An Introduction to Fault-Tolerant Systems

Kjetil Nørvåg
Department of Computer and Information Science
Norwegian University of Science and Technology
7034 Trondheim, Norway

noervaag@idi.ntnu.no
http://www.idi.ntnu.no/~noervaag

IDI Technical Report 6/99, Revised July 2000
ISSN 0802-6394

Abstract

This report is an introduction to fault-tolerance concepts and systems, mainly from the hardware point of view. An introduction to the terminology is given, and different ways of achieving fault-tolerance with redundancy is studied. Knowledge of software fault-tolerance is important, so an introduction to software fault-tolerance is also given. Finally, some systems are studied as case examples, including Tandem, Stratus, MARS, and Sun Netra ft 1800.

Contents

1 Introduction

Whatever can go wrong will go wrong at the worst possible time and in the worst possible way...

–Murphy

Reliability and availability have become increasingly important in today's computer dependent world. In many applications where computers are used, outages or malfunction can be expensive, or even disastrous. Just imagine the computer system in a nuclear plant malfunctioning. Or the computer systems in a space shuttle booting just when the shuttle is about to land... These are the more exotic examples. More close to everyday life, are the telecommunications switching systems and the bank transaction systems. To achieve the needed reliability and availability, we need *fault-tolerant computers.* They have the ability to tolerate faults by detecting failures, and isolate defect modules so that the rest of the system can operate correctly. Reliability techniques have also become of increasing interest to general-purpose computer systems. Four trends contribute to this:

The first is that computers now have to operate in *harsher environments.* Earlier, computers operated in clean computer rooms, with stable climate and clean air. Now the computers have moved out to industrial environments, with temperatures over a wide range, dust, humidity and unstable power supply. All these factors alone could make a computer fail.

Second, *the users* have changed. Earlier, computer operators were trained personnel. Now, with an increasing number of users, the typical user knows less about proper operation of the system. The consequence is that computers have to be able to tolerate more. Haven't we all seen users swearing over a disappeared document in a text editor (Backup? What is that?), or heard about people that accidently have poured coffee into the computer?

Third, the *service costs* increases relative to hardware costs. Earlier the average machine was a very expensive, big monster. At that time, it was common with one or several dedicated operators to keep the system up and running. Today, a computer is cheap, and the user has the job of being the "operator". The user can not afford frequent calls for field service.

The fourth and last trend is *larger systems.* As systems become larger, there are more components that can fail. This means, to keep the reliability at an acceptable level, designs have to tolerate faults resulting from component failures.

So, what can cause outages of equipment, making fault-tolerance techniques necessary? We can split them into outages caused by:

- *Environment:* This is facilities failures, e.g. dust, fire in the machine room, problems with the cooling, earthquakes or sabotage.
- *Operations:* Procedures and activities of normal system administration, system configuration and system operation. This can be installation of a new operating system (requires booting of the machine), or installation of new application programs (which requires exit and restart of programs in use).
- *Maintenance:* This does not include software maintenance, but could be hardware upgrading.
- *Hardware:* Hardware device faults.
- *Software:* Faults in the software.
- *Process:* Outages due to something else, e.g. a strike.

Interesting to note is that, contrary to common assumptions, few outages are caused by hardware faults. In a modern system, fault-tolerance masks most hardware faults, and the percentage of outages caused by hardware faults are decreasing. On the other side, outages caused by software faults are increasing. According to a study on Tandem systems [4], the percentage of outages caused by hardware faults was 30% in 1985, but had decreased to 10% in 1989. Outages caused by software faults increased in the same period, from 43% to over 60%!

```
FUNCTION div(a, b: REAL):REAL
BEGIN
    div := a/b;
END;
```

Figure 1: Example function with a fault.

Organization of This Report

In the next section we will give an introduction to terminology. In Section 3, some basic fault-tolerance techniques are presented. In Section 4 and 5, we show how these techniques are used in existing systems, both general-purpose and special high-availability systems. Finally, in Section 6, we summarize the material presented in this report.

2 Terminology

We have already referred to *fault* and *fault-tolerant* without defining them. Here we will try to give a more precise definition of these, and some other used terms in the area of fault-tolerance. The definitions here are based on those given by Laprie [12], Avižienis and Laprie [1], Gray [5], and Gray and Reuter [4].

When a system or module is designed, its behavior is *specified*. When in service, we can *observe* its behavior. When the observed behavior differs from the specified behavior, we call it a *failure*[1]. A failure occurs because of an error, caused by a fault. The time between the occurrence of an error and the resulting failure is the error latency. An example is the function in Figure 1. This function has a fault, it does not check the value of the variable `b`. This fault results in a latent error in the function `div`. If the function is executed with a zero-value as `b`-argument, that is an error. When the division is executed, we have a program failure. Another example is a cosmic ray that discharges a memory cell (fault), causing an error. When the memory cell is read, we have a memory failure and the error becomes effective. This could be illustrated as:

$$\text{Fault} \rightarrow \text{Error} \rightarrow \text{Failure} \rightarrow (\text{Error latency}) \rightarrow \text{Detect}$$

Faults can be *hard* or *soft* (transient). A module with a hard fault will not function correctly, it will continue with a high probability of failing – until it is repaired. A module with a soft fault appears to be repaired after the failure. A hard fault could be a device with a burnt-up component. This will certainly not fix itself. A soft fault could be electrical noise interfering with the computer. If this has made a data transport on a bus fail, a second attempt could work if there is no noise at that time.

Module reliability measures the time from an initial instant to the next failure event. This reliability is statistically quantified as *mean-time-to-failure (MTTF)*. The average time it takes to repair a module after the detection of the failure is called *mean-time-to-repair (MTTR)*. As a result, we get the *module availability*, which is the ratio of service accomplishment to elapsed time:

$$\frac{MTTF}{MTTF + MTTR}$$

A failure is not necessarily caused by "something wrong happening". It can also be caused by a delay of correct behavior, a *timing failure*. *System availability* is the fraction of the offered load that is processed with acceptable response times. We classify systems into different availability classes as shown in table 1. Currently, most general-purpose systems are operating in class 3 or 4. The best fault-tolerant systems are operating in class 5 [5].

[1] It should be noted that in some literature, failure and error are used interchangeably, and in some literature failure and fault are used interchangeably.

System Type	Unavailability (min/year)	Availability	Class
Unmanaged	52560	90.%	1
Managed	5256	99.%	2
Well-managed	526	99.9%	3
Fault-tolerant	53	99.99%	4
High-availability	5	99.999%	5
Very-high-availability	0.5	99.9999%	6
Ultra-availability	0.05	99.99999%	7

Table 1: Availability of typical system classes [4].

To achieve a reliable, high-available system, two very different approaches can be used: *fault-avoidance* and *fault-tolerance*. While fault-avoidance is prevention of fault-occurrences by construction, fault-tolerance is the use of redundancy to avoid failures due to faults. Fault-avoidance is difficult, and close to impossible in large and complex systems. This makes fault-tolerance the only realistic alternative for the classes of systems we are studying here:

- *General-purpose computer systems:* General-purpose computers in the high-end of the commercial market, employing fault-tolerance techniques to improve general reliability.
- *High-available computer systems:* Systems designed for availability class 5 or higher, often some kind of database system or telephone switching system.
- *Long-life systems:* Systems designed for operating for a very long time without any chance of repair. Long-life systems are typical mobile systems where on-site repair is difficult, or maybe impossible. Examples are unmanned spacecraft systems like satellites or space exploration vehicles. These systems differ from other fault-tolerant systems discussed earlier by having redundancy not only in the electrical systems, but also in mechanical parts. They are also required to achieve correct operation over long periods of time.
- *Critical-computation systems:* Systems doing some critical work where faulty computations can jeopardize human life or have high economic impact. This could e.g. be the computers in a space shuttle, nuclear plant or air traffic control system, where malfunction can be extremely disastrous.

We will in this report concentrate on the first two classes, general-purpose computer systems and high-available computer systems. The basic techniques are the same for the two other classes, except that the requirements for those systems are much higher.

3 Fault-Tolerance Techniques

Module reliability can be improved by:

- Validation
- Error correction (or error processing, as it is called by Laprie [12])

Validation is used to reduce errors during the construction process. There are many ways to do this, one is to develop a model of the system in a formal language and use a validation program for the validation.

Error correction (a taxonomy of the approaches is shown in Figure 2) reduces failures by using redundancy to tolerate faults. *Latent error processing* tries to detect and repair latent errors before they become effective. An example is preventive maintenance. *Effective error processing* tries to

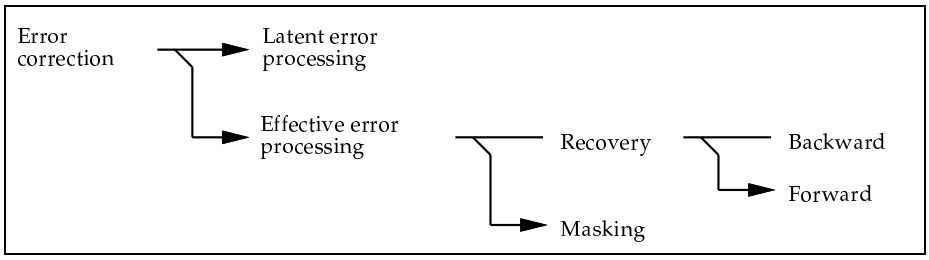
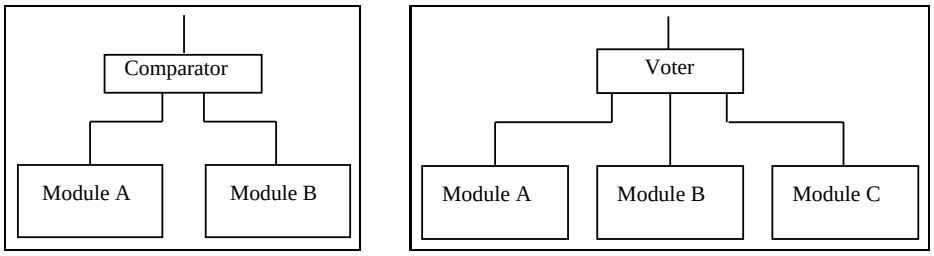


Figure 2: Taxonomy of error correction approaches.



(a) Duplex (b) Triplex

Figure 3: Basic failfast design.

correct the error after it becomes effective. This can be done by *masking* or *recovery*. An example of masking is error correcting codes. Recovery denies the requested service, and sets the module to an error-free state. We have two forms of recovery, *backward* and *forward* recovery. Backward recovery returns to a previous correct state. This can be checkpoint-restart, which means that the state is stored at regular intervals, and at restart time the last stored state is loaded and restarted from. With *forward recovery*, a new correct state is constructed, e.g. by re-sending a message or re-reading a disk page.

Four different types of redundancy are used in fault-tolerant systems, and in the following sections they will be studied:

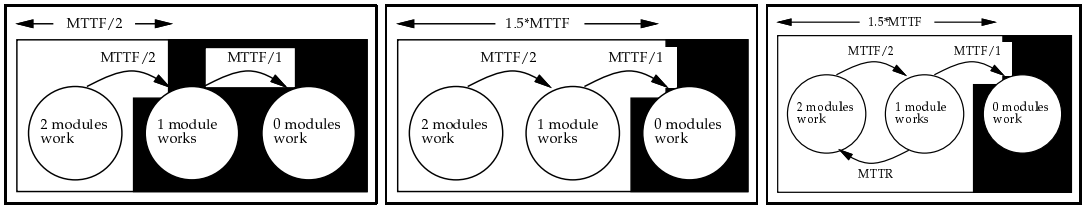
- Hardware redundancy
- Information redundancy
- Software redundancy
- Time redundancy

3.1 Hardware Redundancy

There are basically two approaches in hardware redundancy: addition of replicated modules, and use of extra circuits for fault detection.

3.1.1 Module Replication

To avoid wrong results and actions being made, it is desirable that that failing modules stops execution when a fault is detected, with a small fault latency (that means, as soon as possible after the failure). Modules having this property are called *failfast*. We also talk about *fail-silent*



(a) Duplex based on non-failfast modules

(b) Duplex based on failfast modules

(c) Duplex with repair

Figure 4: Transition diagrams to estimate the MTTF of a duplexed module [4].

modules, which are modules that only deliver correct results (in the value- and time domain). If it is not possible to deliver correct results (because of failure), it delivers no results at all [18].

Making a module failfast can be done by duplication. Two identical copies of a module are employed (see Figure 3a), with a comparator checking the output of the two copies. When the output differs, a fault is detected. The fault is detected immediately (the fault latency is small), and it is therefore failfast. This is a widely used technique, because it is easy to realize, and relatively cheap. The cost of duplication is twice that of an equivalent simplex system, and the duplication-checking is supported by several microprocessors. Two examples are the Motorola 88000 and the AMD AM29000 which have a master/slave ability determined by a "test" pin. The outputs of the slave copy are disabled, although it sees the same input stream as the master. It performs the same operations as the master, and compares with the master's outputs.

Another approach is to use three or more modules (N-plex), and apply voting rather than comparison. If we have three, we have enough redundant information to mask the failure of one of the modules. The masking is accomplished by means of a majority vote on the three outputs. This is called triple modular redundancy (TMR), shown in Figure 3b. We say that this module is *failvote*, since it requires majority. We could also let the voters first sense *which* modules are available and then use the majority of the available modules. This is called *failfast voting*, and can operate with less than a majority of the modules, which gives better reliability than just failvote. To increase the reliability further, we can make these designs recursive; that means failfast/failvote modules connected by a comparator or voter.

Given the MTTF of one component module, the MTTF for various architectures with such modules can be calculated. This is illustrated with transition diagrams in Figure 4. The shaded areas represent failed regions. As we can see, duplex does not increase the MTTF. However, the resulting module is failfast, which is important, as fault-tolerant designs are much easier to design and analyze with failfast modules. If the duplex module i based on failfast modules, the MTTF is increased to 1.5 of the MTTF of the failfast component modules.

To dramatically increase the MTTF for a module, repair of a module while the system is operating is necessary. If the failed module can be repaired sufficiently fast, the MTTF can be very high. One example is triplex failfast with repair. This approach uses three modules, whose outputs are continuously compared to achieve error detection. If a miscompare occurs, the system attempts to identify the failed module, using self-test diagnostics. The failed module is taken out of use, and can be repaired while the rest of the system is operating. It is interesting to note that a triplex of one-year MTTF modules can give a MTTF of $> 10^6$ years using this approach!

However: no chain is stronger than its weakest link. The same applies here as well. If the comparator at the top of the (possibly recursive) hierarchy fails, we have a problem if no other techniques are used as well. An example of a technique solving this problem is self-checking, which is explained in Section 3.2.

Interconnection Reconfiguration

To avoid connection failure (e.g. bus or switch failure) to be fatal, interconnection reconfiguration can be provided. This is easy to provide if the system already uses some kind of interconnection that have alternative paths between modules. An example is a computer connected in a hypercube network. Even if one connection fails, messages can be rerouted vie other connections.

Watchdog Timers and Timeouts

Watch-dog timers and timeouts are used for keeping track of progress. A watchdog timer should be reset within a certain amount of time, if not, a failure is assumed to have occurred. Timeouts are also based on the principle that an operations should not take more than a certain time to complete. If an operation has not completed within a certain maximum time, that indicates a possible failure.

3.2 Information Redundancy

Information redundancy is the addition of extra information to data, to allow error detection and correction. This is typically error-detecting codes, error-correcting codes (ECC), and self-checking circuits.

Error-Detection (and Correction) Codes

Parity codes are used in most modern computers for memory error detection. This is a simple code that does not require much additional hardware. Another, more advanced code is *m-of-n* code. This is a code that requires code words to be n bit of length, and contains exactly m ones. *Cyclic* and *checksum* codes are also common. To check arithmetics operations, *arithmetic codes* can be used. Arithmetic codes are preserved by arithmetic operations [21]. The data presented to the arithmetic operation is encoded before the operation is performed. When the operation is completed, the resulting code is checked to make sure it is valid.

Codes can also be *error-correcting*. Data encoded with error-correcting codes (ECC) can contain errors, but contains enough redundancy to recover the data.

Consistency Checking

This is a verification of the results being reasonable. Examples are range checks; e.g. address checking, and arithmetic operation checking.

Self-Checking Logic

As mentioned in Section 3.1.1, failure in a comparator element at the top of the hierarchy can be disastrous (*checking-the-checker problem*). This single point of failure can be eliminated through self-checking and fail-safe logic design [16].

A circuit is said to be self-checking if it has the ability to automatically detect the existence of a fault, without the need for any externally applied stimulus. When the circuit is fault free and presented a valid input code word, it should produce a correct output code word. If a fault exists, however, the circuit should produce an invalid output code so that the existence of the fault can be detected. Self-checking is needed to make fail-silent modules.

3.3 Software Redundancy

In this report we focus on hardware, but to be able to fully understand the use of the hardware techniques, it is important to know about the software techniques used on top of the hardware. In fact, software is the most challenging problem in the area of fault-tolerance. As mentioned earlier, today's hardware is relative reliable compared to the software.

There are some important differences between software and hardware errors. Physical errors (in hardware) will not recur after they have been discovered and corrected. Unfortunately, this is not the case with software errors. In the process of correcting a programming error, new errors are likely to be created. Software development is also a more complex and immature art than hardware design.

It is said that perfect software is possible — it's just a matter of time and money. This might be true, but for a large and complex software system, there is not enough of either time or money. Several examples proving this could be provided; one is the space shuttle software. Even with extensive testing, the software contained lots of serious errors, the astronauts were, in fact, provided with bug lists before start! One known bug is that it was not possible for them to use more than one keyboard at a time. If they did, the input to the computer would be a *logical or* of the signals from the two keyboards!

3.3.1 N-Version Programming and Software Fault-Tolerance

We have two major software fault-tolerance techniques [4]:

- *N-version programming:* Write the program N times, then operate all N programs in parallel, and take a majority vote for each answer. This is an analogy to the N-plexing of hardware modules.

- *Transactions:* Write the program as a transaction. Use a consistency check at the end, and if the conditions are not met, restart. It should work the second time...

The big disadvantage with N-version programming is its cost. It is expensive, and repair is not trivial. It is also difficult to maintain. To get a majority, we need to have at least 3 versions. Programmers tend to do the same problems, so there is a certain risk of getting the same mistake in the majority of the programs. It is often argued that you can get better reliability if you use one expensive (and supposedly good) programmer, than three average, cheaper programmers.

Gray [5] introduces the concept of Heisenbugs and Bohrbugs. Heisenbugs are transient software errors, while Bohrbugs are solid, deterministic bugs. Heisenbug proponents suggest crashing the system and restarting at the first sign of trouble; the failfast approach. This requires programs implemented as transactions. If not, the system might be brought back to an inconsistent state. If Heisenbugs are the dominant form of software faults: Failfast + transactions + system pairs results in software fault-tolerance. In general, transaction-like approaches are found to be superior to N-version programming [14].

3.3.2 Software Fault Detection

Many faults can be detected in much the same way as hardware faults are detected. These techniques can be implemented either in the operating system or as user processes:

- Watchdog timers and timeouts: A watchdog daemon process can watch the life of an application by periodically sending the process a signal and check the return value to detect if it is alive. Huang and Kintala shows how this can be implemented and integrated into a system in [7].

- Consistency checking/self-checking: The programs can use assertions to check the results of computations. If the assertion is false, a fault is detected. How to resolve this fault is up to the program.

- Time redundancy: Run the same program several times and compare the results. This is NOT as trivial as it might seem.

3.3.3 Software Fault Masking

As said earlier in this report, redundancy is one of the keys to high reliability. It is tempting to try to do the same with software. However, this rises some questions: How do you pair-and spare software modules, messages and remote procedure calls? Gray and Reuter [4] show that *process pairs* could be the solution.

Process pairs need to interact, and a reliable message systems is needed, because messages can be lost, duplicated, delayed, corrupted and permuted. By implementing sessions, timeouts and message sequence numbers, all message failures are converted to lost messages. By combining this simple failure model with message acknowledgment, sender timeout and message retransmission, the message systems becomes highly reliable.

Processes fail by occasionally being delayed for some repair period, having all their data reset to null state, and then having all their input and output messages discarded. With process-pairs, we have one working process, the primary. When the primary fails, the second (the backup) takes over for the primary and continues the computation. One problem is, how should the backup know when the primary has failed? One way to do this is having the primary send *I'm Alive* messages to the backup. When the backup knows it should have received several messages, but have received none, it assumes the primary has failed, and takes over.

Gray and Reuter [4] describes three kinds of takeover:

- Checkpoint-restart
- Checkpoint message
- Persistent

With checkpoint-restart, the primary records its state on a duplex storage module, and the at takeover, the backup starts by reading these pages. The primary could also send its state changes as messages to the backup, and at takeover the backup gets its current state from the most current message. This is called the checkpoint message technique. With third technique, persistent, all state changes are implemented as transactions. At takeover, the backup process restarts in the null state and lets the transaction mechanism clear up (undo) any recent uncommitted state changes.

These techniques have some disadvantages. With checkpoint-restart, we have a long repair time (reading the pages take time). This yields highly *reliable*, but not highly *available* process execution. With checkpoint message, we rely on all messages sent from the primary being received by the backup. This is dangerous! Persistent process pairs seems as the best technique: It is "easy" to implement and easy to understand.

There is also a second approach to implement resilient processes: replicated execution [17]. In this approach, several processes execute the same program concurrently. Here the reliability of the execution could be increased by taking a majority vote on the output from the processes.

3.4 Time Redundancy

Hardware- and information- redundancy requires extra hardware. This could be avoided by doing operations several times in the same module and check the results, in stead of doing it in parallel on several modules and compare the outputs. This reduces the amount of hardware at the expense of using additional time, and is especially suitable if faults are mostly transient. It could also be used to distinguish between permanent and transient faults.

4 Fault-Tolerance in General-Purpose Computers

People may not realize to which extent fault-tolerance techniques are used in general-purpose computers to increase their reliability. Techniques used in general-purpose computers are also utilized in more specialized fault-tolerant computers, so it is a good starting point to study these computers.

	Detection	Recovery
Memory	Parity and double-error-detecting code	Single error-correction code, retry and dynamically reconfigurable memory
I/O	Parity	Retry
Processor	Parity, duplication and comparison	Retry

Table 2: Error detection and recovery mechanisms.

Based on the assumption that most errors are transient, recovery consists primarily of retry by the error-detection mechanisms [16]. A retry is usually not done immediately, but after a pause. During that time, the source of the transient error, e.g. power instability, might have disappeared.

A computer is usually divided into three main sections: processor, primary memory and I/O. These sections often employ slightly different fault-tolerant techniques. Error detection and recovery mechanisms in a typical system is presented in table 2. On memory data, parity is used. In the more expensive computers, and now also increasingly on cheaper computers, double-error-detecting codes are also used. In addition, parity is used on address and control information. Recovery can be done with single error-correcting codes on data and retry on address and control information parity error. Memory, under software control, can in some systems (e.g. VAX 8600) be dynamically reconfigured to exclude bad pages.

Many of the techniques used on memory, can also be used on I/O. Retry is often extensively used here, especially on devices as disks this is an effective approach.

A processor contains many registers. To provide fault-tolerance here, the same techniques as those used on memory can be used. In addition, duplication of control logic is commonly used.

To increase availability, repair time has to be minimized. One way to do this, is *remote diagnostics*. When a fault is detected, either the computer or an operator notifies a service center, possibly located far away from the computer site. The service center can connect to the computer, and use diagnostic programs if necessary. The personnel at the service center can either fix the problem from their site (in case of software problems) or ship a replacement module (in case of hardware failure) to the failing site.

Software techniques are also heavily used to increase fault-tolerance. One widely used technique is transaction processing, e.g. in databases. Computer-, power- or disk-failure should not be enough to damage the database if the necessary precautions are taken.

5 Fault-Tolerance in Some High-Availability Systems

Most high-availability systems are based on dynamic redundancy. Multiple processors and extensive use of error detection and correction techniques are used to improve reliability. The fault-tolerant computers we are going to discuss have much in common:

- Modularity: Hardware and software are based on modules of fine granularity. This makes self-check, diagnostics and repair considerable easier.
- Replication of modules.
- On-line maintenance: It is important to make diagnostics and repair while the system is in service. The systems can also be made capable of contacting service centers after a failure of a hardware component is detected. The service centers can then find the right replacement part, and ship it to the customer together with installation instructions. In fact, the customer might not even have discovered the failure before the courier knocks on the door with the replacement part!
- Simplified user interfaces: Many failures can be traced back to mistakes done by the operator. Simplified user interfaces can reduce the probability of failure considerably.

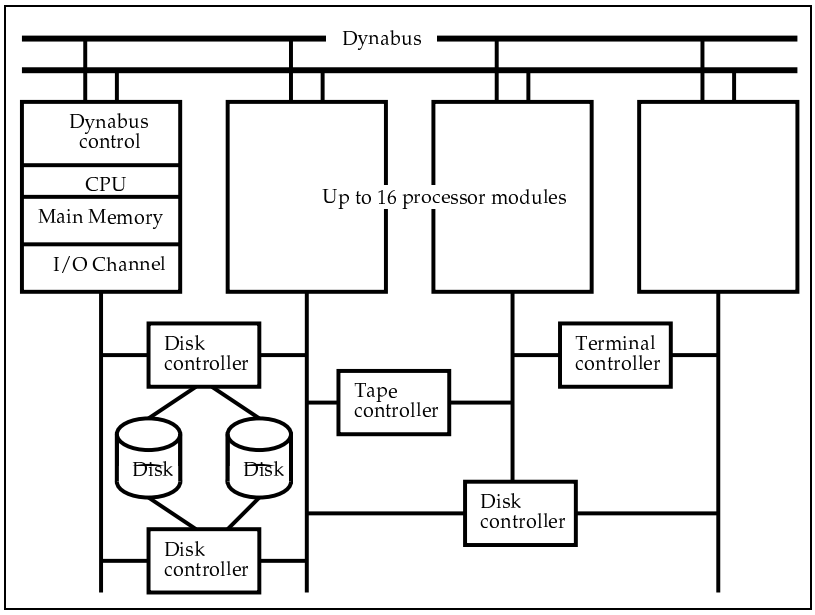


Figure 5: Original Tandem architecture.

- Several independent power supplies. Failure in a single power supply should not make the whole system fail. In addition, many fault-tolerant systems have a battery system, which can supply at least the main memory with enough power. The processor can be put to sleep, and when power is back, the system can be put back in service immediately. This avoids a possibly very time consuming restart.

Traditionally, many of the systems that need high availability have been some kind of on-line transaction systems, e.g. telecommunication databases, banking systems, or travel reservation systems. As a result, most fault-tolerant computers are designed and optimized for such applications.

5.1 Tandem Computers

The first commercially available system designed for high availability in general-purpose computing, was the Tandem 16 NonStop system. The system was designed for transaction processing, and the three main goals was to [9]:

1. Provide autonomous fault detection, system reconfiguration and repair without interrupting the functions of the fault-free components within the system.

2. Tolerate all single faults. When a hardware or software module fails, another module immediately takes over the failing module's function.

3. Using modularity to make the system easy expandable and improve the maintainability of the system.

Hardware

The Tandem concept was based on replication of processors, memories and disks both to tolerate failures and provide modular expansion of computing resources. All modules are designed to be fail-fast. The computer architecture is loosely coupled, consisting of two to 16 multiprocessor units connected via a dual-bus system (see Figure 5).

Checksums, parity error checking, error correcting memories and watchdog timers are used to detect faults on the bus and memories. Each processor in the Tandem system receives power from its own dedicated power supply.

Software

Each processor module runs independently of each other, and the system manage faults through the Guardian operating system. The system uses process pairs and checkpoint-messages. When a process is created, a primary process is created on one processor module, and a backup process is created on another module. As explained in 3.3.3, status information is periodically sent to the backup process. If the primary process fails, the backup process takes over.

The operating system periodically probes the component modules to check for errors. If errors are detected, failing modules are taken out of service. This means that recovery from hardware failures has to be implemented in the applications.

To reduce the probability of faults in the software, consistency checks and defensive programming techniques are used. All communication is done by messages.

Modern Tandem Systems

The first Tandem system was introduced in 1976. The processor module, NonStop I, included a 16-bit processor and up to 512 KB of main memory. Since then, processor architecture has evolved through several models. The last models, CLX800 and Cyclone are high-performance processors employing modern techniques as, e.g., deep pipelines. The buses connecting the processor modules have basically the same structure as in 1976, but they have since 1989 been realized with fiber-optic links.

5.2 Stratus

Another system competing in the same market as Tandem, is Stratus. Tandem is based on pairing, and needs software help to recover from hardware failures. Stratus is based on pair-and-spare, the spare component continues processing until the faulty component can be replaced. No data errors are visible to the software. This gives easy programming. Also, there should be no performance degradation because of faults. The problem with this approach is that it is very hardware intensive. E.g., four processors are used to perform the functions of one processor.

Hardware

Each processor board in the Stratus system consists of two processors. These processors are driven in lock-step by the same clock. Processes running on the system are mirrored, and the outputs are compared. If the outputs do not match, the board removes itself out of service. The operating system now runs diagnostic tests on the board. If the error was caused by a transient fault, the board is put back in service. If the fault was permanent, the board has to be replaced. This can be one while the system is in service, *hot swap*. Two processor boards are running the same functions, with the same data. When one board fails, the other can continue without interruption.

The memory in a Stratus system also work in lock-step pairs. Self-checking logic is used for control, and error-correcting codes used for data.

The processor modules are connected via a duplicated module bus called StrataBus, shown in Figure 6. This bus uses parity on groups of signals.

Software

Stratus supports its own operative system VOS as well as an extended version of UNIX. Both systems provides support for multiple processors and transactions (also distributed transactions).

As said previously, the Stratus architecture isolates users from almost all hardware failures. Still, there might be bugs in the system (read this as "there is certainly bugs in the system"!).

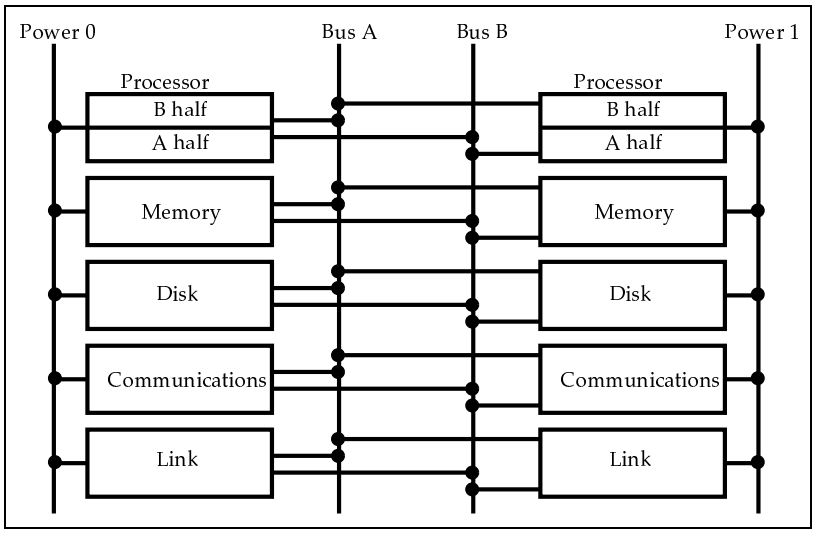


Figure 6: Stratus architecture.

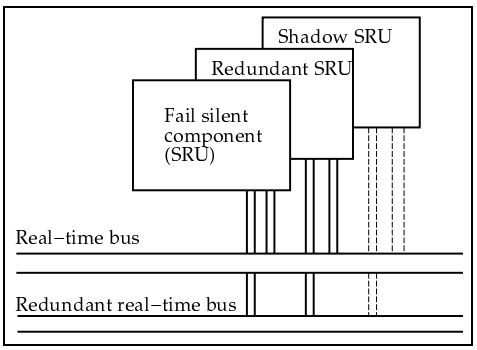


Figure 7: MARS architecture.

The Stratus system uses software recovery techniques to minimize system crashes due to system bugs [16].

5.3 MARS

As mentioned in Section 2, failures can also be caused by a delay of correct behavior, timing failure. In the Tandem and Stratus systems, this kind of failures are usually not considered too critical. In some real-time applications, timing failures can be just as disastrous as other kind of failures. One fault-tolerant system where timing failures are of major concern is the MARS computer system, under development at the Technical University of Vienna. MARS is an architecture for the realization of computer systems for distributed fault-tolerant real-time applications. This is not a commercial system yet, but its planned industrial applications include rolling mills and railway-control systems, in which the controlled system imposes hard deadlines [10].

Hardware

In general, distributed systems are better suited for fault-tolerance than central systems. If we consider the system as operational as long as just some nodes are available (as opposed to having all nodes available), the fault-tolerance capability exists without extra cost, if the appropriate software

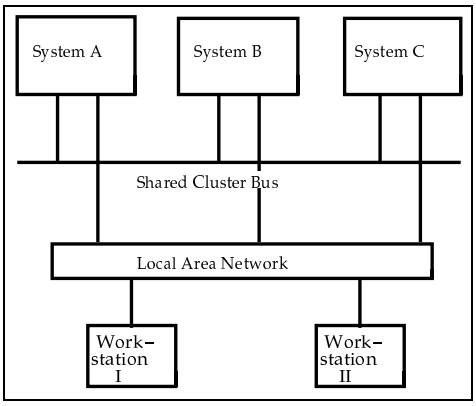


Figure 8: Highly Available Cluster.

mechanisms are used (discussed by Hariri et al [6]). In distributed systems, *smallest replaceable units* (SRUs) can be defined [18]. Two or more fail-silent SRUs can be grouped together and form a *fault-tolerant unit* (FTU). In the MARS system, a FTU consists of three SRUs. One of them work as a "shadow", and transmits nothing on the bus as long as none of the two other SRUs fail. After the failure, the shadow SRU replaces the failed SRU, see Figure 7. As shown on the figure, the SRUs communicate over a duplicated bus, based on Ethernet cables. A TDMA[2] communications protocol is used instead of CSMA/CD to get bounded communication times.

Several FTUs are clustered, and these clusters are connected to form the whole distributed system.

A SRU in the MARS system consists of two independent parts: an application unit and a communication unit. Both are based on off-the-shelf components. To provide error detection, parity bits are used on RAM and FIFOs. A watchdog timer is used on the communication unit.

Software

To be able to have bounded execution times, a very small micro kernel operation systems is used. Software fault-tolerance have to, mainly, be implemented by the application software. According to Kopetz et.al. [11], an architecture for hard real-time applications must be complemented by an appropriate methodology to achieve predictable timing behavior. As a result of this, the MARS design environment has been developed, which helps in design and verification of applications.

5.4 Fault-Tolerance on Clusters

Highly available systems can be made from general-purpose computers with no extra hardware except duplicated interconnection network. The problem with general-purpose machines is that they are not fail-fast, but this can be solved with an abstract machine on top of the hardware.

Several research groups have done studies employing computer clusters[3]. In this report, we present two examples of this approach, the *HA (Highly Available) Cluster*, and ClustRa, a ultra high availability database system based on ordinary workstations.

5.4.1 The HA Cluster

The HA Cluster [2], developed at IBM Israel Science and Technology, provides highly available data through replication of data on a cluster of machines. The cluster consists of two or more machines interconnected by a high-speed bus (see Figure 8). In the prototype, IBM AS/400 machines are

[2]TDMA = Time-Division, Multiple-Access.

[3]A cluster is multiple machines interconnected by a high speed network.

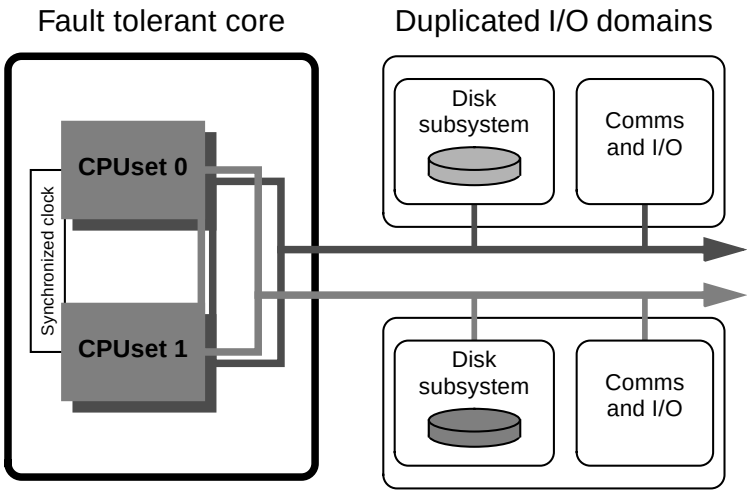


Figure 9: Netra ft 1800 principles [19].

used, connected via a pair of fiber-optic high-speed buses (to get redundancy). Workstations and terminals are connected to the cluster via a local area network.

To get data redundancy, data is replicated at a primary and one or more backup sites. When a site fails, data has to be retrieved from one of the backup sites, and a recovery process has to be conducted when the failing site is repaired. This is not a trivial process, and how to do this efficiently is currently a hot research area. The high availability is transparent, which means that neither applications nor databases need to be changed. The prototype has been tested with a commercial database application, with acceptable throughput.

5.4.2 ClustRa

ClustRa is a database engine originally developed for telephony applications [20, 8]. In addition to high availability, these applications require high throughput (a high transaction rate), and real time response time.

The ClustRa hardware platform is a cluster of workstations, interconnected with a communications network, e.g., ATM. A node is the unit of failure, and if a component inside a node fails, the node can be replaced while the rest of the system is operating. High availability is achieved by replication of data. To handle more serious failures, e.g., environmental failures, the system can be spread over two sites (e.g. one in Trondheim and one in Oslo).

5.5 Sun Netra ft 1800

Even though the trend now is to achieve fault-tolerance by employing ordinary workstations in clusters, single node fault-tolerance is still interesting. The most recent product of this category is the Sun Netra ft 1800 server [19]. Sun Netra ft 1800 is a telecommunication platform, specifically designed for mission critical applications that cannot be down even for short periods, and provides class 5 availability, e.g., > 99.999 percent hardware availability.

Hardware Sun Netra ft 1800 is based on Sun's Ultra Enterprise 450, but have eliminated all single points of failure. Processing, storage and I/O elements are duplicated (Figure 9), and can be replaced while the system is operating, i.e., hot swapping. The power supply module is also duplicated, but no UPS function is provided.

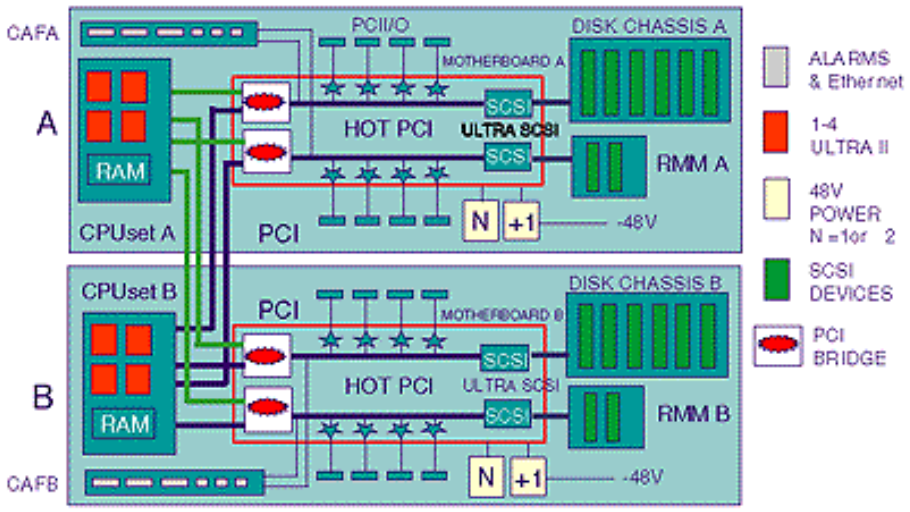


Figure 10: Netra ft 1800 system architecture [19].

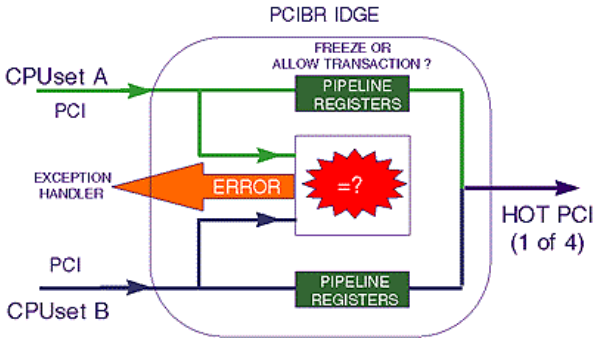


Figure 11: CPU set error checking process [19].

As illustrated in Figure 10, each CPU set can have 4 UltraSPARC processors. Main memory is organized into 4 banks of 4 modules, and data in storage and on bus is ECC-protected.

The CPU sets have synchronized clocks, and runs in tight lockstep. Error checking is done as part of an I/O-transaction (comparing for errors at each clock cycle would have negative impact on performance). The CPU set error checking process is illustrated in Figure 11. After a failed CPU set has been identified and replaced, it is re-synchronized with the other CPU set. Re-synchronization involves copying the processor, register, and I/O states to the new CPU set. This is done while the other CPU set operates as normal. This is important, as the re-synchronization can be time consuming, e.g., copying the memory can take several minutes.

Software The Sun Netra ft 1800 runs a standard Solaris kernel. The kernel runs on top of a fault free virtual machine interface, illustrated in Figure 12.

Achieving > 99.999 percent availability would be impossible if the system must be turned off to perform upgrades. Software upgrades typically takes at least an hour or more, and might be needed more than once a year. 5 minutes downtime would be impossible to achieve under such circumstances. To avoid software upgrade downtime, Sun Netra ft 1800 supports a feature called *split mode*, in which one half of the server can run the applications, while the other is upgraded and tested.

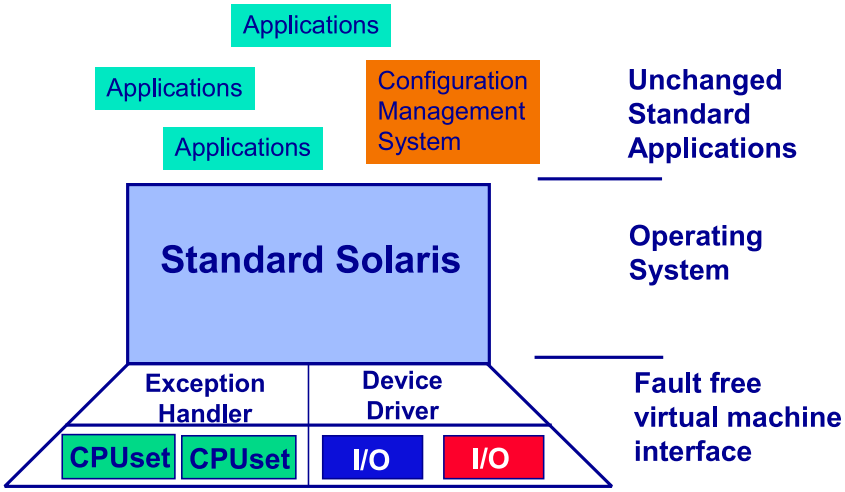


Figure 12: Netra ft 1800 software architecture [19].

5.6 Other Systems

The systems described above represent typical fault-tolerant systems, but many other fault-tolerant computer systems exists, or have existed. Some are, or have been, commercially available, like Sequoia, BiiN, NCR 9800, CCI Power6/32FT, and Parallel XR650, while others are only research computers.

6 Summary

Fault-tolerance techniques will become even more important the next years. The ideal, from an application-writer's point of view, is total hardware fault-tolerance. Trends in the market, e.g. Stratus and Sun Netra, shows that this is the way systems go at the moment. There is also, fortunately, reason to believe that such systems will become considerable cheaper than today. Technology in general, and miniaturization in particular (which leads to physically smaller and in general cheaper systems) contributes to this. Much research is also being done with clusters of commercial general-purpose computers connected with redundant buses. In that case, the software has to handle the failures. However, as shown with the HA Cluster and Sun Netra, that could also be done without affecting the user programs and applications.

7 Bibliographic Notes

A short introduction to the area is given in the article *High-Availability Computer Systems* by Gray and Siewiorek [5]. Other introductory articles are [13] and [3]. A comprehensive guide to the design, evaluation, and use of reliable computer systems is the book *Reliable Computer Systems: Design and Evaluation* by Siewiorek and Swarz [16]. This book also includes case studies and is the "guru" book in this area. An article by Siewiorek [15] gives a more "compressed" presentation of the commercial computers presented as examples in the book. For a more "lightweight" introduction than the book by Siewiorek and Swarz, *Design and Analysis of Fault Tolerant Digital Systems* by Johnson [9] is recommended. Software fault-tolerance through transactions is studied thoroughly in *Transaction Processing: Concepts and Techniques* by Gray and Reuter [4]. For probing further, the other material cited in this report can be consulted, together with the web pages of the relevant commercial companies.

References

[1] A. Avižienis and J.-C. Laprie. Dependable computing: From concepts to design diversity. *Proceedings of the IEEE*, 74(5), 1986.

[2] A. Azagury, D. Dolev, G. Goft, J. Marberg, and J. Satran. Highly available cluster: A case study. In *1994 IEEE 24th International Symposium On Fault-Tolerant Computing*, pages 404–413, 1994.

[3] W. G. Cuan. Fault tolerance: Tutorial and implementations. *Computing Futures (supplement to IEEE Computer November 1989)*, 22(Inaugural issue), 1989.

[4] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques.* Morgan Kaufmann Publishers, 1993.

[5] J. Gray and D. P. Siewiorek. High-availability computer systems. *IEEE Computer*, 24(9), 1991.

[6] S. Hariri, A. Choudhary, and B. Sarikaya. Architectural support for designing fault-tolerant open distributed systems. *IEEE Computer*, 25(6), 1992.

[7] Y. Huang and C. Kintala. Software implemented fault tolerance: Technologies and experience. In *1993 IEEE 23th International Symposium On Fault-Tolerant Computing*, pages 2–9, 1993.

[8] S.-O. Hvasshovd, Øystein Torbjørnsen, S.-E. Bratsberg, and P. Holager. The ClustRa telecom database: High availability, high throughput, and real-time response. In *Proceedings of the 21st VLDB Conference*, 1995.

[9] B. W. Johnson. *Design and Analysis of Fault-Tolerant Digital Systems.* Addison-Wesley, 1989.

[10] H. Kopetz, A. Damm, C. Koza, M. Mulazzani, W. Schwabl, C. Senft, and R. Zainlinger. Fault-tolerant real-time systems: The Mars approach. *IEEE Micro*, 9(1), 1989.

[11] H. Kopetz, G. Fohler, G. Grünseidl, H. Kantz, G. Pospischil, P. Puschner, J. Reisinger, R. Sclatterback, W. Schütz, A. Vrchoticky, and R. Zainlinger. The distributed, fault-tolerant real-time operating system MARS. *IEEE Technical Committee on Operation Systems and Application Environments NEWSLETTER*, 6(1), 1992.

[12] J.-C. Laprie. Dependable computing and fault tolerance: Concepts and terminology. In *The 15th International Symposium On Fault-Tolerant Computing*, pages 2–11, 1985.

[13] V. P. Nelson. Fault-tolerant computing: Fundamental concepts. *IEEE Computer*, 23(7), 1990.

[14] R. Scott, J. Gault, D. McAllister, and J. Wiggs. Experimental validation of six fault-tolerant software reliability models. In *The 14th International Conference On Fault-Tolerant Computing*, pages 102–107, 1984.

[15] D. P. Siewiorek. Fault tolerance in commercial computers. *IEEE Computer*, 23(2), 1990.

[16] D. P. Siewiorek and R. S. Swarz, editors. *Reliable Computer Systems: Design and Evaluation.* Digital Press, 1992 (1st edition published 1982).

[17] M. Singhal and N. G. Shivaratri. *Advanced concepts in operating systems: distributed, database, and multiprocessor operating systems.* McGraw-Hill, 1994.

[18] A. Steininger and J. Reisinger. Integral design of hardware and operating system for a DCCS. In *10th IFAC Workshop on Distributed Computer Control Systems*, 1991.

[19] Sun Microsystems, Inc. Netra ft 1800. White paper, 1999.

[20] Ø. Torbjørnsen and S.-O. Hvasshovd. Application of an abstract machine supporting fault-tolerance in a parallel database server. *International Journal of Computer Systems Science & Engineering*, 9(2), 1994.

[21] J. Wakerly. *Error detecting codes, Self-Checking circuits and Applications.* North-Holland, 1978.