failures; they had lifetimes of a few months. Therefore, the machines rarely operated for more than a day without interruption[1].

Many fault detection and fault masking techniques used today were first used on these early computers. *Diagnostics* tested the machine. *Self-checking* computational techniques detected faults while the computation progressed. The program occasionally saved (checkpointed) its state on stable media. After a failure, the program read the most recent checkpoint, and

continued the computation from that point. This *checkpoint/restart* technique allowed long-running computations to be performed by machines that failed every few hours.

Device improvements have improved computer system availability. By 1980, typical well-run computer systems offered 99% availability[2]. This sounds good, but 99% availability is 100 minutes of downtime per week. Such outages may be acceptable for commercial back-office computer systems that process work in asynchronous batches for later reporting. Mission critical and online applications cannot tolerate 100 minutes of downtime per week. They require *high-availability* systems – ones that deliver 99.999% availability. This allows at most five minutes of service interruption per year.

Process control, production control, and transaction processing applications are the principal consumers of the new class of high-availability systems. Telephone networks, airports, hospitals, factories, and stock exchanges cannot afford to stop because of a computer outage. In these applications, outages translate directly to reduced productivity, damaged equipment, and sometimes lost lives.

Degrees of availability can be characterized by orders of magnitude. Unmanaged computer systems on the Internet typically fail every two weeks and average ten hours to recover. These unmanaged computers give about 90% availability. Managed conventional systems fail several times a year. Each failure takes about two hours to repair. This translates to 99% availability[2]. Current fault-tolerant systems fail once every few years and are repaired within a few hours[3]. This is 99.99% availability. High-availability systems require fewer failures and faster repair. Their requirements are one to three orders-of-magnitude more demanding than current fault-tolerant technologies (see Table 1).

Table 1. Availability of typical systems classes. Today's best systems are in the high-availability range. The best of the general-purpose systems are in the fault-tolerant range as of 1990.

System Type	Unavailability (min/year)	Availability	Availability Class
unmanaged	50,000	90.%	1
managed	5,000	99.%	2
well-managed	500	99.9%	3
fault-tolerant	50	99.99%	4
high-availability	5	99.999%	5
very-high-availability	.5	99.9999%	6
ultra-availability	.05	99.99999%	7

As the nines begin to pile up in the availability measure, it is better to think of availability in terms of denial-of-service measured in minutes per year. So for example, 99.999% availability is about 5 minutes of service denial per year. Even this metric is a little cumbersome, so the concept of *availability class* or simply *class* is defined, by analogy to the hardness of diamonds or the class of a cleanroom. Availability class is the number of leading nines in the availability figure for a system or module. More formally, if the system availability is *A*, the system's availability class is $e^{\log_{10}(\frac{1}{1-A})}$. The rightmost column of Table 1 tabulates the availability classes of various system types.

The telephone network is a good example of a high-availability system - a class 5 system. Its design goal is at most two outage hours in forty years. Unfortunately, over the last two years there have been several major outages of the United States telephone system – a nation-wide outage lasting eight hours, and a mid-west outage lasting four days. This shows how difficult it is to build systems with high-availability.

Production computer software typically has more than one defect per thousand lines of code. When millions of lines of code are needed, the system is likely to have thousands of software defects. This seems to put a ceiling on the size of high-availability systems. Either the system must be small or it must be limited to a failure rate of one fault per decade. For example, the ten-million line Tandem system software is measured to have a thirty-year failure rate[3].

High availability requires systems designed to *tolerate* faults -- to detect the fault, report it, mask it, and then continue service while the faulty component is repaired offline. Beyond the prosaic hardware and software faults, a high-availability system must tolerate the following sample faults:

Electrical power at a typical site in North America fails about twice a year. Each failure lasts about an hour[4].

Software upgrades or repair typically require interrupting service while installing new software. This happens at least once a year and typically takes an hour.

Database Reorganization is required to add new types of information to the database, to reorganize the data so that it can be more efficiently processed, or to redistribute the data among recently added storage devices. Such reorganizations may happen several times a year and typically take several hours. As of 1991, no general-purpose system provides complete online reorganization utilities.

Operations Faults: Operators sometimes make mistakes that lead to system outages. Conservatively, a system experiences one such fault a decade. Such faults cause an outage of a few hours.

Just the four fault classes listed above contribute more than 1000 minutes of outage per year. This explains why managed systems do worse than this and why well managed systems do slightly better (see Table 1).

High availability systems must mask most of these faults. One thousand minutes is much more than the five-minute per year budget allowed for high-availability systems. Clearly it is a matter of degree -- not *all* faults can be tolerated. Ignoring scheduled interruptions to upgrade software to newer versions, current fault-tolerant systems typically deliver four years of uninterrupted service and then require a two-hour repair[3]. This translates to 99.96% availability -- about one minute outage per week.

This article surveys the fault-tolerance techniques used by these systems. It first introduces terminology. Then it surveys design techniques used by fault-tolerant systems. Finally, it sketches approaches to the goal of ultra-available systems, systems with a 100-year mean-time-to-failure rate and a one-minute mean-time-to-repair.

Terminology

Fault-tolerance discussions benefit from terminology and concepts developed by IFIP Working Group (IFIP WG 10.4) and by the IEEE Technical Committee on Fault-tolerant Computing. The result of those efforts is very readable[5]. The key definitions are repeated here.

A system can be viewed as a single module. Most systems are composed of multiple modules. These modules have internal structure, being in turn composed of sub-modules. This presentation discusses the behavior of a single module, but the terminology applies recursively to modules with internal modules.

Each module has an ideal *specified behavior* and an observed *actual behavior*. A *failure* occurs when the actual behavior deviates from the specified behavior. The failure occurred because of an *error* -- a defect in the module. The cause of the error is a *fault*. The time between the occurrence of the error and the resulting failure is the *error latency*. When the error causes a failure, it becomes *effective* (see Figure 1).

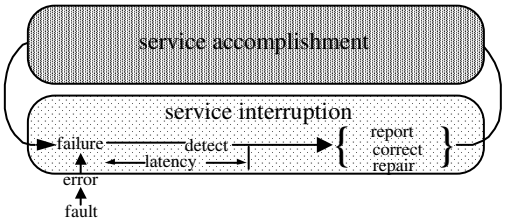

Figure 1. Usually a module's *observed behavior* matches its *specified behavior*. Therefore, it is in the *service accomplishment state*. Occasionally, a *fault* causes an *error* that eventually becomes *effective* causing the module to *fail* (observed behavior does not equal specified behavior). Then the module enters the *service interruption state*. The failure is detected, reported, corrected or repaired, and then the module returns to the service accomplishment state.

For example, a programmer's mistake is a *fault*. It creates a *latent error* in the software. When the erroneous instructions are executed with certain data values, they cause a *failure* and the error becomes *effective*. As a second example, a cosmic ray (*fault*) may discharge a memory cell causing a memory *error*. When the memory is read, it produces the wrong answer (memory *failure*) and the error becomes *effective*.

The actual module behavior alternates between *service-accomplishment* while the module acts as specified, and *service interruption* while the module behavior deviates from the specified behavior. *Module reliability* measures the time from an initial instant and the next failure event. In a population of identical modules that are run until failure, the *mean-time-to-failure* is the average time to failure over all modules. Module reliability is statistically quantified as *mean-time-to-failure (MTTF)*. Service interruption is statistically quantified as *mean-time-to-repair (MTTR)*. *Module availability* measures the ratio of service-accomplishment to elapsed time. The availability of non-redundant systems with repair is statistically quantified as $\frac{MTTF}{MTTF+MTTR}$.

Module reliability can be improved by reducing failures. Failures can be avoided by *valid construction* and by *error correction*.

Validation can remove errors during the construction process, thus assuring that the constructed module conforms to the specified module. Since physical components fail during operation, validation alone cannot assure high reliability or high availability.

Error Correction reduces failures by tolerating faults with redundancy.

Latent error processing tries to detect and repair latent errors before they become effective. Preventive maintenance is an example of latent error processing.

Effective error processing tries to correct the error after it becomes effective. Effective error processing may either *recover* from the error or *mask* the error.

Error masking typically uses redundant information to deliver the correct service and to construct a correct new state. Error Correcting Codes (ECC) used for electronic, magnetic, and optical storage are examples of masking.

Error recovery typically denies the request and sets the module to an error-free state so that can service subsequent requests. Error recovery can take two forms

Backward error recovery returns to a previous correct state. Checkpoint/restart is an example of backward error recovery.

Forward error recovery constructs a new correct state. Redundancy in time, for example resending a damaged message or rereading a disc block are examples of forward error recovery.

These are the key definitions from the IFIP Working Group[5]. Some additional terminology is useful. Faults are typically categorized as:

Hardware faults -- failing devices,

Design faults -- faults in software (mostly) and hardware design,

Operations faults -- mistakes made by operations and maintenance personnel, and

Environmental faults - fire, flood, earthquake, power failure, sabotage.

Empirical Experience

There is considerable empirical evidence about faults and fault tolerance[6]. Failure rates (or failure hazards) for software and hardware modules typically follow a *bathtub curve*: The rate is high for new units (*infant mortality*), then it stabilizes at a low rate. As the module ages beyond a certain threshold the failure rate increases (*maturity*). Physical stress, decay, and corrosion are the source of physical device aging. Maintenance and redesign are sources of software aging.

Failure rates are usually quoted at the bottom of the bathtub (after infant mortality and before maturity). Failures often obey a Weibull distribution, a negative hyper-exponential distribution. Many device and software failures are transient -- that is the operation may succeed if the device or software system is simply reset. Failure rates typically increase with utilization. There is evidence that hardware and software failures tend to occur in clusters.

Repair times for a hardware module can vary from hours to days depending on the availability of spare modules and diagnostic capabilities. For a given organization, repair times appear to follow a Poisson distribution. Good repair success rates are typically 99.9%, but 95% repair success rates are common. This is still excellent compared to the 66% repair success rates reported for automobiles.

Improved Devices are Half the Story

Device reliability has improved enormously since 1950. Vacuum tubes evolved to transistors. Transistors, resistors, and capacitors were integrated on single chips. Today, packages integrate millions of devices on a single chip. These device and packaging revolutions have many reliability benefits for digital electronics:

More Reliable Devices: Integrated-circuit devices have long lifetimes. They can be disturbed by radiation, but if operated at normal temperatures and voltages, and kept from corrosion, they will operate for at least 20 years.

Reduced Power: Integrated circuits consume much less power per function. The reduced power translates to reduced temperatures and slower device aging.

Reduced Connectors: Connections were a major source of faults due to mechanical wear and corrosion. Integrated circuits have fewer connectors. On-chip connections are chemically deposited, off chip connections are soldered, and wires are printed on circuit boards. Today, only backplane connections suffer mechanical wear. They interconnect field replaceable units (modules) and peripheral devices. These connectors remain a failure source.

Similar improvements have occurred for magnetic storage devices. Originally, discs were the size of refrigerators and needed weekly service. Just ten years ago, the typical disc was the size of a washing machine, consumed about 2,000 watts of power, and needed service about every six months. Today, discs are hand-held units, consume about 10 watts of power, and have no scheduled service. A modern disc becomes obsolete sooner than it is likely to fail. The MTTF of a modern disc is about 12 years; its useful life is probably five years.

Peripheral device cables and connectors have experienced similar complexity reductions. A decade ago, disc cables were huge. Each disc required 20 or more control wires. Often discs were dual-ported which doubled this number. An array of 100 discs needed 4,000 wires and 8,000 connectors. As in the evolution of digital electronics, these cables and their connectors were a major source of faults. Today, modern disc assemblies use fiber-optic cables and connectors. A 100-disc array can be attached with 24-cables and 48 connectors: more than a 100-fold component reduction. In addition, the underlying media uses lower power and have better resistance to electrical noise.

Fault-tolerant Design Concepts

Fault-tolerant system designs use the following basic concepts:

Modularity: Decompose the system into modules. The decomposition is typically hierarchical. For example, a computer may have a storage module that in turn has several memory modules. Each module is a unit of service, fault containment, and repair.

Service: The module provides a well specified interface to some function.

Fault containment: If the module is faulty, the design prevents it from contaminating others.

Repair: When a module fails it is replaced by a new module.

Fail-Fast: Each module should either operate correctly or should stop immediately.

Independent Failure Modes: Modules and interconnections should be designed so that if one module fails, the fault should not also affect other modules.

Redundancy and Repair: By having spare modules already installed or configured, when one module fails the second can replace it almost instantly. The failed module can be repaired offline while the system continues to deliver service.

These principles apply to hardware faults, design faults, and software faults (which are design faults). Their application varies though, so hardware is treated first, and then design and software faults are discussed.

Fault-Tolerant Hardware

The application of the modularity, fail-fast, independence, redundancy, and repair concepts to hardware fault-tolerance is easy to understand. Hardware modules are physical units like a processor, a communications line, or a storage device. A module is made fail-fast by one of two techniques[6,7,8]:

Self-checking: A module performs the operation and also performs some additional work to validate the state. Error detecting codes on storage and messages are examples of this.

Comparison: Two or more modules perform the operation and a comparitor examines their results. If they disagree, the modules stop.

Self-checking has been the mainstay for many years, but it requires additional circuitry and design. Self-checking will likely continue to dominate the storage and communications designs because the logic is simple and well understood.

The economies of integrated circuits encourage the use of comparison for complex processing devices. Because comparators are relatively simple, comparison trades additional circuits for reduced design time. In custom fault-tolerant designs, 30% of processor circuits and 30% of the processor design time are devoted to self-checking. Comparison schemes augment general-purpose circuits with simple comparitor designs and circuits. The result is a reduction in overall design cost and circuit cost.

The basic comparison approach is depicted in Figure 2.A. It shows how a relatively simple comparator placed at the module interface can compare the outputs of two modules. If the

outputs match exactly, the comparator lets the outputs pass through. If the outputs do not match, the comparator detects the fault and stops the modules. This is a generic technique for making fail-fast modules from conventional modules.

If more than two modules are used, the module can tolerate at least one fault because the comparator passes through the majority output (two out of three in Figure 2.A). The triplex design is called *tripple-module-redundancy (TMR).* The idea generalizes to N-plexed modules.

As shown in Figure 2.B, comparison designs can be made recursive. In this case, the comparators themselves are N-plexed so that comparator-failures are also detected. Self-checking and comparison provide quick fault detection. Once a fault is detected it should be *reported*, and then *masked* as in Figure 1.

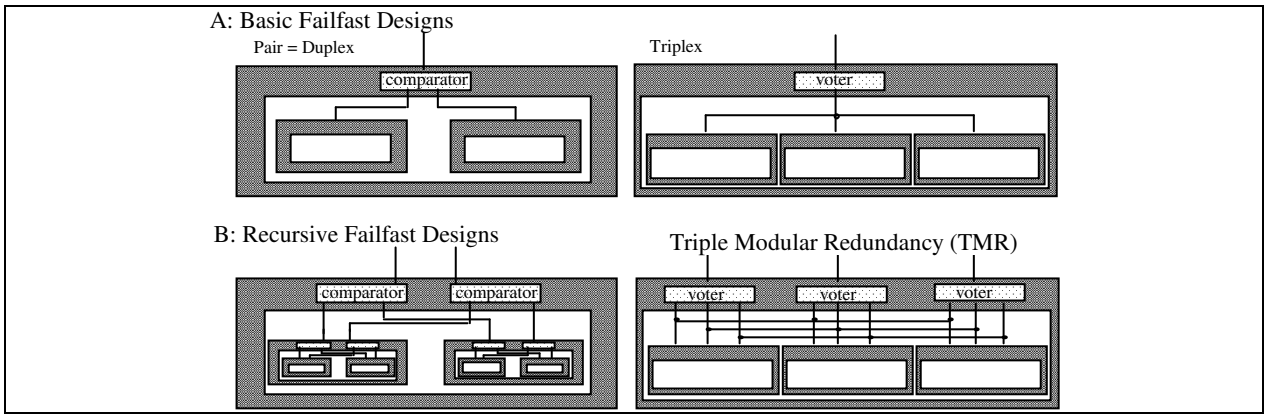

Figure 2: The basic approaches to designing fail-fast and fault-tolerant modules.

Hardware fault masking with comparison schemes typically work as in Figure 3. The duplexing scheme (*pair-and-spare* or *dual-dual*) combines two fail-fast modules to produce a super module that continues operating even if one of the submodules fails. Since each submodule is fail-fast, the combination is just the OR of the two submodules. The triplexing scheme masks failures by having the comparitor pass through the majority output. If only one module fails, the outputs of the two correct modules will form a majority and so will allow the supermodule to function correctly.

The pair-and-spare scheme costs more hardware (four rather than three modules), but allows a choice of two operating modes: either two independent fail-fast computations running on the two pairs of modules or a single high-availability computation running on all four modules.

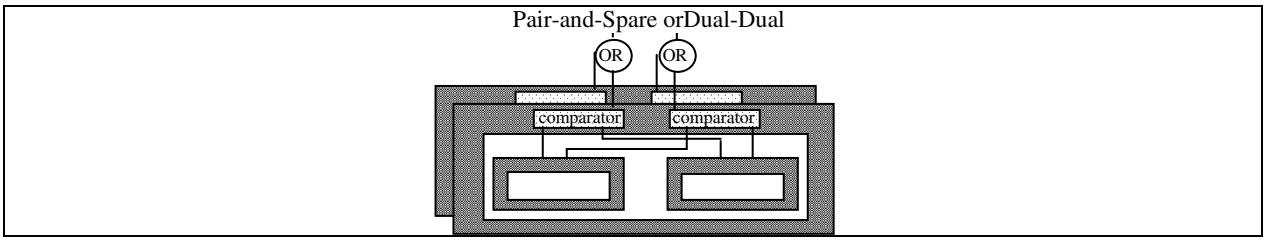

Figure 3: Using redundancy to mask failures. TMR needs no extra effort to mask a single fault. Duplexed modules can tolerate faults by using a pair-and-spare or dual-dual design. If any single module fails, the super module continues operating.

To understand the benefits of these designs, imagine that each module has a one-year MTTF, with independent failures. Suppose that the duplex system fails if the comparitor inputs do not agree, and the triplex module fails if two of the module inputs do not agree. If there is no repair, the super-modules in Figure 2 will have a MTTF of less than a year (see Table 2). This is an instance of the airplane rule: a two-engine airplane costs twice as much and has twice as many engine problems as a one-engine airplane. Redundancy by itself does not improve availability or reliability (redundancy does decrease the variance in failure rates). In fact, adding redundancy made the reliability *worse* in these two cases. *Redundancy designs require repair to dramatically improve availability.*

The Importance of Repair

If failed modules are *repaired* (replaced) within four hours of their failure, then the MTTF of the example systems goes from one year MTTF to well beyond 1,000 year MTTF. Their availability goes from 99.9% to 99.9999% (from availability class 3 to class 6). That is a significant improvement. If the system employs thousands of modules, the construction can be repeated recursively to N-plex the entire system and get a class 8 super-module (1,000 year MTTF).

Online module repair requires the ability to repair and reinstall modules while the system is operating. It also requires re-integrating the module into the system without interrupting service. Doing this is not easy. For example, when a disc is repaired, it is not trivial to make the contents of the disc identical to a neighboring disc. Reintegration algorithms exist, but they are subtle. Each seems to use a different trick. There is no overall design methodology for them yet. Similarly, when a processor is repaired, it is not easy to set the processor state to that of the other processors in the module. Today, online integration techniques are an area of patents and trade secrets. They are a key to high-availability computing.

Table 2: MTTF estimates for various architectures using 1 year module MTTF modules with 4-hour MTTR. The letter ε represents a small additional cost for the comparators. (see Siewiroek[6] derivations).

	MTTF	CLASS	EQUATION	COST
SIMPLEX	1 year	3	MTTF	1
DUPLEX	~0.5 years	3	≈MTTF/2	2+ε
TRIPLEX	.8 year	3	≈MTTF(5/6)	3+ε
PAIR AND SPARE	~.7 year	3	≈MTTF(3/4)	4+ε
DUPLEX + REPAIR	$>10^3$ years	6	$\approx MTTF^2/2MTTR$	4+ε
TRIPLEX + REPAIR	$>10^6$ years	6	$\approx MTTF^3/3MTTR$	3+ε

The simple and powerful ideas of *fail-fast modules* and *repair* via retry or by spare modules seem to solve the hardware fault-tolerance problem. They can mask almost all physical device failures. They do not mask failures caused by hardware design faults. If all the modules are faulty by design, then the comparators will not detect the fault. Similarly, comparison techniques do not seem to apply to software, which is all design, unless design diversity is employed. The next section discusses techniques that tolerate design faults.

Improved Device Maintenance: the FRU Concept

The declining cost and improved reliability of devices allow a new approach to computer maintenance. Today computers are composed of modules called *field-replaceable-units* (FRUs). Each FRU has *built-in self-tests* exploiting one of the checking techniques mentioned above. These tests allow a module to diagnose itself, and report failures. These failures are reported electronically to the system maintenance processor, and are reported visually as a green-yellow-red light on the module itself: green means no trouble, yellow means a fault has been reported and masked, and red indicates a failed unit. This system makes it easy to perform repair. The repair person looks for a red light and replaces the failed module with a spare from inventory.

FRUs are designed to have a MTTF in excess of ten years. They are designed to cost less than a few thousand dollars so that they may be manufactured and stocked in quantity. A particular system will consist of tens or thousands of FRUs.

The FRU concept has been carried to its logical conclusion by fault-tolerant computer vendors. They have the customer perform *cooperative maintenance* as follows. When a module fails the single-fault-tolerant system continues operating since it can tolerate any single fault. The system first identifies the fault within a FRU. It then calls the vendor's support center via switched telephone lines and announces that a new module (FRU) is needed. The vendor's support center sends the new part to the site via express mail (overnight). In the morning, the customer receives

a package containing replacement part and installation instructions. The customer replaces the part and returns the faulty module to the vendor by parcel post.

Cooperative maintenance has attractive economies. Conventional designs often require a 2% per month maintenance contract. Paying 2% of the system price each month for maintenance doubles the system price in four years. Maintenance is expensive because each customer visit costs the vendor about a thousand dollars. Cooperative service can cut maintenance costs in half.

Tolerating Design Faults

Tolerating design faults is critical to high availability. After the fault-masking techniques of the previous section are applied, the vast majority of the remaining *computer* faults are design faults. (Operations and environmental faults are discussed later).

One study indicates that failures due to design (software) faults outnumber hardware faults by ten to one. Applying the concepts of modularity, fail-fast, independent-failure modes, and repair to software and design is the key to tolerating these faults.

Hardware and software modularity is well understood. A hardware module is a field replaceable unit (FRU). A software module is a process with private state (no shared memory) and a message interface to other software modules[10].

The two approaches to fail-fast software are similar to the hardware approaches:

Self-checking: A program typically does simple *sanity checks* of its inputs, outputs, and data structures. This is called *defensive programming*. It parallels the double-entry book-keeping, and check-digit techniques used by manual accounting systems for centuries. In defensive programming, if some item does not satisfy the integrity assertion, the program raises an exception (fails fast) or attempts repair. In addition, independent processes, called *auditors* or *watch-dogs*, observe the state. If they discover an inconsistency they raise an exception and either fail-fast the state (erase it) or repair it[9,11].

Comparison: Several modules of different design run the same computation. A comparator examines their results and declares a fault if the outputs are not identical. This scheme depends on independent failure modes of the various modules.

The third major fault tolerance concept is independent failure modes. *Design diversity* is the best way to get designs with independent failure modes. Diverse designs are produced and implemented by at least three independent groups starting with the same specification. This software approach is called *N-Version programming*[12] because the program is written N-times.

Unfortunately, even independent groups can make the same mistake. There may be a mistake in the original specification, or all the groups may make the same implementation mistake. Anyone who has given a test realizes that many students can make the same mistake on a difficult exam question. Independent implementations of a specification by independent groups is currently the best way to approach design diversity.

N-version programming is expensive. N independent implementations raise the system implementation and maintenance cost by a factor of N or more. It may add unacceptable time delays to the project implementation. Some argue that this time and money is better spent on making one super-reliable design, rather than three marginal designs. There is yet no comparative data to resolve this issue.

The concept of *design repair* seems to further damage the case for design diversity. Recall the airplane rule ("Two engine airplanes have twice the engine problems of a one-engine plane.") Suppose each module of a triplexed design has a 100-year MTTF. *Without* repair, the triple will have its first fault in 33 years and its next fault in 50 years. The net is an 83 year MTTF. If only one module were operated, the MTTF would be 100 years. So the 3-version program module has worse MTTF than any simple program (but the 3-version program has lower failure variance). Repair is needed if N-Version programming is to improve system MTTF.

Repairing a design flaw takes weeks or months. This is especially true for hardware. Even software repair is slow when run through a careful program development process. Since the MTTF of a triplex is proportional to $\left(\frac{\text{MTTF}^3}{\text{MTTR}}\right)$, long repair times may be a problem for high-availability systems.

Even after the module is repaired, it is not clear how to re-integrate it into the working system without interrupting service. For example, suppose the failed module is a file server with its disc. If the module fails and is out of service for a few weeks while the bug is fixed, then when it returns it must reconstruct its current state. Since it has a completely different implementation from the other file servers, a special purpose utility is needed to copy the state of a "good" server to the "being repaired" sever while the "good" server is delivering service. Any changes to the files in the good server also must be reflected to the server being repaired. This repair utility should itself be an N-version program to avoid a single fault in a "good" server or in the copy operation creating a double fault. Software repair is not trivial.

Process Pairs and Transactions: a Way to Mask Design Faults

Process-pairs and *transactions* offer a completely different approach to software repair. They generalize the concept of checkpoint-restart to distributed systems. They depend on the idea that most errors caused by design faults in production hardware and software are transient. A *transient failure* will disappear if the operation is retried later in a slightly different context. Such transient software failures have been given the whimsical name *Hiesenbug* because they disappear when reexamined. By contrast, *Bohrbugs* are good solid bugs.

Common experience suggests that most software faults in production systems are transient. When a system fails, it is generally restarted and returns to service. After all, the system was working last week. It was working this morning. So why shouldn't it work now? This commonsense observation is not very satisfying.

To date, the most complete study of software faults was done by Ed Adams[13]. In that study, he looked at maintenance records of North American IBM systems over a four year period. The study found that most software faults in production systems are only reported once. He described such errors as *benign bugs.* By contrast, some software faults, called *virulent bugs*, were reported many times. Virulent bugs comprise significantly less than 1% of all reports. Based on this observation, Adams recommended that benign bugs not be repaired immediately. Harlan Mills, using Adam's data observed that most benign bugs have a MTTF in excess of 10,000 years. It is safer to ignore such bugs than to stop the system, install a bug fix, and then restart the system. The repair will require a brief outage, and a fault in the repair process may cause a second outage.

The Adams study and several others imply that the best short-term approach to masking software faults is to restart the system[3,13,14]. Suppose the restart were instantaneous. Then the fault would not cause any outage. Restating this, simplex system unavailability is approximately $\frac{\text{MTTR}}{\text{MTTF}}$. If MTTR is zero and MTTF>0 then there is no unavailability.

Process pairs are a way to get almost instant restart. Recall that a *process* is the unit of software modularity. It provides some service. It has a private address space and communicates with the other processes and devices via messages traveling on sessions. If the process fails, it is the unit of repair and replacement. A process pair gives almost instant replacement and repair for a process. A *process pair*[15] consists of two processes running the same program. During normal operation, the *primary process* performs all the operations for its clients, and the *backup process* passively watches the message flows (see Figure 4). The primary process occasionally sends checkpoint messages to the backup, much in the style of checkpoint-restart designs typical of the

1950's. When the primary detects an inconsistency in its state, it fails fast. The backup process is notified when the primary fails, and it *takes over* the computation. The backup is now the primary process. It answers all incoming requests and provides the service. If the primary failed due to a Hiesenbug, the backup will not fail and so there will be no interruption of service. Process pairs also tolerate hardware faults. If the hardware supporting the primary process fails, the backup running on other hardware will mask that failure.

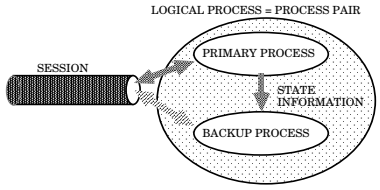

Figure 4: A process pair appears to other processes as a single logical process. Internally, it is really two processes executing the same program and with approximately the same state. The pair typically run on different computers and so have some failure-mode independence. In addition, process pairs mask Hiesenbugs.

One criticism of process pairs is that writing the checkpoint and takeover logic makes the arduous job of programming even more complex. It is analogous to writing the repair programs mentioned for N-version programming. Bitter experience shows that the code is difficult to write, difficult to test, and difficult to maintain.

Transactions are a way to automate the checkpoint/takeover logic. They allow "ordinary" programs to act as process pairs. Transactions are an automatic checkpoint-restart mechanism. The transaction mechanism allows an application designer to declare a collection of actions (messages, database updates, and state changes) to have the following properties:

Atomicity: either all the actions of the transaction will be done or they will all be undone. This is often called the all-or-nothing property. The two possible outcomes are called *commit* (all) and *abort* (nothing).

Consistency: The collection of actions is a correct transformation of state. It preserves the state invariants (assertions that constrain the values a correct state may assume).

Isolation: Each transaction will be isolated from the concurrent execution of other concurrent transactions. Even if other transactions concurrently read and write the inputs and outputs of this transaction, it will appear that the transactions ran sequentially according to some global clock.

Durability: If a transaction commits, the effects of its operations will survive any subsequent system failures. In particular, any committed output messages will eventually be delivered, and any committed database changes will be reflected in the database state.

The above four properties, termed ACID, were first developed by the database community. Transactions provide simple abstraction for Cobol programmers to deal with errors and faults in conventional database systems and applications. The concept gained many converts when distributed databases became common. Distributed state is so complex that traditional checkpoint/restart schemes require super-human talents.

The transaction mechanism is easily understood. The programmer declares a transaction by issuing a BEGIN_TRANSACTION() verb. He ends the transaction by issuing a COMMIT_TRANSACTION() or ABORT_TRANSACTION() verb. Beyond that, the underlying transaction mechanism assures that all actions within the Begin-Commit and Begin-Abort brackets have the ACID properties.

The transaction concept applies to databases, to messages (*exactly once* message delivery), and to main memory (persistent programming languages). Transactions are combined with process pairs as follows. A process pair may declare its state to be *persistent,* meaning that when the primary process fails, the transaction mechanism aborts all transactions involved in the primary process and reconstructs the backup-process state as of the start of the active transactions. The transactions are then reprocessed by the backup process.

In this model, the underlying system implements process pairs, transactional storage (storage with the ACID properties), transactional messages (exactly-once message delivery), and process pairs (the basic takeover mechanism). This is not trivial, but there are at least two examples: Tandem's NonStop Systems, and IBM's Cross Recovery Feature (XRF)[16,17]. With these underlying facilities, application programmers can write conventional programs that execute as process pairs. The computations need only declare transaction boundaries. All the checkpoint-restart logic and transaction mechanism is done automatically.

To summarize, process pairs mask hardware faults and Hiesenbugs. Transactions make it easy to write process pairs.

The Real Problems: Operations, Maintenance, and Environment

The previous sections took the narrow *computer* view of fault-tolerance. Paradoxically, computers are rarely the cause of a computer failure. In one study, 98% of the unscheduled system outages came from "outside" sources[3]. High-availability systems must tolerate environmental faults (e.g., power failures, fire, flood, insurrection, virus, and sabotage), operations faults, and maintenance faults.

The declining price and increasing automation of computer systems offers a straightforward solution to some of these problems: *system pairs* (see Figure 5). System pairs carry the disc-pair and process-pair design one step further. Two nearly identical systems are placed at least 1000 kilometers apart. They are on different communication grids, on different power grids, on different earthquake faults, on different weather systems, have different maintenance personnel, and have different operators. Clients are in session with both systems, but each client preferentially sends his work to one system or another. Each system carries half the load during normal operation. When one system fails, the other system takes over for it. The transaction mechanism steps in to clean up the state at takeover.

Ideally, the systems would not have identical designs. Such design diversity would offer additional protection against design errors. However, the economics of designing, installing, operating, and maintaining two completely different systems may be prohibitive. Even if the systems are identical, they are likely to mask most hardware, software, environmental, maintenance, and operations faults.

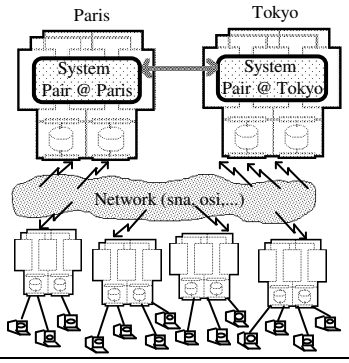

Figure 5: A system pair design. A system is replicated at two sites (Paris and Tokyo). During normal operation, each carries half the load. When one fails, the other serves all the clients. System pairs mask most hardware, software, operations, maintenance and environmental faults. They also allow online software and hardware changes.

Clearly, system pairs will mask many hardware faults. A hardware fault in one system will not cause a fault in the other. System pairs will mask maintenance faults since a maintenance person can only touch and break computers at one site at a time. System pairs ease maintenance. Either system can be repaired, moved, replaced, and changed without interrupting service. It should be possible to install new software or to reorganize the database at one system while the other provides service. After the upgrade, the *new* system *catches up* with the old system and replaces it while the old system is upgraded.

Special purpose system pairs have been operating for decades. IBM's AAS system, the Visa system, and many banking systems operate in this way. They offer excellent availability, and offer protection from environmental and operations disasters. Each of these systems has an ad

hoc design. No system provides a general purpose version of system pairs and the corresponding operations utilities today. This is an area of active development. General purpose support for system pairs will emerge during the 1990 decade.

Assessment of Future Trends

Over the last four decades, computer reliability and availability have improved by four orders of magnitude. Techniques to mask device failures are well understood. Device reliability and design has improved so that now maintenance is rare. When needed, maintenance consists of replacing a module. Computer operations are increasingly being automated by software. System pairs mask most environmental faults, and also mask some operations, maintenance, and design faults.

Figure 6 depicts the evolution of fault-tolerant architectures and the fault classes they tolerate. The density of the shading indicates the degree to which faults in that class are tolerated. During the 1960's fault, tolerant computers were mainly employed in telephone switching and aerospace applications. Due to the relative unreliability of hardware, replication was used to tolerate hardware failures. The decade of the 1970's saw the emergence of commercial fault-tolerant systems using process pairs coupled with replication to tolerate hardware as well as some design faults. The replication of a system at two or more sites (system pairs) extended fault coverage to include operations and environmental faults. The challenge of the 1990's is to build upon our experience and devise architectures capable of covering all these fault classes.

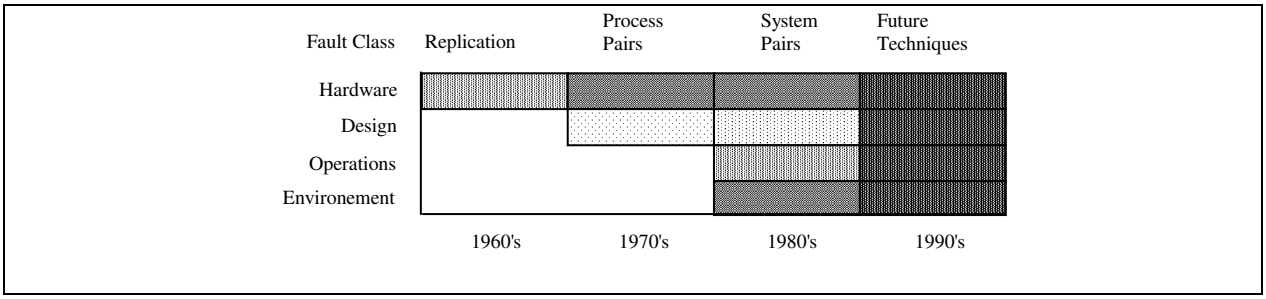

Figure 6. Summary of the evolution of fault tolerant architectures and their fault class coverage.

These advances have come at the cost of increased software complexity. System pairs are more complex than simplex systems. Software to automate operations and to allow fully online maintenance and change is subtle. It seems that a minimal system to provide these features will involve millions of lines of software. Yet it is known that software systems of that size have thousands of software faults (bugs). No economic technique to eliminate these faults is known.

Techniques to tolerate software faults are known, but they take a statistical approach. In fact, the statistics are not very promising. The best systems seem to offer a MTTF measured in tens of

years. This is unacceptable for applications that are life-critical or that control multi-billion-dollar enterprises. Yet, there is no alternative today.

Building ultra-available systems stands as a major challenge for the computer industry in the coming decades.

References

1. Avizienis, A., Kopetz, H. and Laprie, J. C., *Dependable Computing and Fault-Tolerant Systems.* Springer Verlag, Wien, 1987.
2. Watanabe, E. (translator). *Survey on Computer Security,* Japan Information Development Corporation, Tokyo, March 1986.
3. Gray, J., *Why Do Computers Stop and What Can We Do About It,* 6th International Conference on Reliability and Distributed Databases, IEEE Press, 1987, and *A Census of Tandem System Availability, 1985-1990*. IEEE Trans. on Reliability. Vol 39, No.4, 1990, pp. 409-418.
4. Tullis, N.,*Powering Computer-Controlled Systems: AC or DC?*. Telesis. Vol. 11 No. 1, Jan 1984, pp. 8-14.
5. Laprie, J. C., *Dependable Computing and Fault Tolerance: Concepts and Terminology*, Proc. 15th FTCS, IEEE Press, pp. 2-11, 1985.
6. Siewiorek, D.P, Swarz, R.W., *Reliable Computer Systems: Design and Evaluation*, Digital Press, Bedford, 1992.
7. Johnson, B.W., *Design and Analysis of Fault Tolerant Digital Systems*, Addison Wesley, Reading, 1989.
8. Pradhan, D.K., *Fault Tolerant Computing: Theory and Techniques, Vol I and II*, Prentice Hall, Englewood Cliffs, 1986.
9. Anderson, T., ed., *Resilient Computing Systems, Vol I,* John Wiley and Sons, New York, 1985.
10. Tanenbaum, A. S. (1987). *Operating Systems: Design and Implementation.* Prentice Hall, Englewood Cliffs,1989.
11. Randell, B., Lee, P. A. and Treleaven, P. C., *Reliability Issues in Computer System Design*. ACM Computing Surveys. Vol. 28, No. 2, 1978, pp.123-165.
12. Avizienis, A., *Software Fault Tolerance*, Proc. 1989 IFIP World Computer Conference, IFIP Press., 1989.
13. Adams, E., Optimizing Preventative Service of Software Products, IBM J R&D. Vol. 28, No. 1, Jan. 1984.
14. Mourad, J., *The Reliability of the IBM/XA Operating System*, Proc. 15th FTCS, IEEE Press, 1985.
15. Bartlett, J. (1981). *A NonStop™ Kernel*, 8th SIGOPS, ACM, 1981.
16. *IMS/XRF: Planning Guide,* IBM, GC24-3151, IBM, White Plains, NY. 1987.
17. Lyon, J., *Tandem's Remote Data Facility*, Compcon 90, IEEE Press, 1990.