High-Availability Computer Systems

Jim Gray, Digital Equipment Corp.
Daniel P. Siewiorek, Carnegie Mellon University

Paradoxically, the larger a system is, the more critical — but less likely — it is to be highly available. We can build small ultra-available modules, but building large systems involving thousands of modules and millions of lines of code is a poorly understood art, even though such large systems are a core technology of modern society.

Three decades ago, hardware components were the major source of faults and outages. Today, hardware faults are a relatively minor cause of system outages when compared with operations, environment, and software faults. Techniques and designs that tolerate these broader classes of faults are still in their infancy.

This article sketches the techniques used to build highly available computer systems.

Today's highly available systems deliver four years of uninterrupted service. The challenge is to build systems with 100-year mean time to failure and one-minute repair times.

Historical perspective

Computers built in the late 1950s offered a 12-hour mean time to failure. A maintenance staff of a dozen full-time computer engineers could repair the machine in about eight hours. This failure-repair cycle provided 60 percent availability. The vacuum tube and relay components of these computers were the major sources of failures; they had lifetimes of a few months. So 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 and repair, the program read the most recent checkpoint and continued the computation from that point. This checkpoint-restart technique let computers that failed every few hours perform long-running computations.

Device improvements have increased computer system availability. By 1980, typical well-run computer systems offered 99 percent availability.[2] This sounds good, but 99 percent availability is 100 minutes of downtime per week. Such

0018-9162/91/0900-0039$01.00 © 1991 IEEE

Table 1. Availability of system classes.

System Type	Unavailability (minutes/year)	Availability (in percent)	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

outages may be acceptable for commercial back-office computer systems that process work in asynchronous batches for later reporting. But mission-critical and on-line applications cannot tolerate 100 minutes of downtime per week. They require high-availability systems that deliver 99.999 percent availability: at most five minutes of service interruption per year.

The principal consumers of the new class of high-availability systems want them for process-control, production-control, and transaction-processing applications. Telephone networks, airports, hospitals, factories, and stock exchanges cannot afford to stop because of a computer outage. In these applications, outages translate directly into 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 10 hours to recover. These unmanaged computers give about 90 percent availability. Managed conventional systems fail several times a year. Each failure takes about two hours to repair. This translates to 99 percent availability.[2] Current fault-tolerant systems fail once every few years and are repaired within a few hours[3] — this is 99.99 percent availability. High-availability systems must have fewer failures and be designed for faster repair. Their requirements are one to three orders of magnitude more demanding than current fault-tolerant technologies.

Table 1 gives the availability of typical system classes. Today's best systems are in the high-availability range. As of 1990, the best general-purpose systems have been in the fault-tolerant range.

As the nines pile up in the availability measure, it is easier to think of availability in terms of denial of service measured in minutes per year. For example, 99.999 percent availability is about five minutes of service denial per year. Even this metric is a little cumbersome, so we use the concept of *availability class* or simply *class* by analogy to the hardness of diamonds or the class of a clean room. Availability class is determined by the number of nines in a system's or module's availability figure. More formally, if the system availability is A, the system's availability class is $\lfloor \log_{10}(1/(1-A)) \rfloor$. 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 40 years. Unfortunately, over the last two years there have been several major outages of the United States telephone system: a nationwide outage lasting eight hours, and a Midwest outage lasting four days. These outages show how difficult it is to build systems with high availability.

High availability requires systems designed to tolerate faults — to detect a fault, report it, mask it, and then continue service while the faulty component is repaired off line. Beyond the usual hardware and software faults, a high-availability system must tolerate other 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 repairs typically require interrupting service while installing new software. This happens at least once a year and typically takes an hour.
- Production computer software typically has more than one defect per thousand lines of code. When a system needs millions of lines of code, it is likely to have thousands of software defects. This seems to put a ceiling on the size of high-availability systems. The system must either be small or be limited to a failure rate of one fault per decade. For example, the 10-million-line Tandem system software has been measured to have a 30-year mean time to failure.[3]
- Database reorganization is required to add new types of information, 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 on-line reorganization utilities.
- Operators sometimes make mistakes that lead to system outages. A conservative estimate is that a system experiences one such fault a decade. Such faults cause an outage of a few hours.

Faults in these four classes cause more than 1,000 minutes of outage per year in a typical system. This explains why managed systems do worse than this but well-managed systems do slightly better (see Table 1).

High-availability systems must mask most of these faults. One thousand minutes per year is much more than the five-minute-per-year budget allowed for high-availability systems. But even fault-tolerant and high-availability systems cannot tolerate all faults. 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 percent availability — about one minute of outage per week.

This article surveys the fault-tolerant techniques used to achieve highly available systems and 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 the International Federation for Information Processing Working Group 10.4 and by the IEEE Computer Society Technical Committee on Fault-Tolerant Computing. Here we present key definitions from their results.[5]

We can view a system as a single module, but most systems are composed of multiple modules. These modules have internal structures, which are in turn composed of submodules. We discuss 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* is deviation of the actual behavior from the specified behavior. The failure occurs 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).

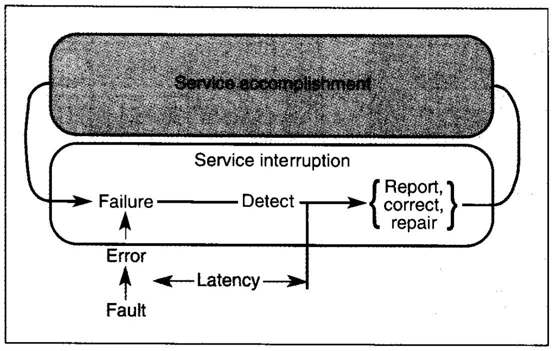


Figure 1. Usually a module's actual behavior matches its specified behavior, and it is in the *service accomplishment state*. Occasionally, a fault causes an error that becomes effective and causes the module to fail (observed behavior does not equal specified behavior). Then the module enters the *service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

For example, a programmer's mistake is a fault that creates a *latent error* in the software. When the system executes the erroneous instructions 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 system reads the memory, 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 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 for all modules. Module reliability is statistically quantified as *MTTF* (mean time to failure). Service interruption is statistically quantified as *MTTR* (mean time to repair). *Module availability* measures the ratio of service accomplishment to elapsed time. The availability of nonredundant systems with repair is statistically quantified as MTTF/(MTTF+MTTR).

Module reliability can be improved both by *valid construction* to reduce failures and by *error correction*.

Valid construction. Validation can remove faults during the construction process, thus assuring that the constructed module conforms to the specified module. Because physical components fail during operation, validation alone cannot assure high availability.

Error correction. Designs with redundancy reduce failures by tolerating faults. *Latent error processing* describes the practice of trying to detect and repair latent errors before they become effective. Preventive maintenance is an example. *Effective error processing* describes correction of 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 used for electronic, magnetic, and optical storage are examples of masking. An error-recovery mechanism typically denies the request and sets the module to an error-free state so that it can service subsequent requests.

Error recovery can take two forms. *Backward error recovery* returns to a previous correct state. Checkpoint-restart is an example. *Forward error recovery* constructs a new correct state. Redundancy in time — for example, resending a damaged message or rereading a disk block — are examples of forward error recovery.

In addition to these key definitions from the IFIP working group,[5] the following terminology to categorize faults is useful:

- *Hardware faults.* Failing devices.
- *Design faults.* Faults in software (mostly) and hardware design.
- *Operations faults.* Mistakes made by operations and maintenance personnel.
- *Environmental faults.* Fire, flood, earthquake, power failure, and 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 causes of physical device aging. Maintenance and redesign cause software aging.

Vendors usually quote the failure rates at the bottom of the bathtub (after infant mortality and before maturity). Transient failures often obey a Weibull distribution, a negative hyperexponential 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 use.

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

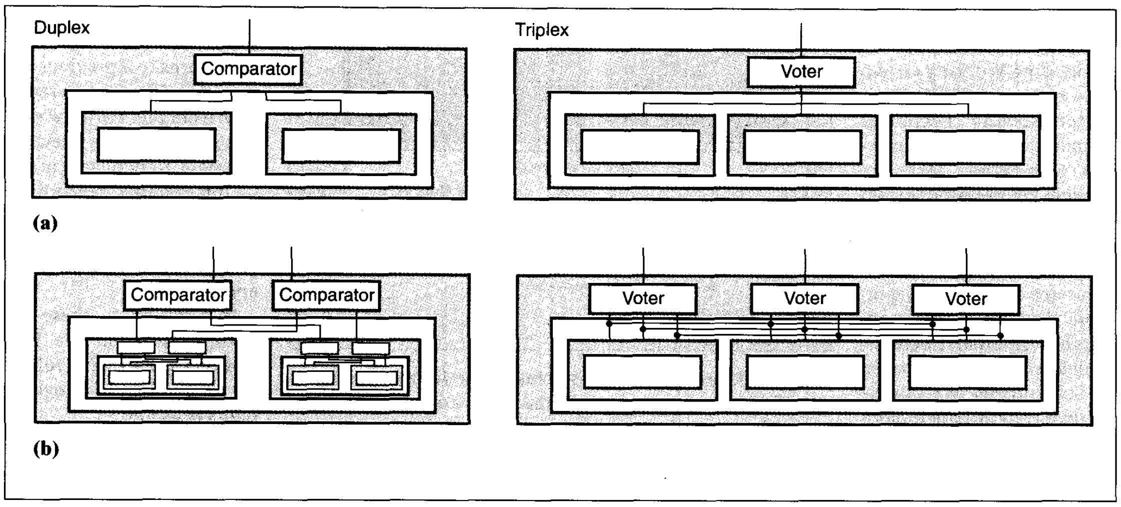


Figure 2. Fail-fast and fault-tolerant modules: (a) basic designs, (b) recursive designs stemming from the basic designs.

success rates are typically 99.9 percent, but 95 percent repair success rates are common. This is still excellent compared with the 66 percent repair success rates reported for automobiles.

Improved devices

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 increased the reliability of digital electronics dramatically:

- *Long-lived 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.
- *Fewer connectors.* Connections were a major source of faults because of 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.

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

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

Fault-tolerant design concepts

These more reliable devices are combined to create complex systems. Certain design concepts are fundamental to making the systems fault tolerant and, consequently, highly available.

- *Modularity.* Designers should hierarchically decompose the system into modules. For example, a computer may have a storage module, which in turn has several memory modules. Each module is a unit of service, fault containment, and repair. If a module fails, it is replaced by a new module.
- *Fail-fast.* Each module should either operate correctly or stop immediately.
- *Independent failure modes.* Modules and interconnections should be designed so that if one module fails, the fault does not affect other modules.
- *Redundancy and repair.* Spare modules should be installed or configured in advance, so when one module fails, the second can replace it almost instantly. The failed module can be repaired off line while the system continues to deliver service.

These principles apply to hardware faults, design faults, and software faults (which are design faults). However, their application varies, so we discuss hardware first, and then design and software faults.

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 such as a processor, a communications line, or a storage device. Designers use one of two techniques[6-8] to make a module fail-fast:

- *Self-checking.* A module performs the operation and also some additional work to validate the state. Error-detecting codes on storage and messages are examples of this approach.
- *Comparison.* Two or more modules perform the operation, and a comparator 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. However, it will likely continue to dominate 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 percent of processor circuits and 30 percent of the processor design time are devoted to self-checking. Comparison schemes, on the other hand, augment general-purpose circuits with simple comparator designs and circuits. The result is a reduction in overall design cost and circuit cost.

Figure 2a shows the basic comparison approach. 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 a design uses more than two modules, the module can tolerate at least one fault because the comparator passes through the majority output (two out of three in the Figure 2a triplex). The triplex design is called triple-module-redundancy. Figure 2b shows how the duplex and triplex designs can be made recursive. The modules are *N*-plexed and so are the comparators themselves, so comparator failures are also detected.

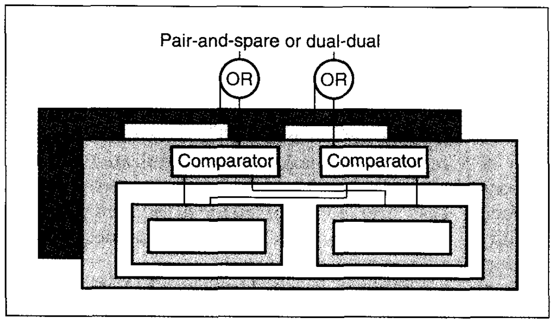


Figure 3. Using redundancy to mask failures. Triple-module-redundancy needs no extra effort to mask a single fault. Duplexed modules can tolerate faults by using a pair-and-spare or dual-dual design. If any single module fails, the super module continues operating.

Self-checking and comparison provide quick fault detection. Once a system detects a fault, it should report and mask the fault, as shown in Figure 1.

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

The pair-and-spare scheme requires more hardware than triplexing (four rather than three modules), but it 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.

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 comparator inputs do not agree, and the triplex module fails if two of the module inputs do not agree. If there is no repair, the supermodules in Figure 2 will have a MTTF of less than a year, as shown in Table 2.[6] This is

Table 2. MTTF estimates for various architectures using one-year MTTF modules with a four-hour MTTR (ε represents a small additional cost for the comparators).[6]

Architecture	MTTF	Class	Equation	Cost
Simplex	1 year	3	MTTF	1
Duplex	~0.5 year	3	≈MTTF/2	$2 + \varepsilon$
Triplex	0.8 year	3	≈MTTF(5/6)	$3 + \varepsilon$
Pair and spare	~0.7 year	3	≈MTTF(3/4)	$4 + \varepsilon$
Duplex plus repair	$>10^3$ years	6	$\approx \mathrm{MTTF}^2/2\mathrm{MTTR}$	$2 + \varepsilon$
Triplex plus repair	$>10^6$ years	6	$\approx \mathrm{MTTF}^3/3\mathrm{MTTR}$	$3 + \varepsilon$

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 lessens reliability in the cases of duplexing and triplexing. *Redundancy designs require repair to dramatically improve availability.*

Importance of repair

If failed modules are repaired or replaced within four hours of their failure, then the MTTF of the example systems goes from one year to well beyond 1,000 years. Their availability goes from 99.9 percent to 99.9999 percent and their availability class from 3 to 6 — a significant improvement. If the system uses thousands of modules, designers can repeat the construction recursively to *N*-plex the entire system and get a class 8 supermodule (1,000-year MTTF).

On-line module repair requires repair and reinstallation of modules while the system is operating. It also requires reintegrating the module into the system without interrupting service. Doing this is not easy; for example, it is not trivial to make the contents of a repaired disk identical to that of a neighboring disk. Similarly, when a processor is repaired, it is not easy to set the processor state to that of the other processors in the module. Reintegration algorithms exist, but they are subtle and each uses a different trick. There is no overall design methodology for such algorithms as yet. Today's on-line integration techniques are trade secrets protected by patents, because they are a key to high-availability computing.

Improved device maintenance

The declining cost and improved reliability of devices permit a new approach to computer maintenance. Computers can be composed of modules called *field replaceable units*. Each FRU has built-in self-tests exploiting one of the checking techniques mentioned above. These tests let a module diagnose itself and report failures electronically to the system maintenance processor and visually as a light on the module itself. A green light means no trouble, yellow means a fault has been reported and masked, and red indicates a failed unit. This system makes repairs easy. 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 of more than 10 years and to cost less than a few thousand dollars, so they can be manufactured and stocked in quantity. A system can consist of tens or thousands of FRUs.

Vendors of fault-tolerant computers have carried the FRU concept to its logical conclusion with a system called *cooperative maintenance*. When a module fails, the fault-tolerant system continues operating because 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 is needed. The vendor's support center sends the new part to the site via overnight courier. In the morning the customer receives a package containing a 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-percent-per-month maintenance contract. Paying 2 percent of the system price each month for maintenance doubles the system price in four years. Maintenance is expensive because each visit to a customer's site costs the vendor about a thousand dollars. Cooperative service can cut maintenance costs in half.

The simple and powerful ideas of fail-fast modules and repair via retry or spare modules seem to solve the hardware fault-tolerance problem. They can mask almost all physical device failures, but they do not mask failures caused by hardware design faults. If all the modules have faulty designs, then the comparators will not detect the fault. Similarly, comparison techniques do not apply to software — which is all design — unless design diversity is used.

Tolerating design faults

Tolerating design faults is critical to high availability. After fault-masking techniques are applied, the vast majority of the remaining computer faults are design faults. (We discuss operations and environmental faults later.)

A study indicates that failures due to design (software) faults outnumber hardware faults by 10 to one. Applying the concepts of modularity, fail-fast modules, independent failure modes, and repair to software and design is the key to tolerating these faults.

Software modularity is well understood. A software module is a *process* with a private state (no shared memory) and a message interface to other software modules.[9]

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. Programming for this is called *defensive programming*. It parallels the double-entry bookkeeping and check-digit techniques used for centuries in manual accounting systems. 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 watchdogs observe the state. If they discover an inconsistency, they raise an exception and either fail fast the state (erase it) or repair it.[10,11]
- *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. For diverse designs, at least three independent groups start with the same specification and produce code to implement it. 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, or a common mistake can arise from the original specification. Anyone who has given a test knows that many students make the same mistake on a difficult exam question. Nevertheless, independent implementations of a specification by independent groups is currently the best way to approach design diversity.

N-version programming is expensive, raising the system implementation and maintenance cost by a factor of *N* or more. Also, it may add unacceptable time delays to the project implementation. Some argue that the time and money are better spent on making one superreliable design rather than three marginal designs. At present, there is 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 one-engine planes. 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 three-version program module has a worse MTTF than any simple program (but the three-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. Because the MTTF of a triplex is proportional to ($MTTF^3$/MTTR), long repair times may be a problem for high-availability systems.

Even after the module is repaired, how to reintegrate it into the working system without interrupting service is not clear. For example, suppose the failed module is a file server with its disk. 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 the current state. Because 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" server while the good server is delivering service. Any files changed in the good server also must be changed in the server being repaired. The repair utility should itself be an *N*-version program to prevent a single fault in a good server or copy operation from creating a double fault. Software repair is not trivial.

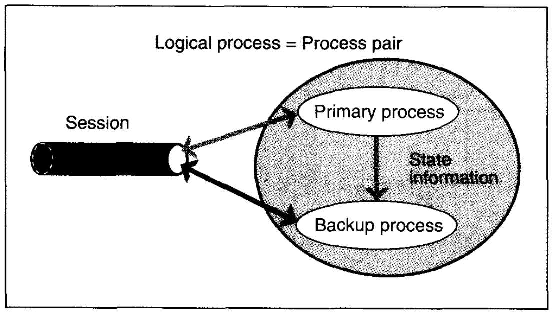


Figure 4. A process pair appears to other processes as a single logical process, but it is really two processes executing the same program and maintaining approximately the same state. The two processes typically run on different computers and have some failure-mode independence. In addition, process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

Process pairs and transactions

Process pairs and *transactions* offer a completely different approach to software repair. They generalize the concept of checkpoint-restart to distributed systems. 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 "Heisenbug" 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? But this commonsense observation is not very satisfying.

In the most complete study of software faults to date, Ed Adams looked at maintenance records of North American IBM systems over a four-year period.[13] He found that most software faults in production systems are reported only once. He described such errors as *benign bugs*. Some software faults were reported many times, but such *virulent bugs* made up significantly less than 1 percent of all reports. On the basis of this observation, Adams recommended that benign bugs not be repaired immediately. Using Adam's data, Harlan Mills 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: Simplex system unavailability is approximately MTTR/MTTF. If the MTTR is zero and the MTTF is greater than zero, then there is no unavailability.

Process pairs are a way to get almost instant restart. Recall that a process is a unit of software modularity that provides some service. It has a private address space and communicates with the other processes and devices via messages traveling on sessions. If a process fails, it is the unit of repair and replacement. A process pair — two processes running the same program[15] — gives almost instant replacement and repair for a process.

As Figure 4 shows, during normal operation the *primary process* performs all the operations for its clients, and the *backup process* passively watches the message flows. The primary process occasionally sends checkpoint messages to its backup, much in the style of checkpoint-restart designs of the 1950s. When the primary process detects an inconsistency in its state, it fails fast and notifies the backup process. Then the backup process becomes the primary process, answering all incoming requests and providing the service. If the primary process failed because of a Heisenbug, the backup will not fail and 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 the failure.

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

Transactions automate the checkpoint-takeover logic and allow "ordinary" programs to act as process pairs. Using the transaction mechanism, an application designer declares a collection of actions (messages, database updates, and state changes) to have the following properties:

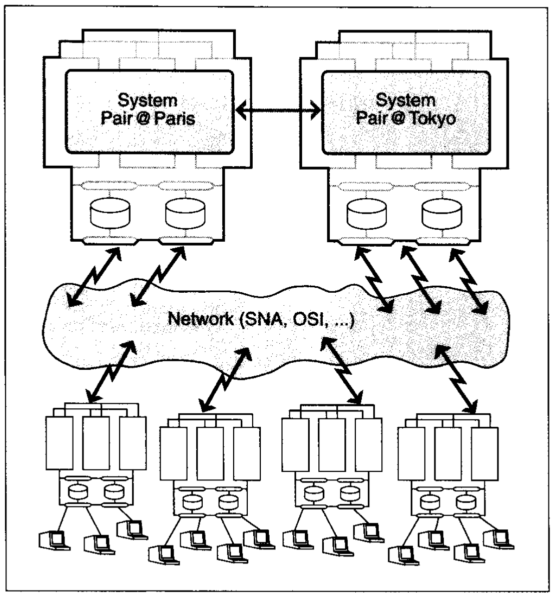


Figure 5. A system-pair design with system replication at two sites, Paris and Tokyo. During normal operation each system carries half the load. When one fails, the other serves all the clients. System pairs mask most hardware, software, operations, maintenance, and environmental faults. They also allow on-line software and hardware changes.

- *Atomicity*. Either all the actions making up 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 *commit* (all) and *abort* (nothing).
- *Consistency*. The collection of actions is a correct transformation of state. It preserves the state invariants (the assertions constraining the values that 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, the system will deliver any committed output messages, and the database state will reflect any committed database changes.

These four transaction properties, termed ACID, were first developed by the database community. The transaction mechanism provides a simple abstraction to help Cobol programmers deal with errors and faults in conventional database systems and applications. The concept gained many converts when distributed databases became common. A distributed state is so complex that traditional checkpoint-restart schemes require superhuman talents.

The transaction mechanism is easy to understand. The programmer declares a transaction by issuing a Begin_Transaction() verb and ends the transaction by issuing a Commit_Transaction() or Abort_Transaction() verb. Beyond that, the underlying transaction mechanism ensures that all actions within the Begin-Commit and Begin-Abort brackets have the ACID properties.

Designers can apply the transaction concept to databases, to messages ("exactly once" message delivery), and to main memory (persistent programming languages). To combine transactions with process pairs, a designer declares a process pair's 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 it stood at the start of the in-progress transactions. The backup process then reprocesses the transactions.

In this model, the underlying system implements 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). With the underlying facilities provided by such systems, application programmers can write conventional programs that execute as process pairs. The computations need only declare transaction boundaries. The checkpoint-restart logic and transaction mechanism are handled automatically.

Operations, maintenance, and environment

The previous sections took the narrow "computer" view of fault tolerance, but computers rarely cause computer failures. One study found that 98 percent of the unscheduled system outages came from outside sources.[3] High-availability systems must tolerate environmental faults (power failures, fire, flood, insurrection, virus, and sabotage), operations faults, and maintenance faults.

The declining price and increasing automation of computer systems offer a straightforward solution to some of these problems: *system pairs*. As Figure 5 shows, system pairs carry the disk-pair and process-pair design one step further. Two nearly identical systems are placed at least 1,000 kilometers apart. They are on different communication grids and power grids, at sites on different earthquake faults and in different weather systems. The maintenance personnel and operators are different. Clients are in session with both systems, but each client preferentially sends work to one system or another. Each system carries half the load during normal operation. When one system fails, the other system takes over. The transaction mechanism steps in to clean up the state at takeover.

Ideally, the systems would have different designs for additional protection against design errors. But 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.

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 because a maintenance person can only touch and break computers at one site at a time. System pairs will ease maintenance. Either system can be repaired, moved, replaced, or changed without interrupting service. It should be possible to install new software or to reorganize the database for one system while the other provides service. After the upgrade, the new system will catch up with the old system and then replace 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 protection from environmental and operations disasters. But each system has an ad hoc design: None provides a general-purpose version of system pairs. This is an area of active development. General-purpose support for system pairs will emerge during this decade.

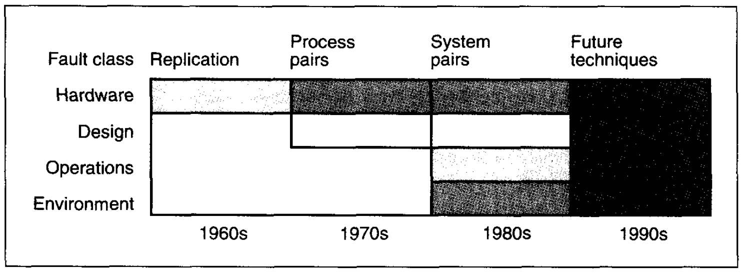


Figure 6. Evolution of fault-tolerant architectures and their fault class coverage.

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 have improved so maintenance is now rare. When needed, maintenance consists of replacing a module. Computer operations are increasingly automated by software. System pairs mask most environmental faults, and also mask some operations, maintenance, and design faults.

Figure 6 shows the evolution of fault-tolerant and high-availability architectures, and the fault classes they tolerate. The density of shading indicates the degree to which faults in a class are tolerated. During the 1960s, fault-tolerant techniques were mainly used in telephone switching and aerospace applications. Because of the relative unreliability of hardware, replication was used to tolerate hardware failures. The 1970s saw the emergence of commercial fault-tolerant systems using process pairs coupled with replication to tolerate hardware and 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 1990s is to build on our experience and devise architectures that can cover all fault classes.

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 on-line maintenance and change is subtle. A minimal system to provide these features will involve millions of lines of code. Software systems of that size have thousands of faults, and we know of no practical technique to eliminate such bugs.

Techniques to tolerate software faults are available, but they take a statistical approach. Their statistics are not very promising. The best systems offer a MTTF measured in tens of years. This is unacceptable for applications that involve lives or control multibillion-dollar enterprises. But today there is no alternative. Building ultra-available systems stands as a major challenge for the computer industry in the coming decades. ■

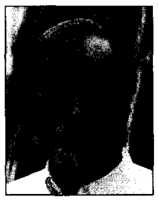
References

1. A. Avizienis, H. Kopetz, and J.C. Laprie, *Dependable Computing and Fault-Tolerant Systems*, Springer-Verlag, Vienna, 1987.
2. *Survey on Computer Security*, E. Watanabe, trans., Japan Information Development Corp., Tokyo, 1986.
3. J. Gray, "A Census of Tandem System Availability, 1985-1990," *IEEE Trans. Reliability*, Vol. 39, No. 4, Oct. 1990, pp. 409-418.
4. N. Tullis, "Powering Computer-Controlled Systems: AC or DC?" *Telesis*, Vol. 11, No. 1, Jan. 1984, pp. 8-14.
5. J.C. Laprie, "Dependable Computing and Fault Tolerance: Concepts and Terminology," *Proc. 15th FTCS*, Computer Society Press, Los Alamitos, Calif., 1985, pp. 2-11.
6. D.P. Siewiorek and R.W. Swarz, *Reliable Computer Systems: Design and Evaluation*, Digital Press, Bedford, Mass., 1992.
7. B.W. Johnson, *Design and Analysis of Fault Tolerant Digital Systems*, Addison-Wesley, Reading, Mass., 1989.
8. D.K. Pradhan, *Fault Tolerant Computing: Theory and Techniques*, Vols. 1, 2, Prentice-Hall, Englewood Cliffs, N.J., 1986.
9. A.S. Tanenbaum, *Operating Systems: Design and Implementation*, Prentice-Hall, Englewood Cliffs, N.J., 1989.
10. *Resilient Computing Systems,* Vol. 1, T. Anderson, ed., John Wiley, New York, 1985.
11. B. Randell, P.A. Lee, and P.C. Treleaven, "Reliability Issues in Computer System Design," *ACM Computing Surveys*, Vol. 28, No. 2, Apr. 1978, pp. 123-165.
12. A. Avizienis, "Software Fault Tolerance," *Proc. 1989 IFIP World Computer Conf.*, IFIP Press, Geneva, 1989, pp. 491-497.

13. E. Adams, "Optimizing Preventative Service of Software Products," *IBM J. R&D*, Vol. 28, No. 1, Jan. 1984, pp. 2-14.

14. J. Mourad, "The Reliability of the IBM/XA Operating System," *Proc. 15th FTCS*, Computer Society Press, Los Alamitos, Calif., 1985, pp. 76-83.

15. J. Bartlett, "A NonStop Kernel," Eighth Sigops, ACM, New York, 1981, pp. 22-29.



Jim Gray is with Digital Equipment Corporation's San Francisco Systems Center, where he is working on enhancements to Digital's database and transaction processing systems. Before joining Digital, he worked at Tandem on a system dictionary, parallel sort, and a distributed structured query language, NonStop SQL. Before that, he worked at IBM Research on projects including System R, SQL/DS, DB2, and IMS-Fast Path. Previously, he worked on Telsim at Bell Labs and managed the development of Cal TSS at UC Berkeley, where he wrote his doctoral dissertation on the theory of precedence parsing.

Daniel P. Siewiorek is a professor in the School of Computer Science and the Carnegie Institute of Technology in the Department of Electrical and Computer Engineering at Carnegie Mellon University. His research interests include computer architectures, reliability modeling, fault-tolerant computing, modular design, and design automation.

Siewiorek received a BS in electrical engineering from the University of Michigan in 1968 and an MS and a PhD in electrical engineering from Stanford University in 1969 and 1972. He is an IEEE fellow and a member of ACM, Tau Beta Pi, Eta Kappa Nu, and Sigma Xi.

Interested readers can write to Gray at Digital Equipment Corp., 455 Market St., 7th Floor, San Francisco, CA 94105; his Internet address is jimgray@sfbay.dec.com.