

**Fault Tolerant Computing
in Industrial Automation
Hubert Kirrmann
ABB Research Center
CH-5405 Baden, Switzerland**

2nd Edition 2005

Abstract:

Fault-tolerant computing encompasses the methods that let computers perform their intended function or at least keep their environment safe in spite of internal errors in hardware and software.

This tutorial on fault-tolerant computing is focussed on industrial automation in general and embedded computers in particular. It describes the computer architecture and the software methods used.

It is divided into 9 sections:

Definition of reliability, availability and safety with their metrics

Behaviour of plants in presence of computer malfunction, and derived requirements

Detection and correction of errors

Dependable computer architectures for safe and reliable applications

Recovery methods

State saving and recovery

Database recovery

Standards

Dependability calculations

0 Introduction

Industrial processes, power generation, transmission and distribution, banks, airlines, railways and numerous other applications rely increasingly on computers for their daily operations. Computer failures are bound with heavy economic losses and in some cases even with danger to life and limb.

The reliability of computers has been traditionally kept high by proper design, testing and installation, preventive maintenance procedures and by a comprehensive field service. The improvement in computer reliability obtained by these traditional methods is considered insufficient in many new installations, especially since computers are increasingly trusted for critical positions formerly filled by humans.

Since technology and the laws of physics limit the reliability of the components, a reliability increase can only be achieved by embedding redundant elements. Computers that incorporate redundancy for protection against failures are known under the generic name of fault-tolerant computers.

A fault-tolerant system has the unique property that its overall reliability is higher than the reliability of its constituting parts. The typical question of fault-tolerance is "Can a reliable bridge be built with weak beams?" and the technical answer is "In principle yes, if enough beams are available and they are correctly mounted". The secret of fault-tolerance is how to structure these redundant beams so that the failure of one does not bring the whole bridge down.

A computer failure manifests itself by wrong or missing data. Depending on the application, one or the other may be more dangerous or costly. In some applications, neither wrong nor missing data is tolerable. So, before speaking of tolerance of the computer, one should consider the tolerance of the plant controlled by a computer with respect to failures of that computer. After all, if someone breaches the law (and gets caught) it is not him, but his judge that must be tolerant.

Fault-tolerant computers incorporate a certain amount of redundancy (duplication, triplication of most parts). Depending on the application, it could be more important to use this redundancy to detect errors (so as to prevent wrong output) or to keep on working (so as to prevent missing data). In fact, error detection and correction are closely related.

Over the last years, the interest in fault-tolerant computers has been cyclic: every now and then, there is a growing attention for them and a few years later, the fever vanishes and the interest focuses on simplex designs again. The times of greatest interest in fault-tolerant computers coincided with the application of computers in new domains which were formerly handled by other means: the first worry of the future users at the introduction of a computer in their domain concerns the dependability of the machine: "what happens if it fails?" Usually, the users are over-cautious and set the requirements so high that they can hardly be fulfilled with the technology available at the time of introduction. The minimum requirement is that the new system must be more reliable than the one it replaces.

For instance, the early computers were equipped with thousands of vacuum tubes which failed at the rate of about one every hour. These designs had no other remedy than to use fault-tolerance. As technology improved, as transistors supplanted vacuum tubes as computing elements, the additional price of redundancy was not justified any more, and simpler computers were built. The same game repeated itself with the introduction of computers in the space exploration, and with the transition from transistors to integrated circuits.

Telephone exchanges and control centres have been using fault-tolerance techniques to achieve a very high availability for decades. These applications tolerate short disruptions and occasional data or communication losses but service must be restored quickly.

Computers have monitored industrial plants for many years. However, they were only seldom used in critical applications, and direct digital control is always backed-up by some analog protection mechanism. This situation will change in view of the increasing complexity of the industrial processes. In power plants, the number of input and output sensors has risen from a few 100 to over 10000 within a few years. The processing of that huge amount of data would require some 200 persons to handle it. Although the decisions are still taken by the operator, computers are already today irreplaceable for monitoring the process and transmitting orders. The reliance on the computer is large: the operator has to trust that the measurements shown by the computer are correct. Although usually the plant can be shut down safely in case of damage to the computer, the economic losses are so high that fault-tolerant computers are required to reduce the frequency and duration of computer-originated downtime.

As industrial plants become more and more complex, the operator will be unable to process the incoming information and react in time. Alarm situations can today produce 60 pages of display within 15 s. Process control computers will have to take decisions by themselves, and enter into areas which were traditionally reserved for the

operator and highly specialized analog and electromechanical devices, like those used in power line protection. Unattended plants will become common, as difficulties increase in finding operators to man them.

In airplanes, the digital autopilot is taking increasing responsibility with every new design. Since the Airbus A320, the aircraft cockpit is digital, the only analog instrument remaining being the mercury ball. The pilot has to rely on the correct displays and the correct forwarding of his commands. The joystick has ceased to have any direct effect on the control surfaces: it is replaced by "side-sticks" similar to the one used in videogames and all commands are relayed digitally over buses like MIL-1553. But even with this "fly-by-wire" technique, the pilot is still in the control-loop or can intervene at any time. Temporary malfunctions can still be tolerated, as long as the computer is back on-line quickly. During the landing phase, even short disruptions are unacceptable.

The aerospace industries, and especially the space missions, have paved the way of highly reliable fault-tolerant computers, as the loss of the computer, even for a short time, is equivalent to the failure of the mission. The Saturn V rocket used triplicate guidance computers. The US Space Shuttle relies entirely on its quintupled control computer for the launch and re-entry phase.

The next airplane generation will move the pilot out of the loop and make him an observer of the computer. The increasing price of energy will bring new airplane designs in which drag is reduced at the expense of loss of the natural stability. For instance, in classical airplanes, the tail's surface does not contribute to lift; on the contrary, it dips the airplane to ensure stability. Some future planes will have the tail acting as a lifting surface, and the centre of gravity will be moved between wings and tail. The plane will then depend on the correct function of the control computer to fly at all, since no pilot could stabilize such a plane by hand. The failure of the control computer would cause the loss of the airplane. The acceptance of such a dependency by the pilots is, however, another matter, although computer-controlled ground following systems that guide the plane just above tree-level have been successfully introduced.

Will the up-and-down of fault-tolerant computers continue in the future? As the technology of integrated circuits progresses, the number of transistors increases at the same rate as the reliability of the individual transistors, so overall reliability reaches a ceiling. There has been even a certain slackening of the reliability of some parts as competition heats up and prices drop. Problems like sensitivity to cosmic rays and trapped alpha-sources in the packages set physical limits to the reliability. On the other hand, hardware costs have dropped enough so as to make redundancy cheap. The complete replication of a computer just for the sake of fault-tolerance would have been unthinkable 20 years ago. Fault-tolerance features only represent today a few percent of the total cost of an industrial control system. It becomes unacceptable to let the function of a complete plant depend on a single integrated circuit.

Fault-tolerant computers are not going to disappear again; rather, all future computers will incorporate fault-tolerant features to an increasing degree. Users have been used to some fault-tolerant features and will continue to ask for them. More than twenty companies are offering dedicated fault-tolerant computers for applications such as time-sharing, transaction processing, communications, industrial control, aircraft and space applications. The economical benefits of fault-tolerance are recognized, even if fault-tolerance only serves the purpose of reducing the repair time. Fault-tolerance is now an established discipline of computer architecture.

The human aspects of fault-tolerant computing are not treated here, but deserve a remark. As the computer assumes more and more routine chores, the human operator is only required in infrequent emergency situations and can be quite bored for the rest of the time. The very presence of a human person in critical applications is disputed. The statistics of aircraft crashes show that human factors can become a major uncertainty factor, although the statistics do not show how many accidents humans avoided. The danger is rather that the introduction of reliable computers could slacken the sense of responsibility of the humans around it, who will trust them more and more, until their reliance becomes excessive.

Summary

This tutorial explains the basic concepts and methods in fault-tolerant computing. It is intended as an introduction to the state of the art and to the understanding of the current literature on this topic, which is best represented by the annual International Conference on Dependable Systems and Networks (www.dsn.org)¹.

The topics explained in this tutorial are represented in Table 1:

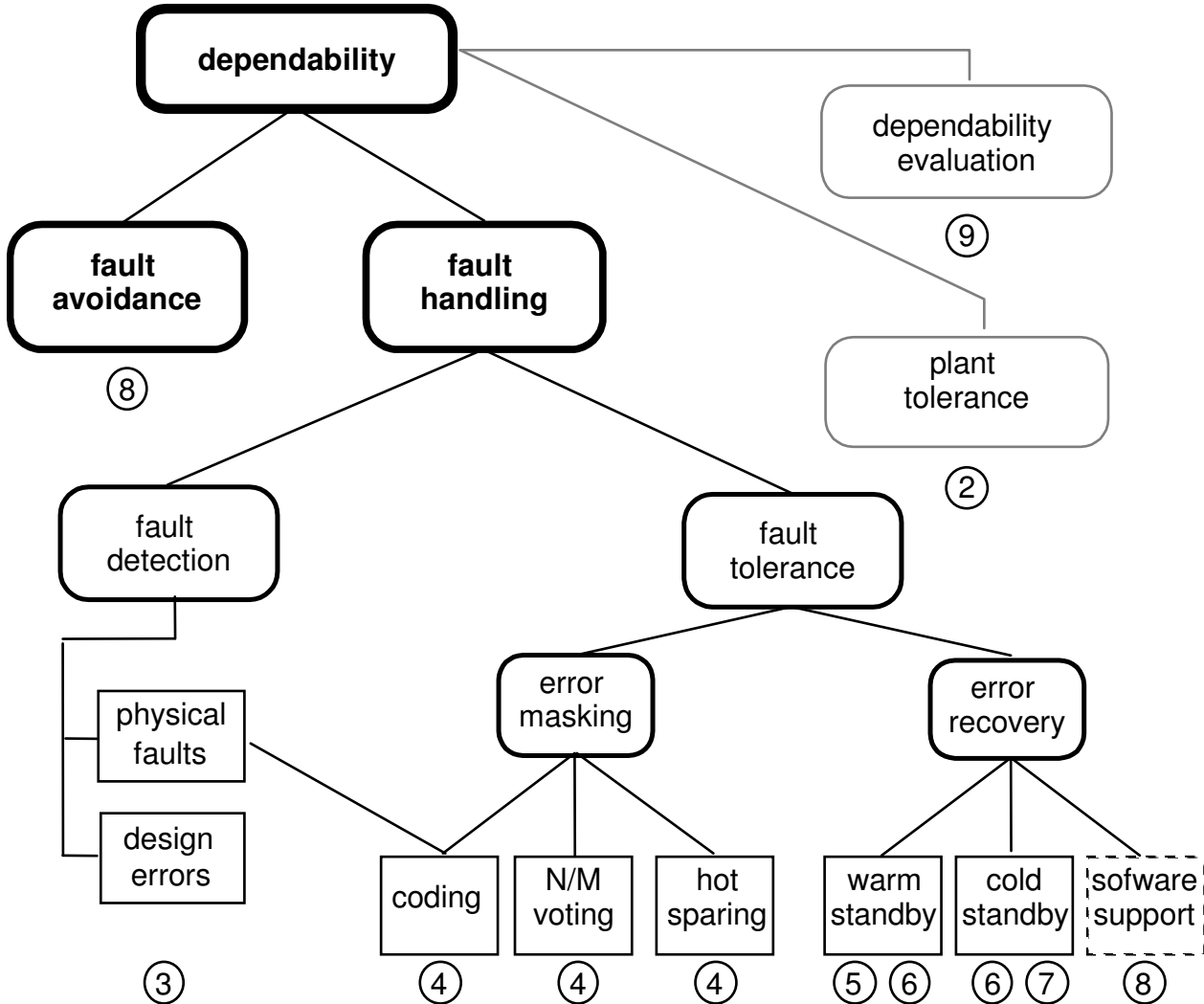


Table 1: Topic Tree of this Tutorial.

Chapter 1 introduces the **notions** and **definitions** of reliability, availability, redundancy, fault-tolerance and reconfiguration with little maths – Chapter 9 repeats these notions under the maths aspect.

Chapter 2 studies the **impact of faults** and **malfunction** of a computer on its environment. Several requirements are deduced from the tolerance of a plant to malfunctions of the computer that controls it. Three classes of computers emerge, corresponding to high integrity, high availability and high reliability.

Chapter 3 explains the kind of **errors** that occur in computing systems, how to **detect** them and how to **survive** them. The fault-tolerant computers are classified according to the principle they use to maintain continuous operation: **work-by** (replicated execution on several processors), or **stand-by** (updating of the spare by the working unit).

Chapter 4 describes **coding** and **work-by** techniques in which a number of processors execute the same task in parallel, with emphasis on the techniques for coding, massive redundancy, voting and dual hot-sparing. The problem of **synchronizing** work-by units is considered in detail.

¹ This conference is a successor to the Symposium on Fault Tolerant Computing (FTCS) sponsored by the IEEE Computer Society, and to the Working Conference on Dependable Computing for Critical Applications (DCCA) sponsored by the IFIP.

Chapter 5 describes the techniques used in **recovery**, in which the spare unit is maintained by regular updates rather than by replicated computing. The methods used to continue computation in case of error are discussed. The difference between **backward error recovery** and **forward error recovery** is outlined and the impact on the outer world is considered. A scenario of recovery is developed.

Chapter 6 considers **state saving and restoring** for recovery. It begins with a model of computation and a model of storage. The storage is divided into volatile storage and stable storage, the outer world into reversible and irreversible. The methods used in saving and restoring the volatile state, in restoring the stable state and in treating interactions with the environment are presented.

Chapter 7 applies the concepts of Chapter 6 to the recovery of **databases**. This Chapter is devoted to software techniques that serve as a second defence against hardware faults.

Chapter 8 considers the **standardisation aspects** of fault-tolerant programming, and especially with conformance testing and certification.

Chapter 9 explains the mathematical tools for the **evaluation of reliability and availability**. The analysis goes to the depth required to evaluate simple systems.

The Appendix lists the most common fault-tolerant computers available on the market in 1986. Some of these products may already have disappeared from this active market, but they are still mentioned under the premise that technical excellence is no guarantee for commercial success.

A second Appendix presents a glossary of the most important terms and their definition.

Each Chapter closes with a list of **references**.

1 Definitions and principles

1.1 Definitions

Definitions may often seem self-evident. However, the intuitive notion everyone has of a term may not be appropriate in the context of reliability. The word "fault", for instance, has other meanings in theology, geology, metallurgy or orthography than in this context. Especially, when using such terms as "reliability" and "availability" caution is at place, since their formal definition are somewhat different from their every day's use. For example, the term "availability" in the sentences "the availability of cheap microprocessors..." and "the availability of the plant exceeded 99%" is used with totally different meanings.

The basic principles of redundancy and fault-tolerance are introduced with no special emphasis on computers, as these concepts could be applied to mechanical and electrical devices as well. This should remind that the computer is only one piece of the equipment. To clarify the terminology, a glossary is appended to this tutorial (Appendix 1).

1.2 Mission, Failure, Malfunction, Faults and Errors

Computers, electronic devices, industrial plants, etc. are complex **systems** that depend on the correct function of a large number of **elements** to work properly. We will speak of an "element" when considering a part of a system which we do not want to detail further, although it may well consist itself of sub-elements. When we do not want to detail whether we consider a system or an element, we speak of an **item**.

An item is required to provide a certain service under given conditions for a stated period of time, that is, to fulfil a specific **mission**, defined by a **mission specification**.

Example:

The mission of a car can be defined as: "transport up to 5 persons over roads at a maximum speed of at least 130 km/h while consuming less than 3 l/km over a useful lifetime of at least 20 years".

A **failure** occurs when car is not capable of performing its mission – regardless how significant the deviation from the specification is. It does not need to be an accident.

Example:

It is not a failure of the car if it can't transport people over railroads, but if the car consumes more than the amount specified, this can be considered a failure with respect to the specification.

Therefore, the definition of a failure is bound to the idea of a contract, and to the notion of quality. Success or failure, like quality, is not a property of an element, but a point of view of an external user. For any analysis, one should first quantify exactly what one understands as a failure.

A general definition of a failure is:

"A **failure** is the termination of the ability of an item to perform its required function"

(by definition of the IEC- International Electrotechnical Commission).

The above definition supposes that the mission specification includes a yes/no criterion like reject/accept. Complex systems exhibit a variety of failure modes, some major, some minor. In these cases, one should define service classes or performance levels, as we shall see below.

The above definition makes no assumption about the duration of the failure: it rather expresses a transition. "Failure" can also express a state. When it is necessary to make the distinction, we will use "failed" (for the state) and "failure" for the event.

An item can cease to provide the required service during a certain time, but return to service shortly afterwards, either because the cause of the disruption (for instance an external disturbance) disappeared by itself or because the item was repaired, either by its own means or by an external action.

"A **malfunction** is a temporary disruption of service."

This disruption is not necessarily caused by damage. In the power utilities, an **outage** is the inability to supply electrical energy – this may be due to network instability.

When the malfunction becomes longer, for instance when a repair must take place, or when the failed state is definitive, one speaks of a **breakdown**. The maximum duration of a malfunction is part of the mission specification: if it is exceeded, then a mission failure happens.

Example:

In air traffic control system, a temporary disturbance of a few seconds could be acceptable, if the radar can catch itself up and generates correct pictures again. A malfunction of ten minutes is clearly unacceptable.

We are interested in the origin of a failure.

"A **fault** is the cause of a failure."

There are two kinds of faults: those due to an external action or aging (physical faults) and those due to a poor design (design faults). A fault in a system does not necessarily lead to a failure of the system, if the fault can be handled properly, e.g. if the system is **fault-tolerant**, or if the fault was only temporary.

If the considered device is a logical machine, then a mission failure can be caused by an error.

"An **error** is a departure from the intended state of a data item, and by extension an incorrect behaviour of a logical machine."

An error can be caused by incorrect algorithms (design faults), or by faulty logical elements (physical faults).

Although the ultimate mission of a computer depends on the application, one can generally state its mission as:

"The mission of a computer is to produce the intended data and only them in due time"

One distinguishes two failure modes of a computer, the integrity breach and the punctuality breach. These two failure modes have different impact according to the computer's role, as we shall see in Chapter 2.

Integrity breach: the computer generates erroneous data (in addition to the correct data).

This failure, that is analogous to lying, is generally considered as the worst, but this may not always be true: the computer produces incorrect data, which have not been recognized and filtered out as such. The merit of a computer with respect to the correctness of its output data is its **integrity**, which is given by the probability that erroneous data are prevented from leaking out to the environment. A computing system that rather stops than output erroneous data is **fail-stop** or **fail-silent**. The fail-stop element is the basis for building integer systems and in general for fault-tolerant systems.

Punctuality breach: the computer is not providing the required data at the requested time.

A malfunction may lead to a failure or not depending on its duration. The mission is considered as failed if the maximum permitted malfunction duration is exceeded, i.e., if the required data are not delivered in time or not at all. The probability that the data are in time is expressed by the **punctuality** of the computer. A computing system that can continue punctual service in spite of faults is called a **persistent** system.

1.2.1 Reliance and Dependability

"**reliance** is a measure of the user's dependence on the correct function of an item."

The higher the reliance, the worse the consequences of an incorrect function [Randell 78].

"**dependability** is quality of the delivered service such that reliance can justifiably be placed in this service" [Laprie 85]

Dependability is a subjective measure that encompasses all aspects related with the eventuality of failures.

Example:

Home users of personal computers know that their drives are not reliable. They back-up them regularly, and when they remain in the word processor for a long time, they save the work to disk before taking a break and also every now and then. Their reliance on the PC is low.

On the other hand, users of servers rely heavily on the automatic back-up and journaling. They do not make back-ups and a major server breakdown would have serious economical consequences. The reliance on the network server is greater, and accordingly, it should have a greater dependability.

1.2.2 Reliability

"The **reliability** of an item is the probability that this item provides the required service under given conditions for a stated period of time"

(By definition of the International Electrotechnical Commission)

Reliability is the probability that a given mission be fulfilled. Reliability depends on many factors, such as manufacturing, testing, environment stress, and in particular on the mission duration. Reliability is normally expressed in function of service time, and as such, it is a function that always declines with time - all things abide. The most important factor of the environment is temperature, the second being mechanical vibrations (Figure 1-1).

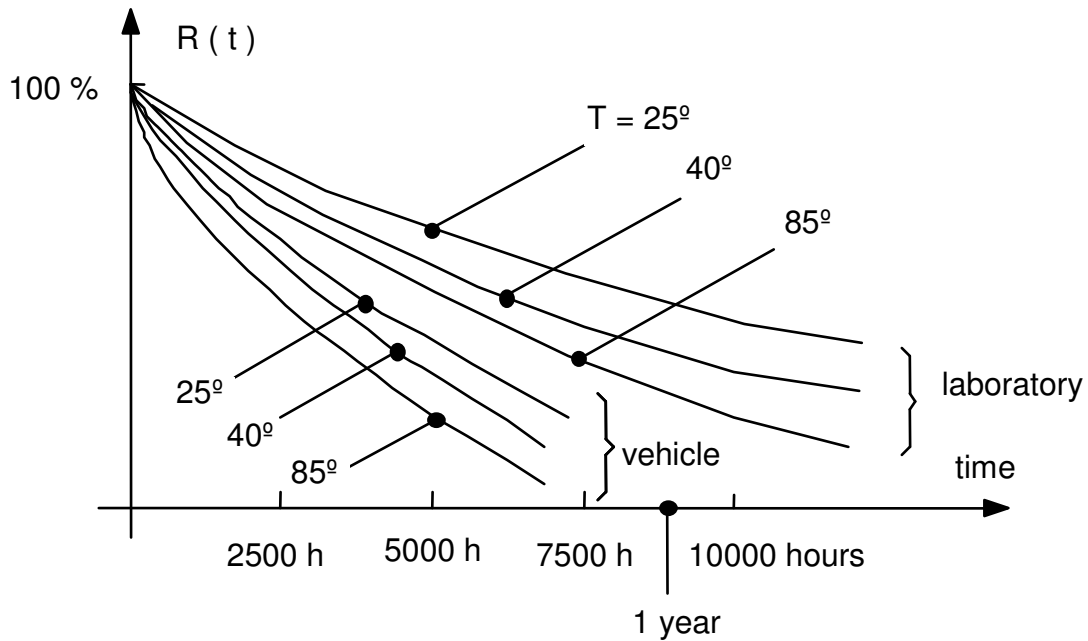


Fig. 1-1: Reliability in Function of time for one Item, with temperature as parameter.

Reliability can be expressed by an analytical function of time and other factors, but for practical purposes, one prefers to express it by a single number, for instance, by the **Mean Time To Failure** or **MTTF**.

The **failure rate** of an item is the probability that the item may fail in the next small time interval.

The **MTTF** is the expected average time during which the item will perform without fault.

Example:

the MTTF of a light bulb is 10000 hours. Its failure rate is 1 per 10000 hours, or 10^{-4} h^{-1} , meaning that there is one chance in 10'000 that it fails during the next hour.

As Chapter 9 will explain, the MTTF is not always an adequate measure for fault-tolerant systems.

Some prefer to express reliability as the probability of accomplishing a certain mission time without failing. This deserves an explanation.

Example:

The space probe Voyager was designed to fulfil a mission of four years on way to Jupiter and Saturn with a probability of success better than 0.95. The spacecraft has in the meantime exceeded its mission time and left Pluto behind, exceeding its specification.

Or conversely, one can express the reliability by stating which mission time can be reached while maintaining an acceptable reliability level (ARL).

Example:

Figure 1-2 shows that item 2 is more reliable than item 1 for the prescribed mission time, and item 1 has a longer mission time than item 2 for the prescribed ARL. As will be explained later, item 2 incorporates redundancy while item 1 does not.

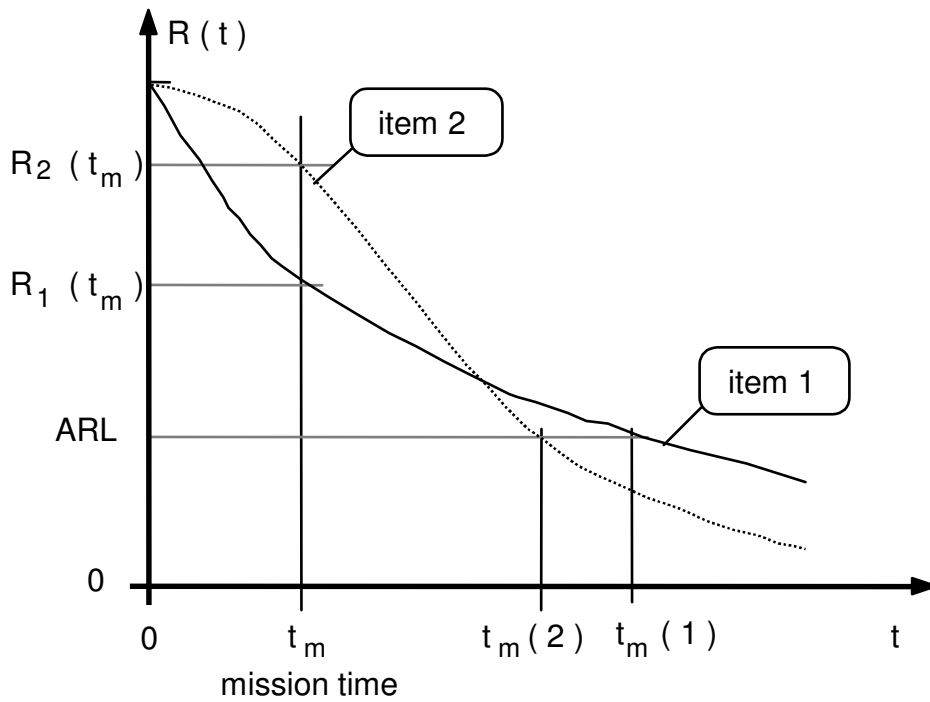


Fig. 1-2: Reliability in Function of Time for Two Different Items.

When a system depends on a certain number of elements for its function, the reliability of the system is the product of the reliabilities of the individual parts, which is always lower than the lowest elemental reliability. This is expressed in the saying: "any chain is weaker than the weakest of its links". We will see that fault-tolerant systems can, contrarily to that saying, achieve a higher reliability than the reliability of their individual elements, since some of their elements are ordered as parallel "chains".

1.2.3 Availability

Once an item fails, one of two things can happen: either the item is discarded (such as a transistor) or the item can be repaired (such as a car - usually). A repairable item functions during a certain time, called **operational** or **up time**. When it ceases to function, it is repaired. The duration of the non-operational time is called **down time**. According to our previous definition, down time is a malfunction.

The reliability of the item defines the **Mean Up Time (MUT)**, which is the mean time during which the item is operational after a repair. The MUT is for our purposes identical to the above MTTF (mean time to fail).

The **maintainability** defines the quality of the repair team or organization. It is expressed by the mean repair time or **Mean Down Time (MDT)**. The term **Mean Time To Repair (MTTR)** is sometimes used in place of MDT, or to express the fact that the down time is divided into a fixed latency period and a repair period.

The interval between two successive failures is called **Mean Time Between Failures (MTBF)**, which is equal to $MUT + MDT$.

A repairable item is defined by its **availability**.

Stationary availability is defined as the relation of the sum of all operating times to the useful lifetime.

$$A = \text{availability} = \lim_{t \rightarrow \infty} \frac{\Sigma \text{ up times}}{\Sigma (\text{up times} + \text{down times})} = \frac{MUT}{MUT + MDT} = \frac{MTTF}{MTTF + MTTR}$$

Availability is frequently expressed in %, or in down time per operation time; more intuitively, availability can be expressed by the **unavailability, (1-A)**:

Example:

a telephone exchange is required to have an availability of 99.9994%, i.e. 3 minutes of downtime per year.

Thus, there exists a fundamental distinction between **reliable** items and **available** items. When a reliable item fails, its life ends. When an available item fails, it can be repaired or otherwise returned to service after a relatively short down time. An available item oscillates all its life long between the states "up" (working) and "down" (out of service). We can express the life cycles of these two classes of items in the Figure 1-3:

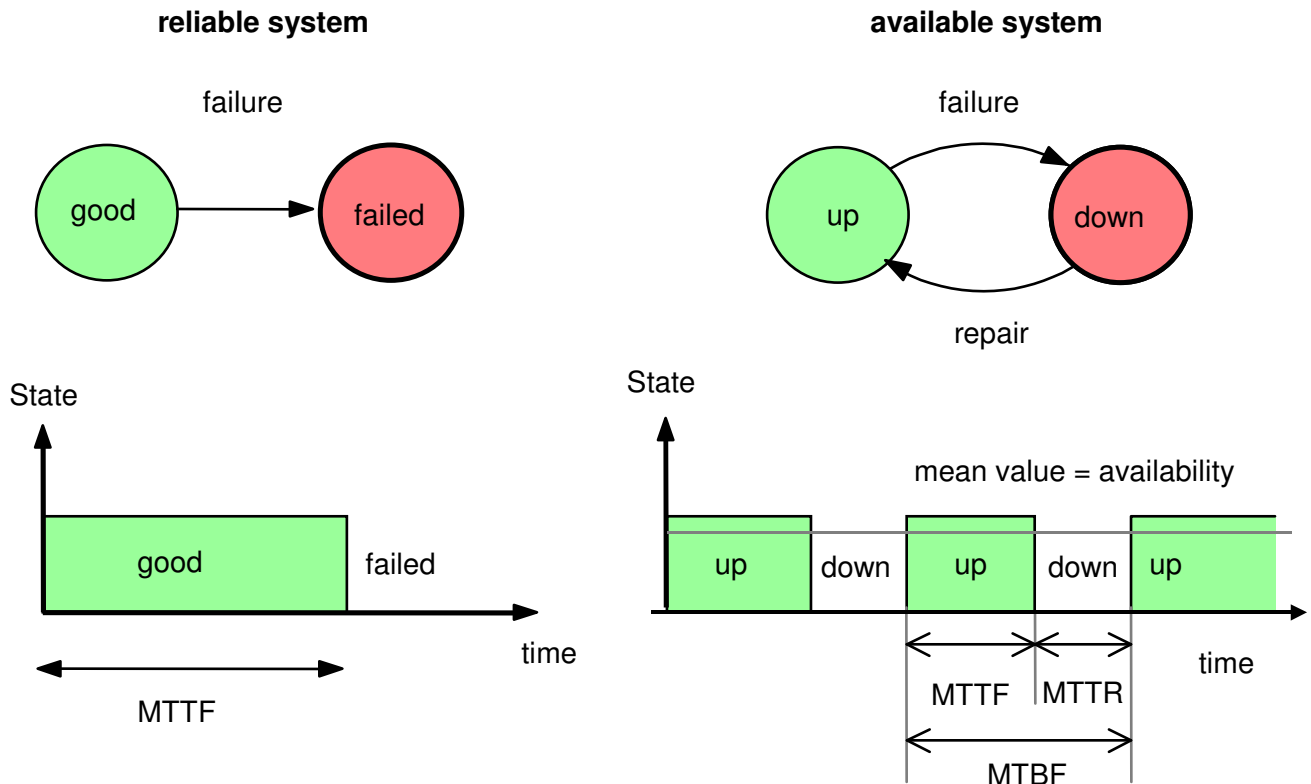


Fig. 1-3: Reliability and Availability

Each of the above states can consist of several sub-states, as we shall see.

Any real system has an end, that is, every system is ultimately a non-repairable system. A car for instance is an available system when considering smaller repairs such as tyre or battery changes, and as long as it does not suffer too high a damage or breaks down into rust. Therefore, when we consider an available system, we consider implicitly only a part of its lifespan, called the **useful lifespan**.

A reliable system is in principle non-repairable, while an available system is in principle repairable. When considering a system, consisting of different elements, the elements may be themselves repairable or non-repairable. Ultimately, available systems consist of reliable elements:

Example:

A light bulb is a reliable element: once it has failed, it cannot be repaired and is thrown away. The lighting of the cellar is an available system: When a bulb fails, one can replace it by another, but the replacement takes some down time.

Conversely, reliable systems may consist of repairable elements - but only if the system is fault tolerant, as we shall see (Section 1.2).

Note that the same system can be both available and reliable, depending on the specification of the mission.

The distinction is best stated in terms of state and transition diagrams. The system transits from one state to the other at every fault or repair transition. A reliable system is characterized by absorbing states, or **trapping states** from which there is no return (or repair). An available system has no trapping state. Note also that the reliable system can also be repaired, as long as it does not run into a trapping state. Therefore, the same system can be reliable or available, depending on whether we consider trapping states or not.

Example:

Figure 1-4 shows the transition diagram of the same system, once considered as a reliable and once as an available system. If one of the two repair transitions would be removed, the available system would become a reliable one.

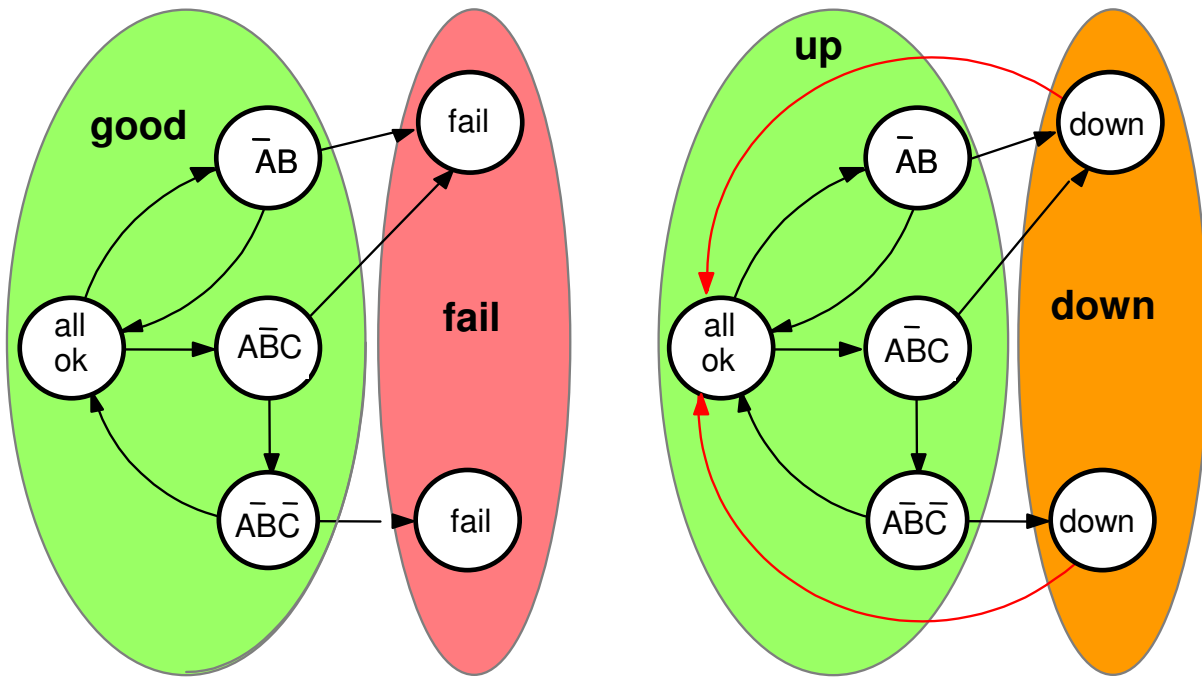


Fig. 1-4: Reliable and Available Systems, Trapping States.

Finally, the distinction between a malfunction in a reliable system and down-time in an available system should be made, although both correspond to a non-function state: In principle, the duration of the down time of an available system has no upper bound, while an excess duration of a malfunction in a reliable system leads to a failure. This distinction will become important in fault-tolerant systems that require a certain time to repair themselves, and therefore produce a malfunction of limited duration. If the fault cannot be handled within the maximum permitted malfunction duration, a failure occurs.

1.2.4 Graceful Degradation

The above notions of availability and reliability base on a yes/no criterion: either the item is up or down, either the mission is performed or not. A more subtle distinction is made when considering that the system may loose only a part of its functionality at each fault, without failing completely. This is termed **graceful degradation**. Graceful degradation supposes that classes of functionality or performance are defined in the mission specification, and that a correspondence is made between classes of faults and classes of performance.

Example:

the functions of a computer in a power-dispatching centre may be divided into important functions, like display and command, and other less important ones like operation statistics. Also, one can specify that the display actualisation must be made in 3 s in the normal case, but accept that this rate goes down to 10 s under degraded conditions. Then, one must specify which percentage of the time a degraded operation is allowed. Finally, the designer of the computing system must meet these requirements by defining the fault modes and assigning to them functions and performance.

Graceful degradation can be expressed by in the state diagram, by weighting the different states in function of the remaining functionality. Gracefully degradable systems exhibit a set of states that correspond respectively to full functionality, to diverse degrees of degraded functionality and to the loss of functionality.

Example:

Figure 1-5 describes a triplicated system, for instance three independent production robots operating in parallel. The total throughput is a function of the number of robots remaining. The availability of each machine defines the average throughput.

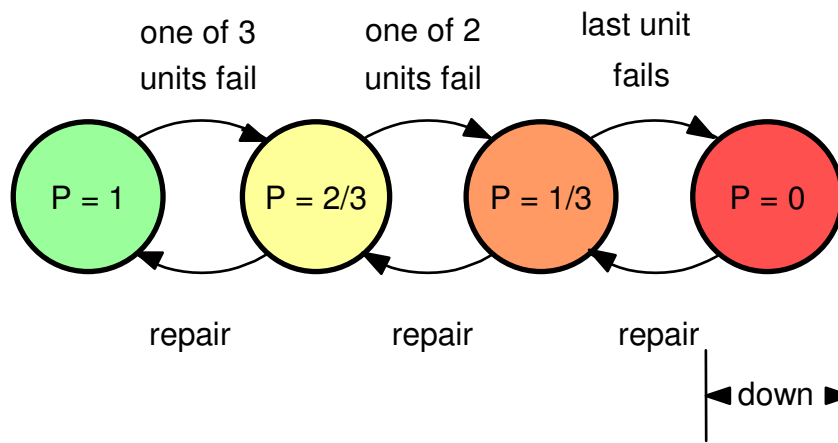


Fig. 1-5: Gracefully Degradable System

In some cases, the objective of graceful degradation is not to continue to provide the original service, but to provide a reduced service sufficient to lead the system to a safe state.

For instance, a twin-motor aircraft that suffers a motor failure should use its redundancy to reach the nearest airport, not to fly its intended route with one motor at reduced speed. The risk of a catastrophic second failure would be too great.

1.2.5 Safety and Security

Safety is the probability that an element does upon a failure enter a state that can cause damages.

Safety is not a function of the element that may fail, but of the function of that element in a system. An element is considered safe if it does not fail in a forbidden way.

Example:

a switch used in a high-power distribution grid is considered to fail safely if it fails in the open position. Its safety is defined by the probability of not going to the closed state upon a failure.

One assumes that an element is safe as long as it does not fail. That is, increasing the reliability always increases the safety. A system is considered fail-safe if it can cope with faults of its elements in such a way that the resulting damages are acceptable. A state diagram similar to that of graceful degradation can express the safety of a system. Some states denote full functionality, others fail-safe and other fail-unsafe states (Figure 1-6).

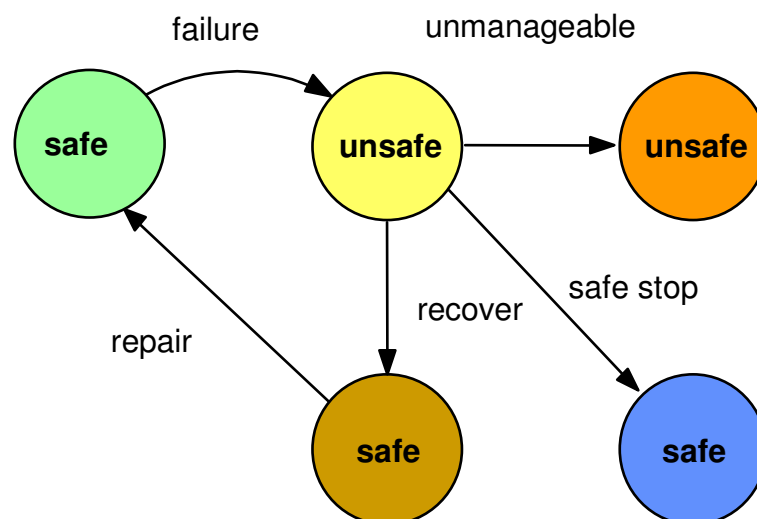


Fig. 1-6: Safe System States

In the above diagram, the system is unsafe when an unmanageable fault occurs, and safe either when the system is working or stopped.

A computer's safe and unsafe fail states depend on the application, as will be seen in Chapter 2. Most of the time, one assumes that the computer is safe if it ceases operation, unsafe if it outputs erroneous data. In this case, the

safety is given by the integrity of the computer. But some plants have no safe side and depend on computer control to remain in a safe state.

Example

if an autopilot is controlling an unstable aircraft, a few erroneous data items would increase the noise of the control system and can be tolerated if infrequent, but a lengthy malfunction leads to a crash. Thus, safety is given by the reliability of the computer.

Safety considerations often dictate the availability of an item.

Example

The space shuttle depends on a flight computer to return to earth. For safety reasons, there are five of them on board, only one being necessary to fly the plane. The probability that all five fail during the same flight is extremely thin. However, that probability rises to unacceptable levels if the shuttle had only three computers, or even only one computer running at take-off. Any malfunction of one of the computers leads to cancellation of the flight, i.e. decreases the shuttle's availability.

Safety and availability are to a certain extent mutually exclusive, they compete for the same resource: redundancy,

Example:

If one replaces the fuses of the electric appliances by copper bars, the availability of the device increases, but the safety is dramatically decreased.

Safety problems occur generally when the trust in the system is excessive.

Example:

Anti-skid brakes have a negative impact on safety since drivers of ABS-equipped cars tend to drive faster, trusting that the ABS will shorten the brake distance, which is not always the case.

Security, finally, is a related notion that concerns the protection against malicious generation or falsification of data and against malicious reading of data. Security in computing systems is generally enforced by physical shielding and data encryption with cryptographic algorithms. In general, increasing the security will reduce the availability, since more elements are present. Increasing the security can also reduce the functionality and render the computer awkward to use. A completely secure computing system is one to which nobody has access. We do not consider security issues here, although they are without doubt a fundamental problem which can hamper the development of computers, and which involves legal as well as technical aspects.

1.3 Increasing dependability

The following section introduces general notions of dependability improvement, which are not limited to the computing field, but apply to a large measure to electronic circuit design and mechanical design as well. These notions will be refined in Chapter 3 with special emphasis on computing systems.

1.3.1 Fault Avoidance and Fault Tolerance

There are two basic philosophies to increase the dependability of a system: **fault avoidance**, which is related to component quality, and **fault tolerance**, which is related to repair. In both cases, one considers the system as consisting of items which can fail.

Fault avoidance considers the system as one chain of links, the failure of any of which causes a system failure. Fault avoidance aims at improving the reliability of the individual links, and especially of the weakest. Fault avoidance is normally the cheapest way of achieving moderate improvements in reliability. The key to fault avoidance is **quality**: improvement of the component reliability, careful and generous design, exhaustive testing, regular maintenance and proper care to the equipment. However, fault avoidance reaches its limit when the required reliability is higher than the reliability of the worst element. As requirements in reliability increase, the cost of reliable elements soars and above a certain level, fault avoidance is not any more cost-effective and fault tolerance should be introduced.

Similarly, if availability is looked for, then fault avoidance includes means to reduce the mean time to repair, but even with a great effort, the mean time to repair cannot be brought down to less than a few minutes, and fault tolerance comes to play.

Since this tutorial is dedicated to fault tolerance, we will not return to the otherwise interesting subject of fault avoidance. We refer for this to the literature: [Siewiorek 82, Spectrum 81].

Fault tolerance on the other hand improves the dependability by replicating portions of the chain, that is, by introducing **redundant** portions that are not needed for fault-free operation. Then, individual links can fail without breaking the chain and the reliability of the chain can be higher than the reliability of the best link, provided one could distribute the load on the remaining links. Fault tolerance requires means to recover from a failure by inserting redundant links as links fail. This can be done by hand or automatically.

Fault tolerance expresses the ability of a system to survive the failure of some of its parts and continue to provide its intended function when this occurs (reliability), or at least a part of that function (graceful degradation).

Now, the difference between a repairable system and a fault-tolerant one is hazy. We assume that a fault-tolerant system is capable of performing a **reconfiguration** without human intervention when a fault of one of its elements occurs, that is, a new distribution of its functions among the remaining parts, with only a short disruption of service or even none.

Although conceptually similar, a repairable system needs a relatively long malfunction time during which it is not capable of performing its function. So, in principle, a system that can be manually repaired within an acceptably short time should be termed a fault-tolerant system, but we will restrict our definition to automatic repair:

A fault-tolerant system can overcome faults of its elements without human intervention and with an acceptably short malfunction, while retaining a part or totality of its functionality afterwards.

1.3.2 Implementing fault tolerance

Redundant functional units, called spares, achieve continuous operation in spite of faults. A spare generally corresponds to a unit that is exchangeable, repairable and replaceable. It is called a field (or line) replaceable unit (RU). Figure 1-7 gives the general outline of a fault-tolerant system:

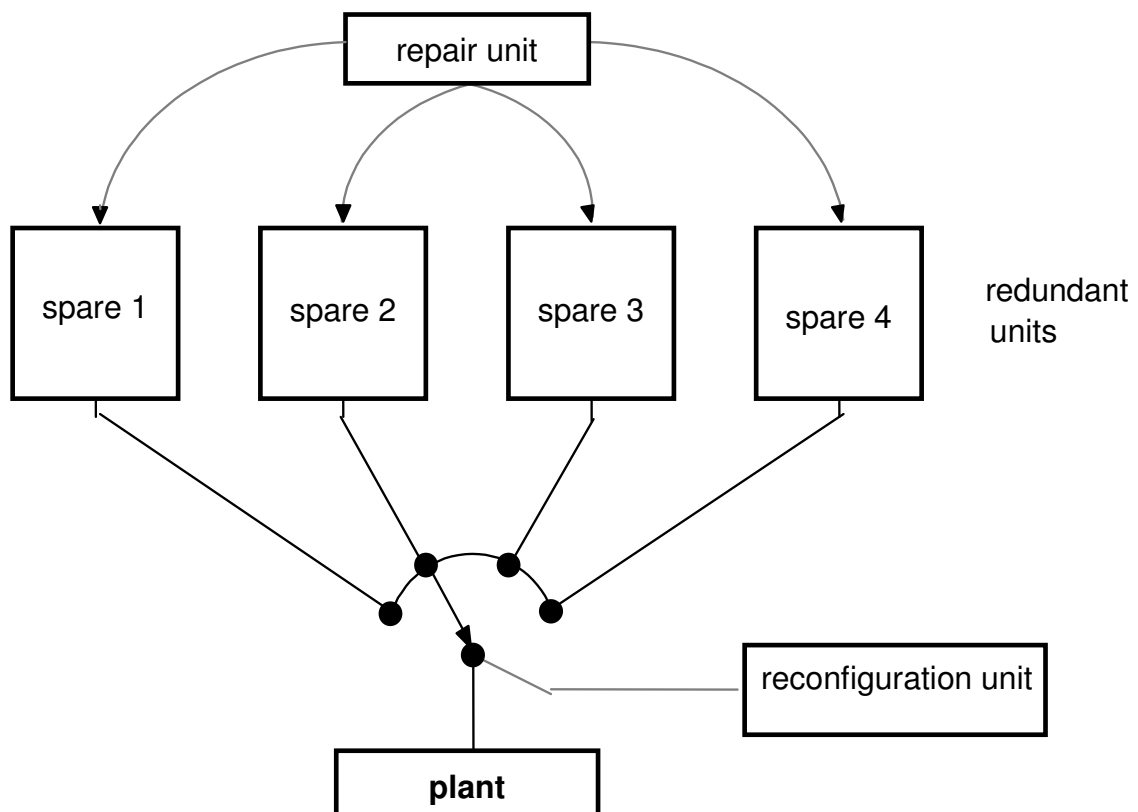


Fig. 1-7: Functional Redundancy.

One (spare) unit performs the intended function. Upon failure of that unit, another unit is switched in its place by the **reconfiguration unit**. A repair unit, if it exists, can repair failed units. The repair unit could imply manual intervention. Repaired units can be **reintegrated** to return the system to its original reliability.

The quantity of spares determines how many faults a fault-tolerant system can survive. Note that a human-repairable system has in principle an infinite number of spares.

1.3.3 Single Point of Failure

A fault-tolerant system should be designed so that it can overcome a fault of any of its elements. This is not always technically feasible nor economically viable. A non-redundant element is termed a "**single point of failure**", since its fault causes the failure of the whole system.

Example:

if three redundant hydraulic equipments are installed on an aircraft, but all circuits are located on the same tail structure, the loss of that structure would rip out all redundant circuits at the same time. However, considering this structure to be reliable enough can be a legal design trade-off.

Single points of failure may be introduced by some mode of failure rather than by the presence of elements that are not physically replicated. For instance, spares should always fail independently. If the fault of one spare causes the fault of another spare, or if the same cause originates the fault of several spares, the system may lose in one fault all its redundancy. We are then in the presence of the feared **common mode fault**, which can defeat any fault-tolerant system. A common mode fault is to be treated as a single point of failure.

Example:

if computers are physically replicated, but use the same program, a bug in that program will bring the system down. A very frequent assumption is that only one fault can take place at a time. A fault during reconfiguration can defeat the recovery mechanism and is therefore also considered as a single point of failure.

1.3.4 Removing a Single Point of Failure

Most fault-tolerant systems use some kind of switching or voting unit like Figure 1-7 shows. This switching unit is a single point of failure. For that reason, one tries to replicate the switching device also, or to move it as far as possible into the controlled process.

Figure 1-8 shows a replicated valve controller. If the outputs of both units go to the valve, then a failure of the valve or the wire to it brings the whole system down. A replication of the valve by a system of 4 valves allows to open or to close the circuit under any circumstances, when either valve circuit is stuck open or stuck closed.

Now, if the system is fail-safe when the valves are closed, then a simple series connection is sufficient. If it is fail-safe when the valves are open, then a parallel connection is sufficient.

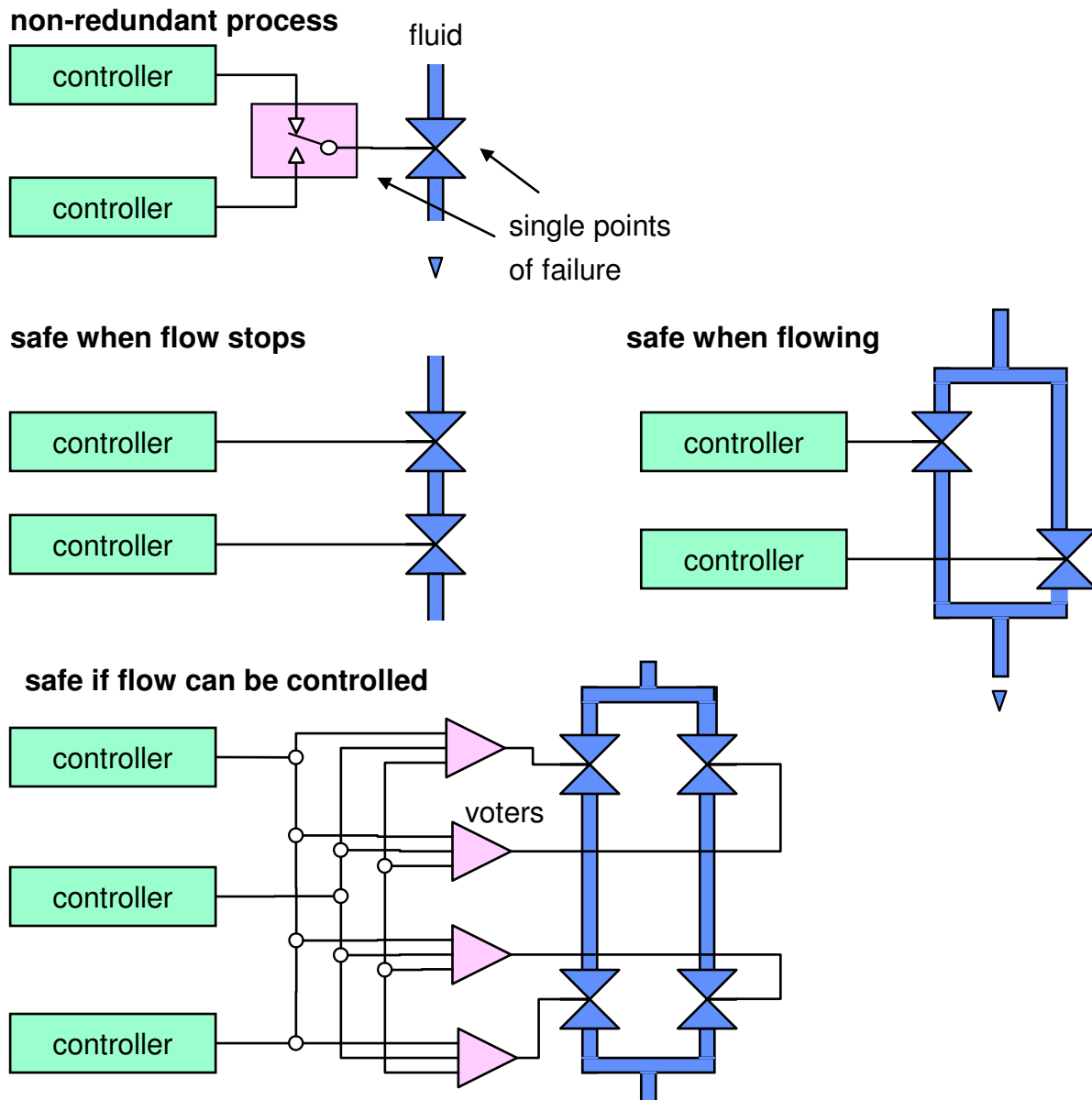


Fig. 1-8: Displacing the Single Point of Failure into the Process.

In the Figure 1-8, the highly reliable case (no fail-safe side) is solved by a series/parallel system of four valves. Any of the valves may fail, either stuck open or stuck closed. Additionally, the system is safe-open for a double fault in which two valves on opposite tubes are stuck open. To achieve fail-closed for stuck-open valves, a by-pass must be added (dashed line).

However, such a quad valve circuit does not protect against malfunctions of the controller, which could open two valves at the same time. A second redundancy of the controller is therefore required. This is depicted here in form of a triplicate system, whose outputs are voted upon by voters. The voters are unreliable elements, but the failure of a voter is identical to that of a valve, so there are no consequences.

Thus, the switching unit may be implemented by using redundancies in the plant, rather than by an actual switch. In any case, the switch being a critical element will require special attention in the design. Replicating portions of the process, or even the whole of it, displace a single point of failure into the process. However, at some point, economic considerations may dictate the presence of a single-threaded component.

1.3.5 Massive and Sparing Redundancy

In **massive redundancy**, (also called "static redundancy"), fault tolerance is achieved by numerous parallel units which perform the same function independently, the process being designed in such a way that it can tolerate the failure of any of these units without taking special actions to remove the faulty unit or include a repair. The redundant units all participate actively in the function; there is no dedicated spare or working unit.

Example:

20 cables support a bridge, but 15 of them are sufficient to bear the load. Upon rupture of one cable, the load redistributes itself among the remaining cables if the anchorage has been properly designed.

On the other hand, one considers **sparing redundancy** (also called "dynamic redundancy"), in which one unit does the work and one or several spares exist, which could perform the same function. Upon fault of the working unit, it is removed from service and a spare is inserted.

Example:

The spare tyre in a car is a sparing redundancy. Its inclusion requires a removal and insertion operation, which takes a certain time. A truck with dual tyres is, by contrast, an example of massive redundancy. It can continue to run upon puncture of one tyre, possibly at reduced speed, until it reaches a repair place.

Sparing redundancy requires fewer resources than massive redundancy and avoids problems due to the parallel activity of several replaceable units. On the other hand, it requires a certain amount of intelligence and switching to recover from a fault, and has problems of its own, which are detailed in the next section.

To reduce the number of spares, one must consider the class of reconfigurable systems for which on-line repair is done by **spare pooling**. In spare pooling, one spare can be shared by several functional groups. This supposes that the spare can adapt to different functionalities:

Example:

A company holds spare personal computers. Whoever needs one can obtain one from the spare pool while his own is being repaired. The number of spares can be calculated knowing the failure rate of a computer, the Mean Time to Repair and number of users, as a function of the required availability and of the probability of spare exhaustion.

1.3.6 Implementing Sparing Redundancy

We detail here some typical problems that arise with sparing redundancy.

An important premise for sparing redundancy is that the spare is in a working condition at the time of switchover. Spares may also fail, even when they are not active. If this situation is not dealt with, the redundancy may be lost, and, what is worse, without notice. Such an undetected fault is termed a **lurking fault**. It is one of the most dangerous situations, since a lurking fault may appear only when the spare is inserted on-line and cause a system failure. Periodical exercise is a traditional method to find lurking errors.

Example:

a lurking fault exists when the spare tyre of a car is flat, but the driver does not notice it. The car has lost its redundancy. For that reason, the driver should check periodically the function of its reserve tyres.

1.3.7 Spare readiness and replacement time

The switchover from the working unit to the spare is not an instantaneous operation. Further, a spare may require some time to be ready to take over the function of the faulty unit.

The switchover can be compared to the removal and insertion of a spare tyre, and the readiness to the necessity to inflate the replacement tyre.

One distinguishes "hot spares" for which the activation of the spare takes a negligible time, "warm spares" which require a certain time to come on-line and "cold spares", which are in a dormant state.

In fault-tolerant computers, spare computing elements exist as redundant hardware. This hardware must be loaded with the correct information in its memory to be ready for service. Three situations may occur:

- **hot spare:** the spare hardware is exactly in the state of the working unit (possibly because it is doing the same calculations at the same time) and it can be inserted just by flipping a switch.

- **warm spare:** the spare is loaded with a valid, but obsolete state and requires a certain time, e.g. a dozen of seconds, to be actualised (brought up-to-date) before inserting. Care must be taken about what happens during the switching time.
- **cold spare:** the hardware is void and must be loaded completely (for instance from a disk) to start up again. Further, the spare must get itself acquainted with the current situation (for instance it may have to ask all the peripherals for their status before starting work). This can take several minutes.

NOTE: the terms "hot redundancy", "warm redundancy" and "cold redundancy" used in reliability theory (Chapter 9) are not related to the above terms. They refer to the failure rate of the spares, (respectively whether they fail at the same rate as the working part or at a lesser rate), and not to the degree of readiness of the spare.

1.3.8 Decomposing a Fault-tolerant System

Modular decomposition applies to the design of complex and large systems. Fault tolerance can also be applied at the element or module level, rather than at the system level. Replaceable units correspond then to modules or groups of modules of the system. The size of a replaceable unit i.e. its **granularity**, is a design option which is strongly influenced by technology.

There are several advantages in dividing a complex system into smaller fault-tolerant units:

- Modular decomposition allows an easier design, and simpler repair and maintenance of available systems. It is also able to distribute better the redundancy, by increasing the reliability of the weaker parts.

Example:

In a computing system, the reliability goal can be achieved by triplication of the power supply, duplicating the disk and with a single CPU, rather than duplicating the whole.

- Crossover becomes possible. The smaller the replaceable units are, the better the theoretical reliability. Indeed, if a large system is replicated, one single fault in the working systems is sufficient to bring that system down and require the activation of the spare, which may fail at the first single fault it experiences. If the large system is divided into numerous duplicated units, then it can suffer the failure of several individual units without failing. This is shown by Figure 1-9:

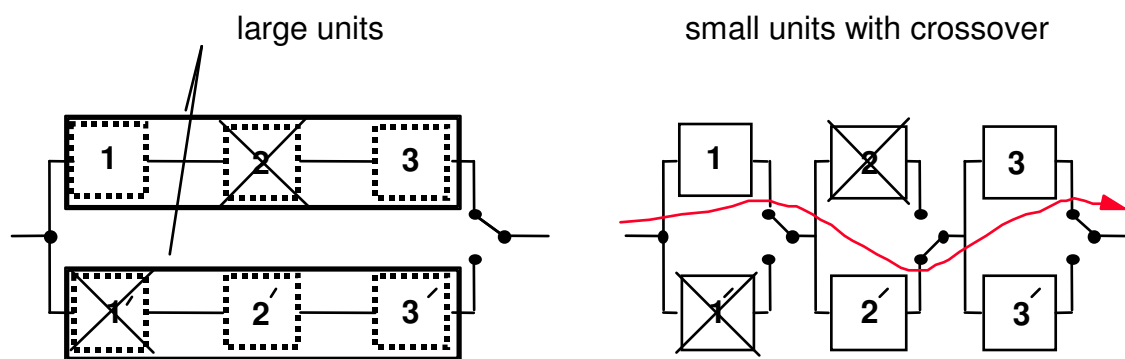


Fig. 1-9: Large and Small Size of the Replaceable Units.

There are also two limits that must be considered, which will be discussed in Chapter 9:

- Small units are located close to one another and therefore, independency of fault is difficult to achieve. If both units are, for instance, located on the same integrated circuit wafer, it is probable that the redundant unit will fail when the working unit fails, for instance because of overheating.
- Insertion of a spare is bound with a certain risk that it fails. When large units are replicated, there is at one extreme only one switching unit, while at the other there is a large number of switching units, exactly one for each RU. The probability that any of these switching units fail is not negligible, even if they are themselves replicated.

1.3.9 Fault Tolerance and Repair

Although we ruled out (manual) repair in the definition of fault-tolerant systems, repair can be combined with fault-tolerance to increase reliability and availability.

Consider first a non-repairable system, as Figure 1-10 shows.

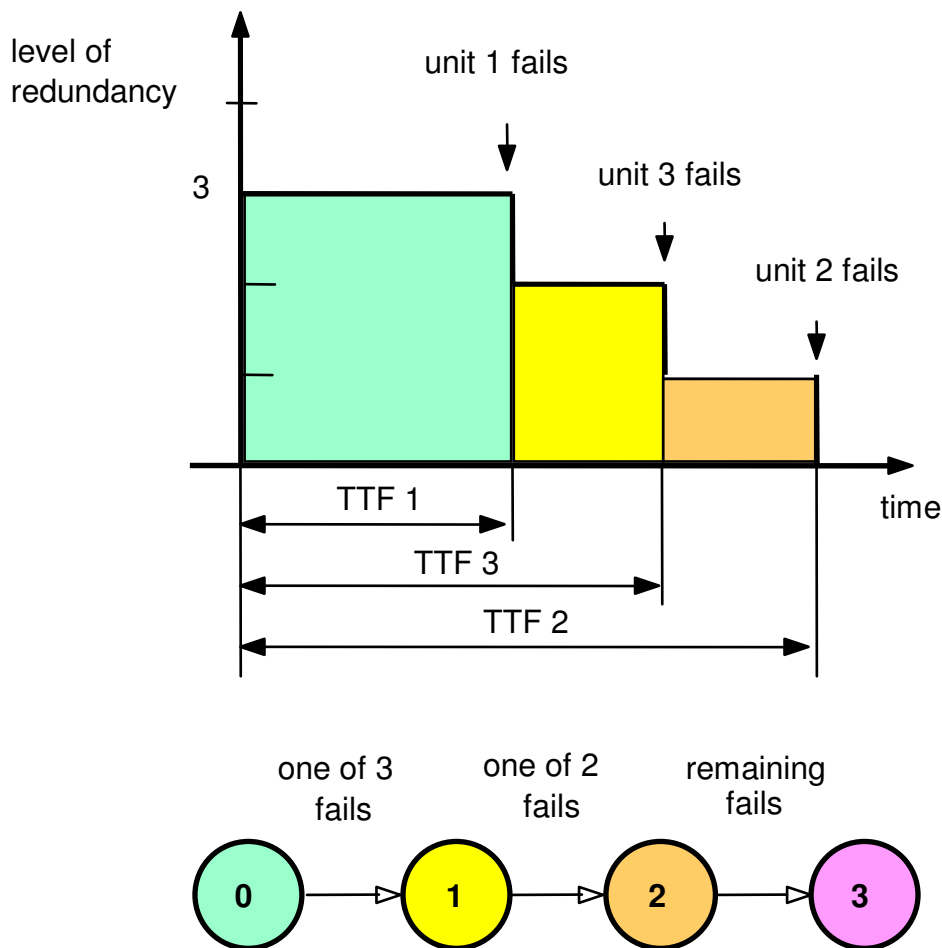


Fig. 1-10: Non-repairable, Redundant System.

This system will fail when all spares are exhausted. As Chapter 9 will show, such a non-repairable system brings a significant increase in reliability for a short mission time only, but on a long mission time it brings no increase in reliability:

Example:

A pilot has an average lifetime of 70 years, and there is a certain probability that he fails on a flight of ten hours. For that eventuality, there is a co-pilot beside him, which has about the same life expectancy as the pilot. Note that the probability that one of them suffers of heart attack on this flight is double now, but it is very unlikely that both the pilot and the co-pilot fail during the same flight, since they are both healthy people, except possibly if they ate the same turtle soup before take-off (common mode failure).

However, it would make little sense to send the same crew in a spaceship on a 100-year trip to Sirius, and expect one of the pilots to be still alive at arrival. Even manning the spaceship with 100 pilots will not help. One can envision a solution by keeping the redundant crew hibernated (by reducing their failure rate). Another solution is reproduction on board, which is a regeneration or repair process, which we will consider next.

The key to long mission times is repair. We must distinguish here between **on-line** and **off-line** repair.

On-line repair depicts the repair of a spare without disruption of the operation.

Off-line repair requires a momentary stop of operation to repair the system.

Thus, a reliable system may be repaired on-line, but not off-line. Fortunately, most reliable systems may be halted from time to time for repair, when the mission involves phases during which the computer is not required. This is then called **scheduled repair**.

Example:

A competition car in a rally over the desert can be considered a reliable system. The car loses the race if a tyre is punctured and a reserve is not immediately available for mounting. The acceptable malfunction duration is only about one minute, sufficient to change a tyre, but not to repair it.

The car carries a certain amount of redundancy: if a tyre bursts, one of the (few) reserve tyres is used. In the non-repairable case, the car fails if all its reserve tyres are exhausted and a tyre bursts.

Now consider that a set of new reserve tyres can be obtained at determined oasis in the trip (service station): the reliability is equal to that of the exhaustion of spares in the interval between oasis. The reliability of the car will depend on the time it takes to go from one station to the other. This illustrates the concept of scheduled repair.

A better reliability can be achieved if the tyres are repairable during the trip, for instance by the crewman while the car is racing. In that case, the reliability is equal to the exhaustion of spares while the others are being repaired. It depends therefore on the mean time to repair and on the mean time to fail of each tyre. This illustrates the concept of on-line repair: the car needs not stop long for repair.

Now, if the failure of the car is not considered dramatic, for instance if the same car is used for a leisure excursion, then one can consider it as an available system. Tyre exhaustion leads then to an off-line repair.

Thus, the distinction between an available system and a reliable one depends on the consequences of the failure of that system, whether one would like to consider it as catastrophic or just a nuisance.

The following Figure 1-11 shows the life of a reliable, on-line repairable system. Whether it is considered as a reliable or available system depends on whether one likes to consider off-line repair (dashed line):

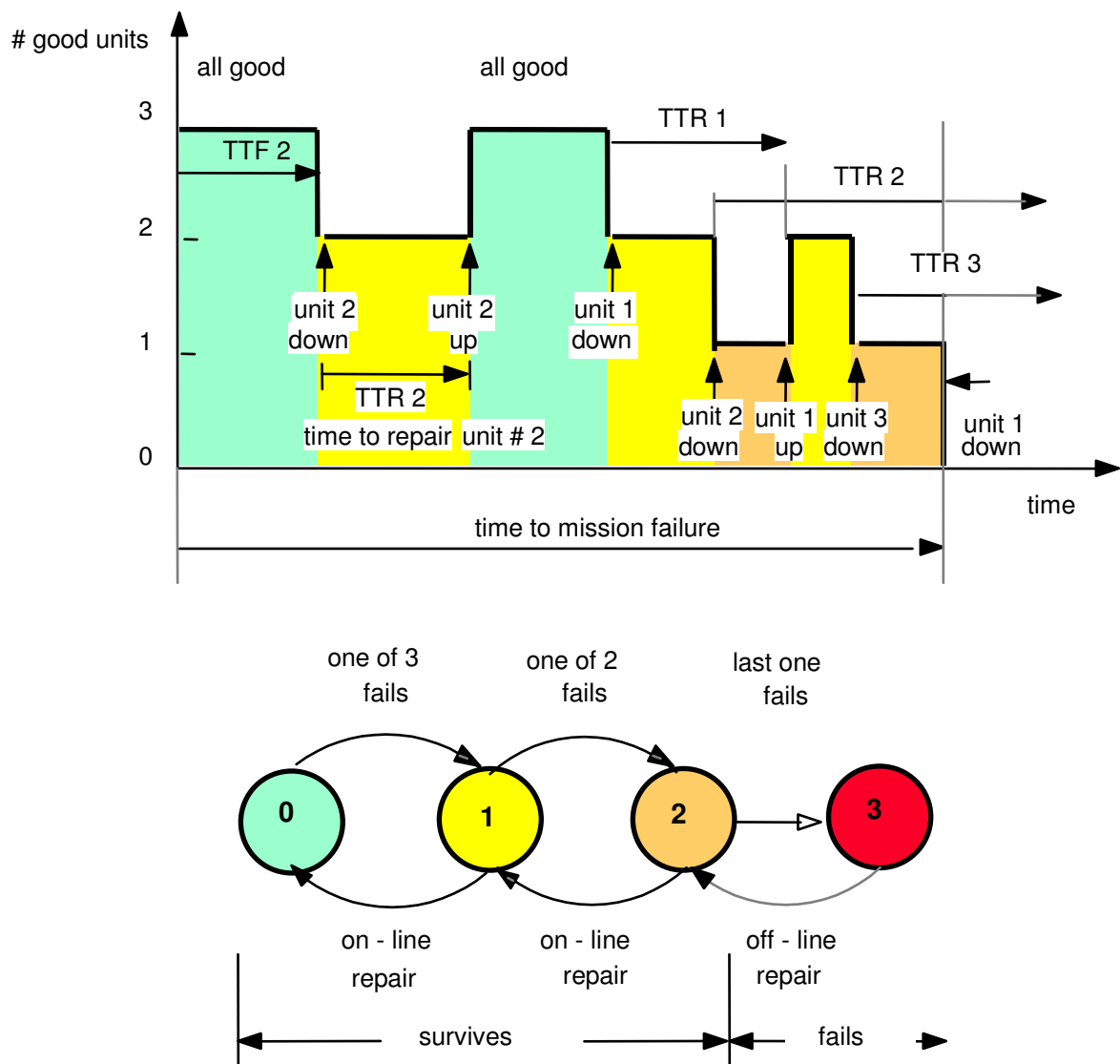


Fig. 1-11: Life of a Triplicated, Fault-tolerant, On-line Repairable System.

The mean time to restore the previous configuration (to reconfigure) is in all cases critical, since a fault during reduced reliability can lead to redundancy exhaustion and to mission failure. Only a reliability study can decide whether the maintenance team should be called at each detection of a persistent error, or only on schedule during the work hours, or regularly every week for instance. Here, the economics of fault-tolerant systems come to play: it may be cheaper to duplicate or triplicate a computer than to maintain a repair team in constant alert 24 hours a day.

Although on-line repair increases reliability and availability, it is less evident that on-line repair can also reduce them. On-line insertion remains a somewhat dangerous operation: automatic reconfiguration can fail because of a second fault, and the presence of a maintenance team significantly reduces the reliability of the working unit(s)

Actual example: the maintenance team comes in and switches off the working machine instead of the faulty machine.

Furthermore, depending on the hardware design, the operation of removing and inserting faulty units may be problematic.

Example:

inserting a repaired board in a working bus (live insertion) can cause currents of several amperes to be switched as the on-board capacitors charge. This current surge will disturb the bus signals and cause transmission errors.

1.3.10 Fault-tolerance and Dependability Definitions

The introduction of fault-tolerance obliges us to adapt the above definitions.

We define the **reliability** of a fault-tolerant system as the probability that the system as a whole has not ceased fulfilling its mission at time $t = T$, (assuming it was in working conditions at the origin time $t = 0$). Reliability is expressed by the **Mean Time To Mission Failure** or MTTMF.

Similarly, we define the **availability** of a fault-tolerant system as the probability that it is fulfilling its mission at time $t = T$, provided it was in working conditions at $t = 0$ (it may have failed several times in between). Availability depends on the **Mean Time Between Mission Failure** MTBMF and the Mean Time to Mission Repair (MTTMR).

The interesting factor is how often components of the system fail, and how frequently the system must be repaired on-line. This factor will dictate the organisation of the repairs and influence the economic factors, and in particular how often the repair team must be called – and paid. We define the **susceptibility** as the probability that one or more components be in a failed state at time t . The susceptibility is expressed by the **mean time between element failures** (MTBEF). The maintainability of the elements is expressed by the **mean time to element repair** (MTTER).

Note that a fault-tolerant system has always a higher susceptibility than a non-redundant one: the higher the amount of redundancy, the higher the probability that a component fails.

The sure thing about a duplicated system is that it will fail twice as often as a non-replicated one.²

Finally, we define the **persistence** of a fault-tolerant system as the probability that it survives a fault in one of its components.

Normally, fault-tolerant systems are 1-persistent, i.e. they can outlive the failure of one of their components, but not necessarily a second failure while the first one is not yet repaired. A multi-persistent system is able to outlive N simultaneous, independent element failures.

1.3.11 Summary

Figure 1-12 summarizes different classes of systems in terms of state diagrams:

² Assuming spare have the same failure rate, i.e. hot redundancy is used.

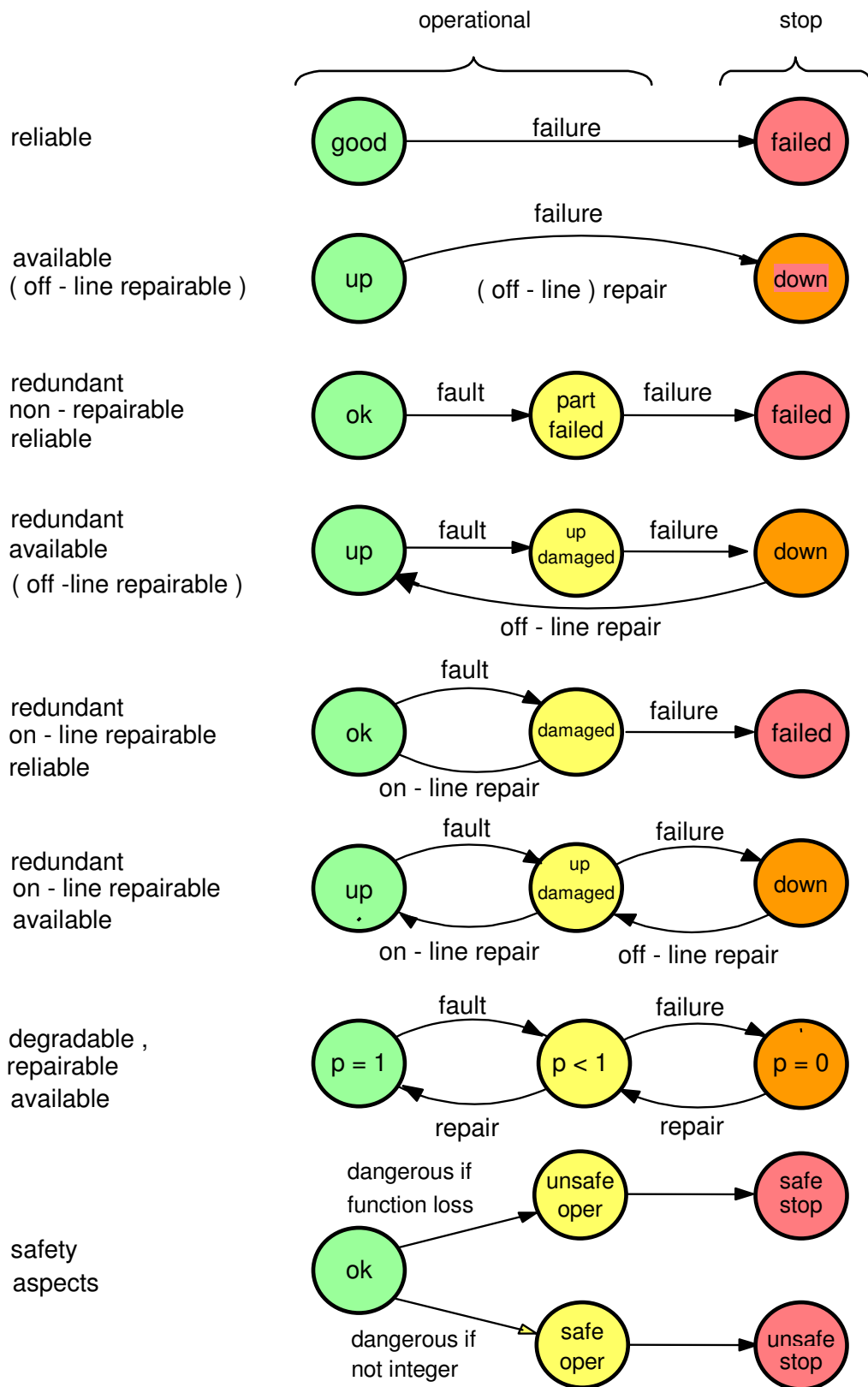


Fig. 1-12: Different Classes of Systems

References

- [Avizienis 78] A. Avizienis,
"Fault Tolerance: the Survival Attribute of Digital Systems",
Proceedings of the IEEE, Vol. 66, No. 10, pp. 1109,
October 1978
- [Laprie 85] Laprie, J.C. ,
"On Computer System Dependability: faults, errors and failures"
Proceedings of the IEEE COMPCON Silver Spring,
1985
- [Randell 78] B. Randell
"Reliability Issues in Computing System Design"
Computing Surveys, Vol.10, No. 2, pp. 123- 165,
June 1978
- [Spectrum 81] Special Issue on Reliability,
IEEE Spectrum, Vol. 18, No 10,
October 1981
- [Siewiorek 82] D. P. Siewiorek & R. S. Schwarz ,
"The Theory and Practice of Reliable System Design" DIGITAL Press,
1982

2 Plant behaviour in presence of computer failure

2.1 Introduction

After the preceding discussion on terminology and operating principles, we will consider here faults and the fault tolerance of the computer from the point of view of the world external to the computer, i.e. from the point of view of the plant. How the plant reacts to computer failures will to a large extent dictate the dependability features of the computer.

A computer is practically always an "embedded system", that is, it is only one part of a larger system, which it controls and monitors up to a certain extent. This system can be an industrial plant, a banking organization, an airplane reservation system or an aircraft. We will use the generic name of "plant" for the part of the system that is not the computer. Therefore, before one defines the specification of a fault-tolerant computer, one should consider the system consisting of a (controlling) computer and a (controlled) plant, and verify the consequences that a malfunction of the computer will have on the plant. The basic question is: "how tolerant is the plant to failures of the computer?" Only then can the question "how tolerant must the computer be to failure of its parts?" be answered.

This Chapter explains that T_{domax} , or "the maximum time a plant can tolerate a loss of its controller without reaching an unrecoverable state" is a key design factor. This factor affects directly the architecture and the mechanisms for standby or spare switching, and thence the computer's architecture.

The second relevant factor is T_{errmax} , or "the maximum time during which a plant tolerates erratic behaviour of the controller without reaching an irrecoverable state". This factor directly affects the choice of the error detection mechanism.

2.2 Controller and plant

We assume that a computer is part of a system, which consists of the computer itself, or "controller" and the external world or "plant". The controller interacts with the plant to control and monitor it. We consider only failures of the controller, not of the plant. We do not consider human operators as part of the controller, but consider them as a part of the plant. The term "plant" is familiar in process control, but it applies also to any other physical process, like a chemical distillation column, an aircraft, a banking corporation or an airline reservation system. Where the controller ends and the real world begins is a matter of definition.

For instance, in an airline reservation system, which consists of several computers, file storage, terminals, etc. arranged in a network, which interacts with the users, the controller can be one specific computer, while the plant consists of the rest of the network and of the users. Alternatively, the whole network can be viewed as a controller and the plant consists entirely of the users of the system.

We will limit the controller to that part of the whole system that we expect to fail, and to which automatic recovery procedures apply in case of failure. As the physical world generally escapes to digital control, the most common assumption is that the plant will be the physical world outside of the computing system. The same view applies to a time-sharing computing centre, a microcomputer controlled appliance like a television or a banking organization.

In summary, we consider as "controller" that part of the system which may fail and to which fault-tolerant procedures apply. We consider as "plant" the rest of the world, which is considered fault-free, but which can be brought into an erroneous or dangerous state by a failure of the controller. Our starting assumption is that the characteristics of the plant will dictate the dependability attributes of the controller, and hence its architecture.

The physical variables of a plant may assume several states, some of which are proper (ensure production as planned), while others are improper. The states can be safe (the plant may remain indefinitely in such a state), others unsafe (permanence in these state will lead to damage). A plant may return from an unsafe state by proper control if its state is still reversible, otherwise it reached an irreversible state. A plant can be in several states with regard to fault tolerance. Figure 2-1 shows a simplified, but typical model:

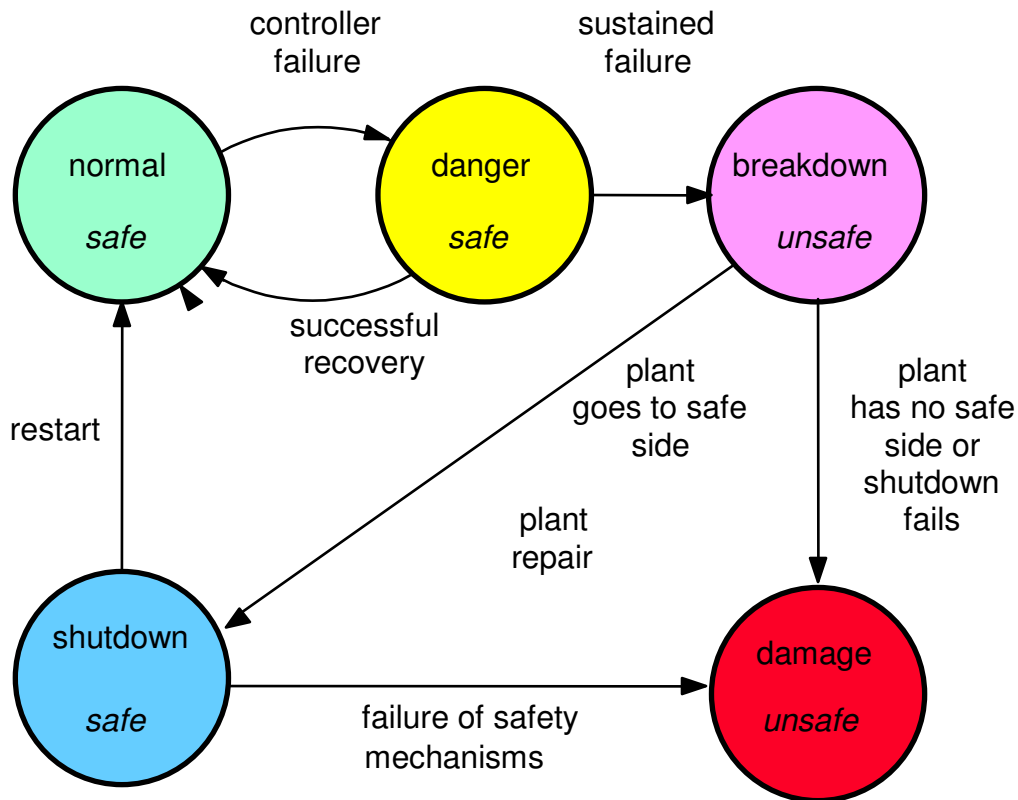


Fig. 2-1. Plant States

Normal: Proper, safe state: this is the normal operation state. All plant physical variables are within the limits of proper operation. This covers the safe operation (this part of the state space is called the safe operation area, or **SOPAR**).

Danger: Improper state, safe, recoverable: some of the physical variables have left the proper state space, but have not yet reached a critical point. The plant can return to the safe area provided an adequate control is undertaken within a limited time.

Breakdown: The plant cannot be brought back to proper operation by automatic action of the controller. This situation will lead either to a safety shutdown, to a human intervention or to damage after a certain time.

Shutdown: Improper state, not recoverable, safe: the plant is brought to a safe, shut-down state (e.g. standstill for a train). The plant cannot return to operation immediately, since the reason for the unsafe operation must be first found and corrected. Restarting of the plant is usually bound with human intervention and safety prerequisites (acknowledgement by a superior authority), which may take time and is subject to economic penalties. Safety-shutdown supposes that the plant has a safe side that can be reached without intervention of the controller (e.g. by an emergency brake).

Damage: Improper state, not recoverable, unsafe: the plant has operated outside of the unsafe area and reached a damage point. The plant cannot be returned to the safe, working state, from the state it has been brought into by a controller failure. It is not any more operational and return to service is either impossible or a costly operation which requires human intervention and repair, as well as a long standstill of the plant. This case is economically the worst.

Of course, at some high level of abstraction, if time and energy do not count, sufficient effort can make all plants reversible. Severe damage can be repaired, even a city reconstructed, but this involves human intervention and will not be considered here.

The above model of a plant's state is coarse. Practical evaluation models work with dozens or hundred of states, depending on the complexity of the installation.

Example:

the behaviour of a simple substation for power distribution requires some 35 states (see Chapter 9)

2.3 Failure mode and plant state

2.3.1 Erratic and fail-stop behaviour of the controller

The basic question is "how long can the plant tolerate the failure of its controller?" The answer is dependent on the one hand on the type of plant, (expressed by its behaviour in case of loss of controller and its transfer characteristics), and on the other hand on the mode of failure of the controller.

Although a computer may fail in a number of ways, we consider only two kinds of failure of the controller that can cause a plant to leave the safe operation area. These are the erratic behaviour and the fail-stop behaviour:

Erratic behaviour: The behaviour of a computer is not predictable in the case of a component failure, due to the large number of elements involved. The computer's output is unpredictable and may consist of wrong data that cannot be recognized as such by the plant. This includes the sending of the correct data, but at an inappropriate time. The erratic behaviour includes malicious malfunction, that is, a malfunction that looks like an intelligent attempt to bring the plant down. One does not know in advance whether the erratic output will have either dangerous effects, safe effects or no effect at all, and worst case should be assumed.

Fail-stop behaviour: This second case is a simplification of the first, in which one assumes that all errors can be detected by adequate means. In the worst case the plant will not receive any inputs from the controller during a certain time.

An erratic behaviour can be turned into a fail-stop one, provided:

- That any error can be detected
- That the erroneous information is prevented from acting on the plant.

In most plants, it is much more likely that damage occurs when the controller is erratic than when it is stopped. Erratic behaviour may bring the plant into a state from which the safe working state can no longer be reached, yet a silence of the controller is more likely to leave the plant in the current state. The reason lies in the fact that most real world processes are not reversible and therefore cannot be easily brought back into a state, once that state has been left. Digital computers and finite state machines in general can be brought into an undefined state by the erratic behaviour of their inputs.

The term fail-stop is preferable to fail-safe, since a fail-safe operation only applies to a plant that has a safe side. A plant has a safe side if the plant remains in a safe state when the controller is silent. Safe-side can also be achieved by technical means such as emergency shutdown which brings the plant to a safe state upon detection of an abnormal situation, by-passing the controller.

There are cases in which erratic behaviour is less dangerous to the plant than a fail-stop behaviour. When the mechanisms used to implement fail-stop imply that the duration of the silence far exceeds the duration of the erratic behaviour, it can jeopardize some kinds of plants, especially those which lack a safe side.

Since the effect of these two kinds of malfunction, erratic and fail-stop can be radically different, they will be considered separately.

2.3.2 Effect of a Fail-stop Controller on the Plant

In case of fail-stop, the controller ceases to generate outputs to the plant during a certain time. Usually, the plant's inputs are designed in such a way that the plant keeps on working with the previous orders. After the disruption, the controller returns to work and tries to bring the plant from its current state back to safe operation.

To analyse this situation, let us assume that the disruption of a controller is bound with costs that increase non-linearly with the duration of the disruption. An empiric plot is shown in Figure 2-2:

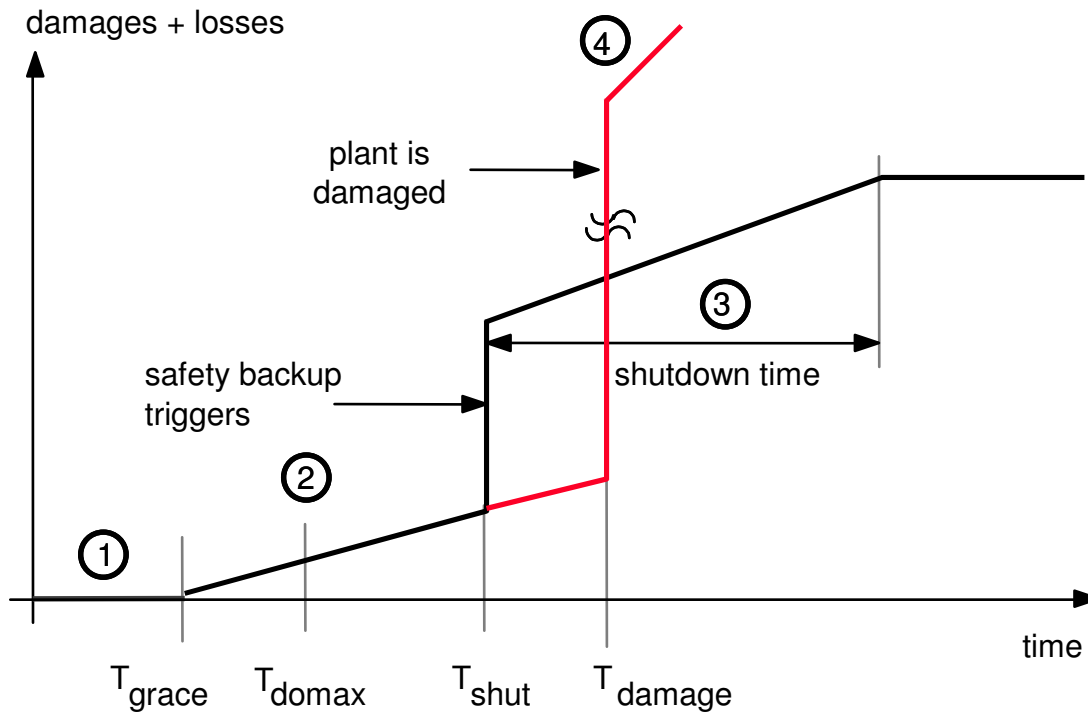


Fig. 2-2: Damage Cost in Function of Malfunction Duration for Different Plants.

This scenario is strongly dependent on the individual plant and Figure 2-2 has only a qualitative value. It represents a probability, since it depends on what happens while the computer is down: if the plant's operation is a continuous process which remains in equilibrium, and no perturbations occur while the computer is down, then the plant will not notice the loss of control at all.

The costs of a failure can be roughly divided into 4 zones: safe zone, danger zone, safe stop and damage zone.

Zone 1: safe zone. The plant does not leave the safe operating area. Practically all plants ignore a silence of the controller that lasts less than a certain "grace time", T_{grace} . An observer in the plant will not even notice the malfunction of the controller if it is short enough, for instance when service is restored before the next command is due or between two samples. This is especially true when the plant has a low-pass behaviour with a time constant which is long with respect to the malfunction duration.

The grace time is a direct function of the main time-constant of the plant. There are no damage costs bound to the grace time.

Zone 2: danger zone. The plant enters the unsafe operation area because it is not controlled for a time larger than T_{grace} . When a plant is stressed outside of the safe operating area, damages begin to show up. The damage cost is usually directly measurable, for instance in term of idle time of employees, or production delay. The cost may also take the indirect form of a risk taken, for instance because of additional stress and wearing of the equipment due to operation outside of the safe area or because of quality degradation of the product.

For most plants, the economic losses in this region are a linear function of the outage. The duration of the linear losses is also a function of the time-constant of the plant.

Zone 3: safe shutdown zone. When a plant is equipped with a safety back-up system, the plant is shut down if it leaves the safe area during a time longer than T_{shut} .

A safety shutdown is bound with a high fixed cost, due to: kicking off users and loosing their work, loosing batches of material, establishing protocols, notifying authorities, reorganizing the production, returning to function, etc.

At the same time, the plant ceases to produce. The standstill costs can climb even steeper than in the former zone, because related plants of the factory may be forced to cease production as a consequence.

Zone 4: damage zone. Damage appears when the plant's safety back-up does not respond, or if the plant has no safe side and depends on the controller's function to remain in the safe operating area. For instance a guided vehicle may be severely damaged if the loss of control exceeds a time T_{damage} . The outage costs soar to account for payment of damages to lives and property, repair of the plant, restoration of production (if it is still possible) and general overheads associated with the accident. However tragic the accident, its total cost will tend to an upper limit.

We therefore define a design factor T_{domax} , which is the maximum time during which a plant can tolerate a loss of the controller function without reaching an irreversible state (either a shutdown or damage). This time will be somewhat greater than the grace time T_{grace} , but smaller than T_{shut} .

2.3.3 Effect of Erratic Behaviour on the Plant

A plant has no means to tell the wrong data from the correct data it receives from the controller. A correctly computed output that is issued at the wrong time is counted as erroneous data. The probability of unrecognised false data is an important parameter of a computing system, and it is known as the RER (residual error rate).

The effect of an erratic behaviour on the plant is threefold:

it can bring a continuous, reversible plant (e.g. a speed governor) into an unsafe state sooner than a fail-stopped controller would by issuing erroneous orders (e.g. an erroneous set-point): T_{grace} is reduced.

it can bring a discrete plant (e.g. a bottle filling plant) into an unsafe state out of which it is difficult to return back to safe operation, even if no damage occurs. Therefore, it inhibits recovery and increases the down time, even after the erratic behaviour has ceased.

Some processes concede no grace time in the presence of erratic behaviour. This is true when the process is not reversible: closing of a high power switch, cutting of a metal sheet, closing a banking transaction. In this case, irreversible damage occurs at the first wrong output: safe behaviour can only be obtained by excluding the erratic behaviour.

We can plot the outage cost in function of the outage time. In general, the curve is similar to Figure 2-2, with the difference that the time scale is shorter and the grace time may be zero.

We define T_{errmax} as the maximum time during which a plant can tolerate an erratic behaviour of its controller without reaching an irreversible state, either shutdown or damaged. T_{errmax} will be greater or equal to T_{domax} . If $T_{errmax} = 0$, then any false output can lead the plant to an irreversible state.

2.4 Failure and recovery

This section discusses a general model of the behaviour of the system consisting of a plant and its controller in the presence of an error.

2.4.1 Fault and Recovery Cycle

The system states are shown in Figure 2-3 as a function of time:

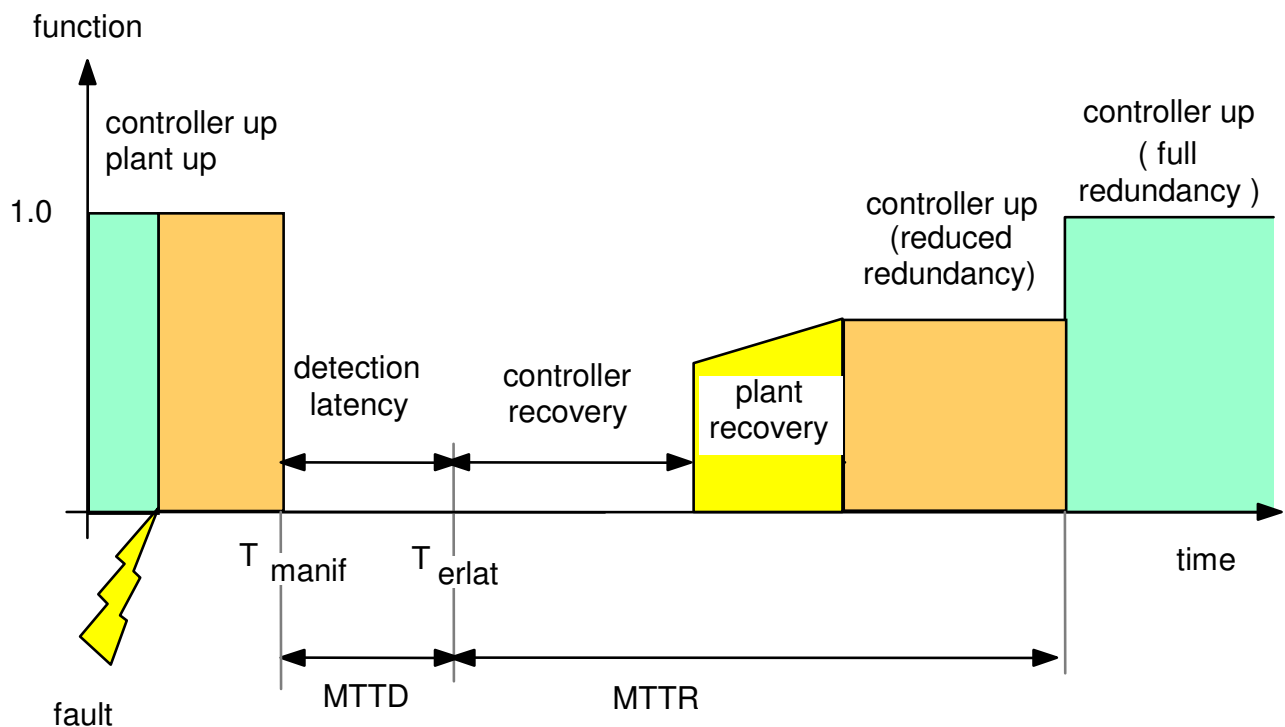


Fig. 2-3: Failure History.

The fault treatment is divided into several phases:

- 1-error manifestation
- 2-error detection
- 3-recovery of the controller
- 4-recovery of the plant
- 5-restoration of redundancy

2.4.2 Error Manifestation

From the moment the failure occurred until it manifests itself, a certain time T_{errman} elapses. "Manifestation" means that after a period of delay, certain conditions cause the error to show up, either as silence or as erratic behaviour. The plant may already be at that time in an unrecoverable state because of a lurking error, especially when the lurking fault affects the redundancy or the error detection mechanism.

2.4.3 Error Detection

After a time T_{errdet} , the error will (hopefully) be detected. Only then can a recovery action be undertaken. It is possible that the controller was giving false outputs since the manifestation of the error until this moment.

The time $T_{errman} + T_{errdet}$ is called the latency time T_{errlat} . It plays an important role since the probability of an undetected error dictates the reliability of the whole controller in a fault tolerant system.

The latency time is often expressed by the mean time to detect an error (sometimes incorrectly termed as "coverage").

2.4.4 Recovery of the Controller

The recovery of the controller will be the topic of the next Chapters. For now, it is sufficient to define the following controller classes:

- **masking**: the controller masks all its internal failures. The above discussion is then irrelevant since there is no failure of the controller that would be visible to the plant.
- **fail-stop**: the controller ceases function when it fails and no erroneous data reach the plant. Return to service requires human intervention. Fail-stop generally calls for duplication and comparison, with the output being stopped when there is disagreement.
- **recoverable**: recovery consists of two phases: the detection phase during which the error is detected, and the switchover phase during which recovery is attempted.

The detection phase takes a time T_{errlat} during which erroneous data can leak out to the plant. For instance, if error detection is done by a test that runs every 20 ms, then T_{errlat} can amount to 20 ms, assuming an error is detected at 100 %. If the controller is fail-stop, then T_{errlat} is essentially zero. This is the ideal case. The switchover phase which follows takes a non-negligible time T_{switch} during which the controller is not operational (silent). This phase can take microseconds for hot stand-by or instructions retry, milliseconds for warm stand-by, or dozens of seconds for cold stand-by.

- **interaction recoverable**: although the controller is not operational during recovery, it is capable of monitoring its inputs and outputs. It will not ask again for inputs already received and will not output commands already sent. Such behaviour requires warm stand-by redundancy. Obviously, $T_{errlat} + T_{switch}$ must be $< T_{domax}$ for successful recovery.

2.4.5 Recovery of the Plant

Now, once the faulty situation has been overcome, the controller is again operational. The question is: "what action must the controller undertake to correct its past faulty behaviour?" The response depends on what it did to the plant (or did not) during the outage and especially on the kind of plant.

The basic requirement for plant recovery is that the plant is reversible, that is, that it can be brought back into the safe state space by proper controller action. This is not always feasible, since time cannot be rolled back. On the other hand, if one can afford any cost for reversibility, then every plant can be made reversible. We will assume that a plant is reversible as long as no shutdown and no damage took place.

For recovery, the most important factor is whether the plant is **monotonic** or **not monotonic**.

Most plants in the analog control world, e.g. speed control, voltage control, are monotonic. The output of the plant can be brought back into the initial state by applying a signal in the reverse direction of the faulty signal. Monotonic plants only have a short-term memory.

If the plant is simply proportional, then an easy way to correct errors exists: it is sufficient to output the correct value again.

An example of a proportional, monotonic plant is a furnace. Its temperature tends to be proportional to its input, that is, to the fuel throughput. If a wrong fuel set point is issued, then the controller can correct the situation by issuing the correct fuel set point. The furnace will, after a while, not "remember" the wrong input.

When the plant exhibits integral behaviour, or when there exist large perturbations, then the method of correction for a monotonic plant is well known from the process control field. The regulation loop is probably the oldest known error recovery mechanism, and it can also recover from controller failures. It relies on a comparison between the measured output and the desired output, and the corrective action is straightforward.

Most digital systems and most technical processes are not monotonic. This is especially true with computer systems. The plant has discrete states, and once a plant passed into an incorrect state, one cannot normally return to the original state simply by applying the reverse order. For instance, once a "delete" command has been given, there are few operating systems that would allow an "undelete" to recover a file that was unduly deleted.

The non-monotonic plants possess a long-term **memory**, while monotonic plants only have a short-term memory and are capable of **forgetting**. Error correction in a non-monotonic plant requires a more complicated recovery mechanism, but the recovery method is basically always the same:

The state of the plant is divided into 2 parts, the suspect part, which is suspected to contain erroneous states and the trusted part, which is considered proper.

For instance, if the plant is a computing centre, the memory can be considered suspect, while the disks may be considered sound.

The suspect part is corrected by placing it into a defined state. If this state is one that has existed before, one speaks of **retry**. Restarting the work from that state will yield the state the plant would have reached if the controller had not failed, or at least yield a proper state that corresponds to it.

The suspect part may also be replaced by a state that never existed before, to bring the plant directly into a proper state. This is the **compensation** method. Compensation assumes that the extent of damage is known and requires an intelligent evaluation of the erroneous state.

Both compensation and retry are used in everyday life to recover from errors. Compensation corresponds to a "storno" in bookkeeping: erased in the document is not allowed, but the faulty entry must be compensated for in the opposite column. Retry correspond to throwing away the page and writing a correct one.

Compensation is of course faster, but requires some intelligence to tell what is erroneous and correct it without introducing a new error. Retry requires more redundant information and extra time to redo the calculations.

These methods apply quite well to the recovery of a digital computer, because digital computers are reversible machines, but less well to the physical world that are most of the time irreversible. We have seen that an erroneous output can lead the plant into an irreversible state (shutdown or breakdown), which is of course unrecoverable.

Retry for instance would require the restoring of the plant in a previous state. Sometimes it is easy, but when the operation of the plant consists of parallel sequences of operations, it is practically impossible.

Consider the problem of restoring a car production line in its former state: the conveyors should be rolled back, already inserted screws removed, painting stripped off, etc. Compensation is in this case the only correct solution, but it requires a great deal of intelligence to define a new starting point for production: one must remove the faulty parts, reinitialise the part counters, etc.

When dealing with hardware errors, it is much more difficult to determine the extent of the damage. There is no known procedure that allows one to determine how much of the state has been contaminated by erroneous hardware. One tries during the design phase to at least reduce the amount of interaction between the parts by

establishing firewalls, either "horizontally", by separating physically the parts (e.g. by a network), or "vertically" by layering the structure and implementing error recovery at each layer.

Therefore, considerable effort is made so that no erroneous output ever reaches a plant. This is the main objective of the safety signalling used in the railways. The general solution is to make the controller fail-stop.

On the other hand, it is too easy to state that "no erroneous output should ever be sent to an irreversible plant". There are indeed situations in which an operation requires subdivision into several steps, each of which must be completed before the next can be done, and which additionally requires that all steps are done or none of them. This is the concept of **atomic actions**.

We will come back to this topic when considering recovery of the environment (Chapter 6) and within a data base (Chapter 7).

2.5 Plant classification

2.5.1 Properties

A plant's state may be:

- **Reversible or irreversible:** reversible in the sense that the system by itself can return to operation without human intervention. Most processes are physically irreversible. We consider a plant that has been shut-down to be in an irreversible state.
- **Recoverable or unrecoverable:** there exist means by which the system can be brought back to operation once erroneous orders have been given. This must be done by proper control within a limited time. If the plant is not recoverable then there should be no false outputs ever issued.
- **Repeatable or non-repeatable:** a non-monotonic plant whose failed operation is repeatable can be recovered by replacing it in a previous state and resuming the computation starting from that state. If the operation is non-repeatable, a positive action must be undertaken to bring the plant directly from the improper state into a proper state.
- **Monotonic or non-monotonic:** the recovery strategy for a monotonic plant is achieved by placing controller and plant in a regulation loop. For a non-monotonic plant, a correction action is involved.
- **Proportional or integral:** a proportional plant can be corrected just by issuing the correct data again as it has only a short-term memory. An integral plant requires correction.

2.5.2 Plant Parameters

On the base of the above considerations, we divide the plants in several categories. We assume that if a plant cannot be brought back into proper operation without human intervention, it is in an irreversible state. This state includes the shut-down and the damage states. We do not consider failures of the plant itself.

The plants are characterized by four criteria:

1. the **mean down time** over the mission time, expressed by the (un)availability, which is the sum of the down times caused by failures of the controller. The MDT includes both successful and unsuccessful reconfiguration times. The latter applies only to plants with a safe side or reversible plants, since in case of unsuccessful reconfiguration, it is assumed that a plant without safe side is lost. The MDT does not include the down time due to the failures of the plant itself.
2. the **maximum uncontrolled time** T_{domax} , which is the maximum controller down time the plant can tolerate a malfunction of the controller without irreversible effects. It defines the kind of reconfiguration to be used to bring the plant back into operation before an irreversible state is reached: (massive, masking redundancy, hot standby, warm standby,...).
3. the (**recoverable**/not recoverable) attribute defines whether the plant tolerates erroneous commands or not. If the plant is not recoverable, then an integer behaviour of the controller is required.
4. the (**safe side** / no safe side) attribute specifies whether a fail-safe shutdown can be executed in case an unrecoverable error is detected. If a plant has a safe side, then a Mean Shutdown Time must be indicated, and a fail-stop behaviour of the controller may be sufficient.

2.5.3 Classes of Plants

2.5.3.1 Benign plants

(Availability < 1 , $T_{\text{domax}} \gg 0$, Recoverable, Safe side)

The damage is a direct function of the outage (outage costs are linear)
There is no shut-down state and the plant cannot be damaged by action of the controller.

Example : a household appliance

2.5.3.2 Irreversible plants

(Availability < 1 , $T_{\text{domax}} > 0$, Not Recoverable, Safe side)

These plants suffer irreversible damage from an erratic behaviour of the controller, but can remain a relatively long time in the shut-down state.

Example: a banking system, teller machine.

2.5.3.3 Protected Plants

(Availability < 1 , $T_{\text{domax}} > 0$, Recoverable, Safe side)

Operation outside of the safe area is detected by an equipment independent of the controller. This device triggers a safety mechanism that brings the plant into a safe state. The erratic behaviour of the controller is acceptable until the safety mechanism is triggered. This is possible because the plant and the controller can be placed within the same regulation loop. The triggering of the safety mechanism is associated with rising operating costs.

Example: a railway with emergency brake and overspeed control.

2.5.3.4 Unstable plants

(Availability ~ 1 , $T_{\text{domax}} \ll$, Recoverable, No Safe side)

These plants do not have a safe side and rely entirely on the controller to remain in the safe operating region.

Example: airplanes in flight, some chemical processes.

2.5.3.5 Critical plants

(Availability $= 1$, $T_{\text{domax}} \ll$, Not Recoverable, No Safe side)

These plants do not allow any false input. Any outage of service larger than t_{domax} leads to damage.

Example: air traffic control, airplanes during landing.

Whether the outage was due to erratic or to fail-stop behaviour of the controller is not relevant, as long as the outage duration includes the return of the plant to safe operation.

2.6 Implication of the plant parameters on the controller structure

We have seen that a plant is characterized by 4 factors:

1. how often downtime is permitted (given by the element reliability)
2. how long an outage may last without shutdown (given by the grace time),
3. whether the plant is irreversible (i.e. requires controller integrity)
4. whether the plant has a safe side (i.e. allows fail-stop)

The controller of the plant is characterized by 5 factors:

1. the Mean Time To System Failure, which is the probability that redundancy within the controller is exhausted or recovery fails.

2. the **mean time between element failure** (of one of the redundant units). The MTBEF tells how often a failure/reconfiguration can be expected.
3. the Mean Recovery Time which is the expected time the controller needs to recover from a failure. For success, this time must be smaller than the grace time.
4. the Coverage, which is the probability of success of a recovery.
5. the Mean Time To Repair of a damaged part. Together with the MTTEF, the MTTER determines the probability of a second failure during the repair time and therefore dictates the amount of redundancy required.

Although values for these parameters are rather difficult to obtain or to measure, they present more meaningful starting points for the design of fault tolerant systems than the well-known notions of MTBF or availability.

An availability of 99.9 % (better: an unavailability of 1/100) means that the plant may be down about 9 hours a year, or 90 seconds every day. But the damage may be intolerable in the first and acceptable in the second case. Therefore, the specification must include a value for the maximum permitted outage.

2.7 Typical plants

The following is a summary of the expected dependability goals for different kinds of plants.

2.7.1 Telephone exchange

The main goal of telecommunication systems is availability. Occasionally dropped communications do no harm, the user redials by himself if the line is lost. Misconnections and transmission noise are more problematic. The system must be back in operation as soon as possible.

Availability: $A = 99.994 \% (T_{\text{down}} < 3 \text{ minutes/year})$ (taken from Ericsson's AXE)

"sustain 99.9875% of all established calls (no more than 1.25 out of 10000 are cut off). (AT&T's 3B20B)

"occasionally dropped messages and brief outages are acceptable; outages of more than a few minutes are undesirable, even if scheduled in advance. (BBN's Pluribus)

Integrity: no more than 1 false connection in about 10 000.
(accounting requires however full integrity)

Grace time: < 3 minutes

Solution: dual, hot-sparing computers

2.7.2 Power Plant and Telecontrol

Computers in power plants perform SCADA functions (display of process data and forwarding of operator commands). A failure of the computer renders the operator blind and can even lead to the issuing of false commands. Power circuits are generally protected by independent safety equipments, so that false commands can be caught up, but nevertheless represent a risk factor.

Power plants are therefore required to have a very high availability. Outages can be tolerated and plant failures shielded from the user by the interconnection grid. Outages are bound with heavy financial losses, so that a total power availability is specified by the utility.

Availability: $A = 99.972 \% (12 \text{ minutes per month})$

Integrity: very high (no falsification of commands)

Grace time: < 60 s

Solution: duplicated, hot- or warm-sparing computers.

2.7.3 Chemical Plants

In distillation columns and cracking equipment, the quality of the produced products degrades rapidly with the outage time. Eventually plant shut-down may be required since the products cannot be used any more. Availability

should be high. A high reliability is seldom required. It is requested mostly for processes which have no safe side like explosives and toxic products mixing.

An important requirement is integrity: computer malfunction should not have an effect on the process, and especially should not lead to opening of valves or product mixing.

Reliability: medium

Integrity: very high

Availability: high

Grace time: depends on the process, less than 2 minutes in some cases

Solution: duplicated computers with self-check for integrity and availability.
Triplification and voting for safety applications.

2.7.4 Cars and Household Appliances

Stringent cost factors rule out component redundancy. The cost of redundancy must be traded off against failure during warranty time. In cars, electronic injection and exhaust regulations have introduced microcomputers on board. Until now, there are no fault-tolerant computers for these applications. One notable exception is the anti-skid used in brakes. Here, complete decoupling between the electronic parts is required.

Availability: similar to that of the motor

Grace time: < 2 s

Reliability: no more than 1 failure/10 years

Integrity: not asked for, except in braking system.

Solution: fault avoidance, duplication and massive redundancy for brakes.

2.7.5 Rail Vehicles

On rail-guided vehicles such as trains or subways, electronics have been used for a long while. Recently, on-board computers have been used for automatic subways such as San Francisco's BART as a step toward unmanned vehicles, although this solution is not yet generally accepted. The first goal is fail-stop behaviour, since rail vehicles have a safe side (except possibly when performing an emergency braking on a bridge). Pure fail-stop impairs availability. If the failure rate of the on-board computer and of its checking logic would be 10-4/h, then about three trains a day would experience an emergency stop on the swiss railway network only (one of the densest of the world, however). Although reliability makes great progresses through proper design, continuous service is increasingly asked for.

Availability: similar to that of the current hardware

Grace time: < 10 s

Reliability: no more than 1 failure/10 years/vehicle

Integrity: very high, fail-stop required for safety features

Solution: TMR computers (Brown Boveri's LZB), fail-stop logic [Strelow 78]

2.7.6 Electronically Guided Vehicles

Non-track guided vehicles are investigated as urban transporters. The vehicle is guided by a wire in the ground or similar means. The same considerations apply to drive-by-wire cars, where the steering wheel commands the wheels by computer. No failure of the guiding computer and associated sensors can be tolerated, any error must be detected. Fail-stop alone is not acceptable since a guiding error could lead the vehicle off-track (at 36 km/h, the vehicle could be off-track by 10 m in 1 s [Schmidt 84]. Availability requirements are identical to those of railroads.

Availability: similar to that of the motor

Grace time: < 1 s

Reliability: 10-8

Integrity: availability more important (fail-stop can be dangerous)
Solution: triplicated guidance system with analog back-up (diverse redundancy)

2.7.7 Bank Teller

A bank teller has no stringent reliability requirements. The availability must be high so as not to discourage customers. The primordial requirement is integrity: no error of the computer should have any effect on the account balances. Most operations are transaction-oriented (consist of an indivisible sequence of operations). Atomicity of transactions must be preserved. Protection against voluntary malfunction ranks high, since potential benefit

s for intruders are large.

Availability: 1 failure/ 18 months

Integrity: very high

Solution: fail-stop computer and short mean time to repair (warm stand-by)

Some of the architectures used in transaction processing will be described in Chapter 7.

2.7.8 Aerospace

The reliability of the computer must be very high when the plane depends on the computer's function to fly. The requirements set up by the NASA are that the computer reliability must be equal to the probability of not losing a wing, which has arbitrarily been set at 10^{-10} /h. (This is approximately one case in 1000 years assuming that 10'000 planes are aloft at one instant). To fulfil these requirements, the NASA contracted SRI International and Charles Draper Labs to design a highly reliable fault tolerant computer (which were respectively named SIFT and FTMP [Hopkins 78, Wensley 78]). General Electric [Schmid 84] and Bendix have since developed highly reliable computers for this purpose. Ground following computers for combat aircrafts such as the Tornado have similar reliability goals.

The Space Shuttle [Sklaroff 76, ACM 84] is also a fly-by-wire vehicle. It relies entirely on the (quintupled) computer to fly. In the most critical missions phases (about 60 s after launch and in the landing phase), only a few milliseconds of outage would be tolerated. All five computers perform the same calculations at the same time, the result of four of them are voted upon, the fifth computer is of different design and serves as back-up in case a generic software fault would affect the other four. On orbit, reliance is lower: the synchronous mode is abandoned and the computers are dedicated to separate tasks, some can be powered off to lessen consumption and increase reliability.

The reliability requirements are less strict for computers that perform common chores, navigation or flight optimisation. Here, availability is asked for, i.e. the largest possible percentage of operating time. There is one exception however: The computing equipment used during the last minute before landing is safety-decisive and therefore should have a very high reliability. This equipment is generally triplicated like in the Airbus or Boeing 767. The mission time is about 10 hours for an aircraft, but can amount to several weeks for the Space Shuttle. Therefore, the Space Shuttle requires more redundancy since repair on orbit is not foreseen. Some of these architectures will be described in Chapter 4.

Availability: about same as engines

Reliability: 10⁻¹⁰

Grace time: some ms.

Solution: triplication and voting, spare pooling.

2.7.9 Space Probes

A typical space probe like Voyager has a mission time of several years. No repair is possible. The spacecraft depends on its computer to perform its mission and therefore it requires a highly reliable computer. The reliability goal is achieved by a high degree of fault avoidance: only proven components and design techniques are used, which let some design features look obsolete. The components are carefully screened and tested before and after insertion. The modules are individually tested and their interfaces verified. The fault-tolerance technique uses duplication of the computing elements with self-check. Error confinement is achieved by hardware and software methods [Jones 79, Rennels 78].

Special precautions must be taken for some functions in the spacecraft like firing of thrusters and in general for consumption of expendable resources.

Availability: depends on the mission phase. Malfunction tolerable during ride, but not at fly-by.

Reliability: 95% success probability for a mission of 5 years

Grace time: depends on the function and the mission phase

Solution: reconfiguration techniques around a hard-core, fault-tolerant computer.

2.8 References

- [ACM 84] Communications of the ACM,
Special Issue on Computing in Space, September 1984
- [Jones 79] Jones, C.P.,
"Automatic Fault Protection in the Voyager Spacecraft",
Amarin Institute of Aeronautics and Astronautics AIAA Paper No. 79-1919
- [Schmid 84] H. Schmid, J. Lam, R. Naro, K. Weir,
"Critical Issues in the Design of a Reconfigurable Control Computer",
14th Int. Symposium on Fault-Tolerant Computing, FTCS-14 Orlando, Florida, June 1984
- [Schmidt 84] A. Schmidt, R. Faller & W. Poettig,
"Elektronisch gelenkter Bus"
ELEKTRONIK, 15, pp 41-46, July 1984
- [Sklaroff 76] J.R. Sklaroff,
"Redundancy Management Technique for Space Shuttle Computers",
IBM J. Res. Develop. Vol. 20, No. 1, pp 20-28, January 1976
- [Strelow 78] H. Strelow, H. Uebel,
"Das Sichere Mikrocomputersystem SIMIS",
SIGNAL+DRAHT Vol. 70, No. 4, pp. 82-86, January 1978

3 Error detection and correction

In this Chapter, we consider in detail the failure modes of a computer, means to detect errors and the basic techniques for fault-tolerance.

3.1 Errors and faults in a computing system

3.1.1 Model of a Computing System

We will consider here only a simple model of a computer, consisting of a processor interconnected with a memory and peripherals I/O over a parallel bus. This computer may communicate with other computers and with further peripherals over a network (Figure 3-1).

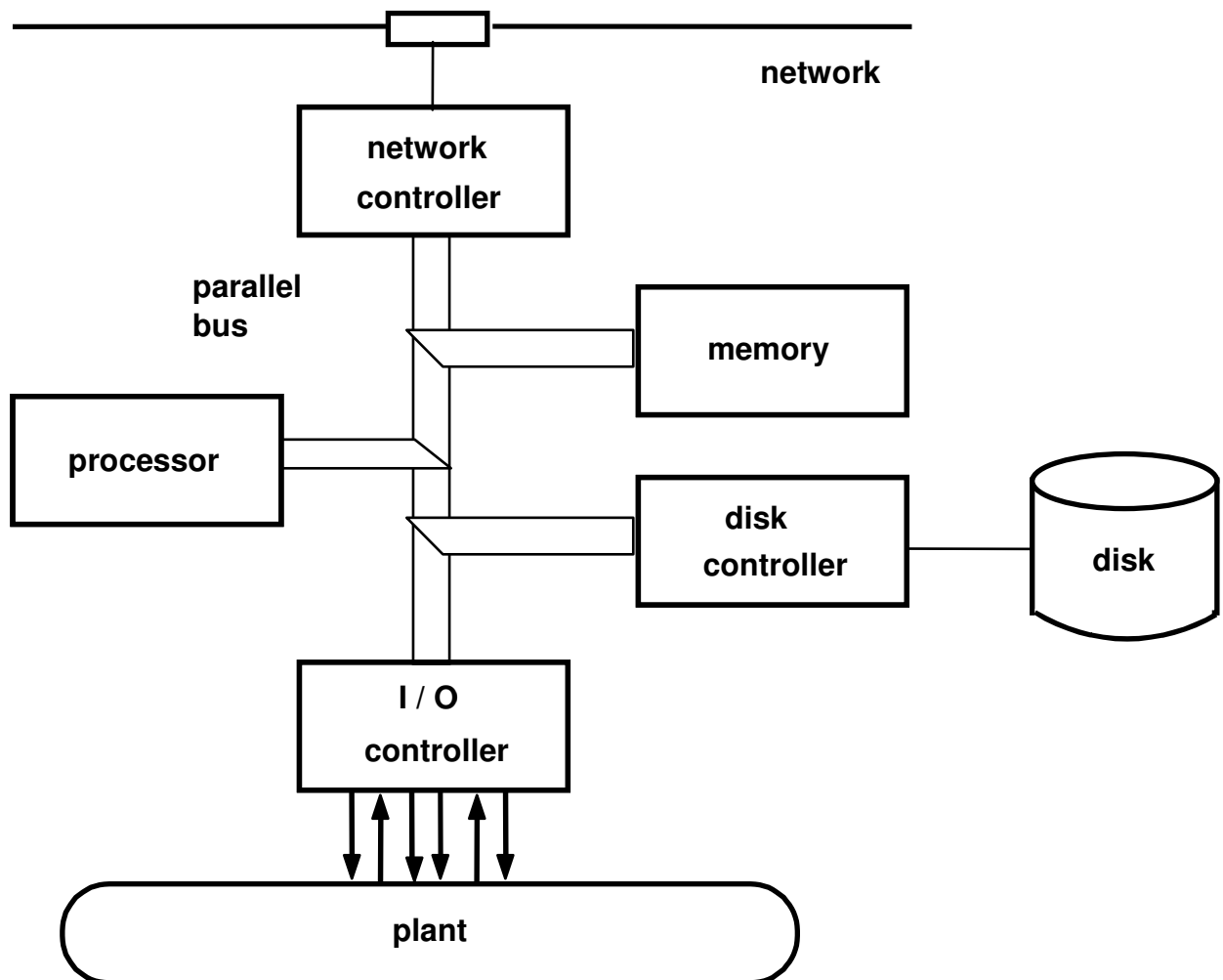


Fig. 3-1: Computing System.

The techniques for fault-tolerance distinguish five kinds of elements:

- **Busses and transmission links** are basically memoryless. Errors can be corrected by retransmission.
- **Memories** (RAM, disks) have an internal state and can memorize errors. Errors can be corrected by overwriting.
- **Processors**, which are a complex mixture of combinatorial and sequential logic. Although theoretically a computer can be clearly divided into a combinatorial part (arithmetic unit) and a sequential part (registers), actual components must be treated as a whole. Errors must be corrected by resetting the processor and bringing it to the desired state by letting it execute a start-up program.

- **External world**, which consists of elements which can be memoryless, assignment correctable, reversible or not correctable (see Chapter 2). Error correction may require a complex set of operations, if it is at all possible.
- **Auxiliaries** such as power supply, cooling and mechanical protection. The correct design must include protection against:
 - temperature increase,
 - power supply variations,
 - electro-magnetic interferences,
 - door locking,
 - operator mishaps, etc..

The protection against external threats can often increase reliability more than straightforward application of fault-tolerance techniques. But since these techniques are very much installation-dependent, they will not be contemplated here.

3.1.2 Failure Modes

A computing system, as a complex machine, has usually only one correct operation and an infinity of possible incorrect operations. These include pathological cases in which the computer acts in a "malicious" way, defeating the measures taken by its designers against its possible failure. We have distinguished two general behaviours which a failing computer exhibits:

erratic behaviour: providing in addition to the required service another service (**integrity breach**);

fail-stop behaviour: not providing the required service during a certain time (**functionality breach**).

3.1.3 Fault Origin

Errors are incorrect data items, which cause the computer's state to deviate from the state intended by its designer or customer.

Errors in digital systems are the consequences of both PHYSICAL FAULTS and DESIGN FAULTS, the first being of physical nature and the second of human origin.

- Physical, or hardware faults concern malfunctions of elements due to aging, temperature, or other external causes.
- Design, or software faults concern specification, software, and hardware design faults, which are consequences of errors of the designer (whether it is a human or a machine).

There is a clear distinction between design and physical faults:

A corrected design fault will not show up again, while a corrected physical fault may occur again at any time.

Accordingly, both fault modes should be treated differently. This clear theoretical distinction gets rather blurred in practice. A physical fault may occur because of a design fault.

For instance, a circuit may fail repeatedly because of a poor circuit design, a missing cooling fan or out-of-date manufacturing instructions.

Conversely, a design fault may behave like a physical fault: as software complexity increases, so does the number of hidden bugs in it. It is not possible to guarantee that it is fault-free. When the complexity of a piece of software becomes high, one could also indicate a "Mean Time To Failure" for that software module.

So, design and physical faults should be treated as conceptual helps. Figure 3-2 (taken from [Toy 78]) shows a statistic of the origin of failures on the ESS processors used in telephony:

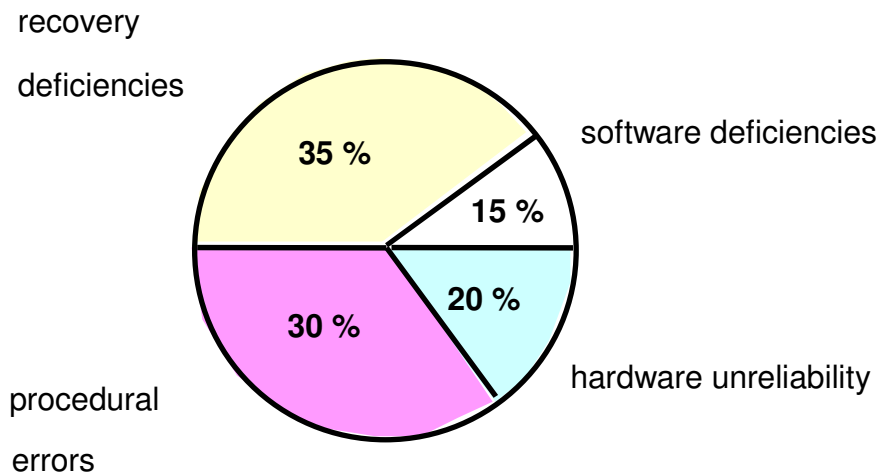


Fig. 3-2. ESS Telephone Exchange - Down-Time Allocation [Toy 78]

This diagram shows that only 20% of the errors can be attributed to physical damages, and only 15 % clearly to software bugs. The rest is caused by failure of the recovery process (insertion of redundancy) or is of unknown origin (possibly secondary faults during the recovery process).

Faults due to human intervention are a special case. Some consider them as a design fault. Others consider them as the consequence of an improper design, since their origin is external to the machine: it seldom occurs that the operator of the machine also wrote the programs. Human faults can be avoided by making the handling of the machine "fool-proof". Therefore, human faults fall into the same category as improper shielding for the hardware.

One should therefore distinguish between logical design faults and protection design faults. A logical design fault, such as a programming mistake, will lead to an error even if the hardware is perfect. A protection design fault is an insufficient protection against external threats, including arbitrary human interaction.

3.1.4 Fault Duration

Faults may be persistent (definitive) or transient (volatile).

- **Persistent** faults relate to physical damage, for instance burned transistors. They are due to external interference, but can also be caused by the aging of the components. A permanent fault remains until repaired.
- **Transient** faults are caused mostly by external disturbances, such as electromagnetic interferences, mechanical vibrations, ray flashes or decaying radioactive elements. A transient fault may cause a permanent error if it is memorized. Transient faults depend on the environment of the equipment. For computers in an aircraft, transient faults are about 100 times more frequent than persistent ones. This number drops to about 10 for fixed location computers. Furthermore, transient faults due to external interferences tend to occur in bursts of limited duration. In any case, the methods used to deal with transient faults are quite distinct from those used to fight permanent errors.

We avoid using the terms of "**hard**" error for a permanent error and "**soft**" error for a volatile one, because of the confusion with hardware and software faults, although these terms are often used to describe memory faults.

3.1.5 Fault Latency

When a system depends on all its elements for its mission, then a fault of any of its elements is equivalent to the failure of the mission. If not all the elements are stressed at the same time, then a fault of an element does not necessarily cause a failure of the system, at least not immediately. The time that elapses between the occurrence of a fault and its manifestation is called **fault latency**.

A fault of an element of a computing system does not lead immediately to a failure of the system, because not all parts of a computer are needed at all times.

For instance, an algorithm to compute the square root may give a wrong result only when required to take the square root of 0, but it can take years until somebody notices it or the computer is scrapped. Equally, a transistor may be short-circuited in a part of the logic that is very seldom used.

This provides a chance for the fault to be found and repaired before the element is actually used. However, if such a fault remains undetected, it can cause a failure at an unexpected moment.

In a fault-tolerant system, latent faults may exist in the redundant, unused spares. The latent faults are called **lurking faults**. The technique to uncover lurking faults is to exercise all functions regularly.

Example:

without regular testing, fire extinguishers and fire alarms lose effectiveness, but their presence gives a false sense of confidence.

3.1.6 Fault Summary

We resume the different forms of faults:

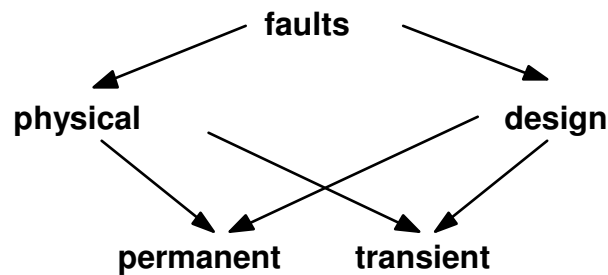


Fig 3-3 Résumé

3.2 Redundancy forms in computing systems

3.2.1 Redundancy Role

Redundancy is any resource which is included in expectation of faults, but which is not needed for normal operation. Fault tolerance supposes that redundancy is available, i.e. more resources are provided than for an error-free operation.

Redundancy serves two different purposes in a fault-tolerant computer:

- _ Detect errors,
- _ Correct errors and continue operation.

The same redundancy cannot be used for both error detection and error recovery.

Example:

Two identical computers execute the same program on the same variables at the same time. Their output should be identical (Figure 3-3):

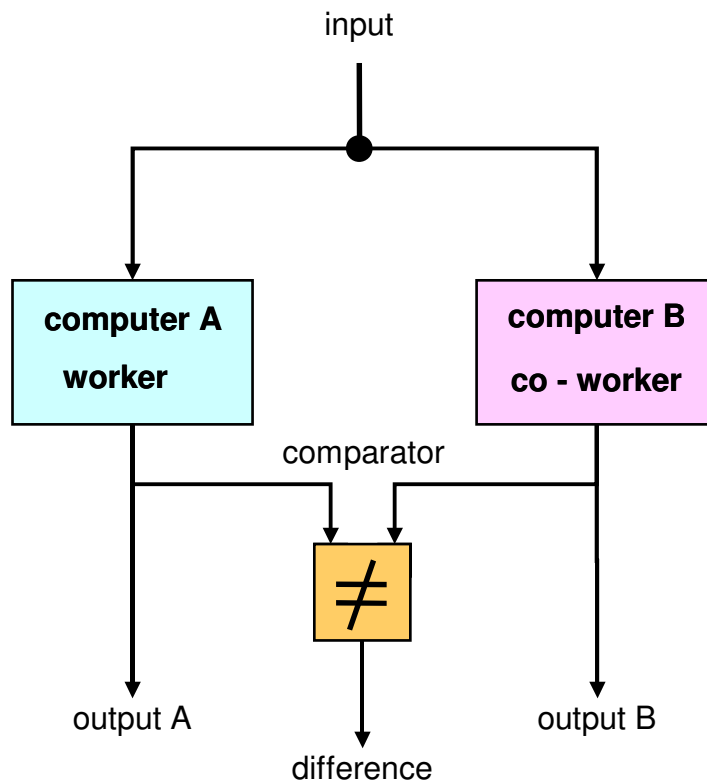


Fig. 3-3: Duplication and Comparison.

One can use this redundancy to detect an error by comparing the outputs. A discrepancy means that one of the computers has failed, but does not tell which one. This information can be used to shut down both computers to obtain a "fail-stop" behaviour, but it is insufficient to provide continuous operation. For this, a second redundancy would be required to decide which unit is still working.

For error detection, **plausibility redundancy** is required, i.e. the redundancy does not need to execute the same function as the unit it checks, but should detect as many faults as possible in that unit.

For fault-tolerance, **functional redundancy** is required, i.e. the redundant unit must be capable of performing the same function as the unit that failed if required.

Example :

Consider a communication link which transmits files. If no errors are expected, the file is transmitted and treated as correct. Now, if errors are expected, the file is transmitted with a checksum. The checksum has typically a length of 1 byte, while the file may have a length of 32 bytes. An adequate checksum algorithm can detect all single errors in the transmission. A wrong checksum signals an error. If the process has a safe side, this error signal can prevent the user from using the files, and this is sufficient to implement an integer system.

The checksum is not sufficient to correct an error. In case of a discrepancy, the receiver knows that an error occurred, but not which bits are faulty.

Correction must be done by requesting a new version of the file, which hopefully will be fault-free. Therefore, correction needs a functional redundancy.

In reality, more redundancy is needed because of the overhead associated with the handling of error messages.

Note that redundancy can reduce the **availability** of a system. For instance, the above system used as a fail-stop pair will have a lower reliability than a single module, since it will now fail at least twice as often because of the added unit and error detection circuits. The fail-stop pair will however have a far higher **integrity** than the simplex system.

One key question of fault tolerance is whether the unreliability due to the added complexity introduced for this purpose can be outweighed by the fault tolerance properties achieved. This question will be discussed in Chapter 9.

It should be remembered that redundancy is useless if it is not continuously checked. The presence of lurking faults is a critical issue in fault-tolerant computers. Even non-repairable systems need checking of the redundancy in order not to insert a faulty spare, while a good one exists.

The redundancy may be present in form of additional **hardware**, in form of additional **software** or in form of additional **time**.

3.2.2 Hardware Redundancy

Hardware redundancy consists of additional hardware elements that duplicate and check the functionality of existing elements. It is used to detect and correct hardware errors. We distinguish three forms of hardware redundancy:

1. **Plausibility redundancy** consists of hardware circuits, which indicate certain kinds of error. Among them are power monitoring circuitry, signal quality checks in serial links, and protocol monitoring functions.
2. **Functional redundancy** is a total replication of the functional part (hardware). It can execute the same operations as the working unit, provided it is loaded with the correct information. Unlike the preceding redundancy forms, functional redundancy can be used either for error detection or for continuous operation (but not both at the same time).
3. **Coding redundancy** consists of circuits used for error detection that apply coding techniques. Examples are: parity computation, checksum, error code generation and checking. Such codes can detect a limited number of errors in the data items they check. Some codes also correct errors up to a certain number. Coding redundancy therefore holds an intermediate place between plausibility and functional redundancy.

Plausibility costs little and coding causes only moderate costs, but functional redundancy requires a duplication of the hardware and at least doubles the cost.

3.2.3 Software Redundancy

Software redundancy is used to fight software or design errors. Software redundancy requires a redundancy of design to reduce the failure due to a common source, i.e. alternate or **diverse** designs. A requirement for diverse software is the independence of the designs. This leads to a complete replication of the software production process by independent teams (N-version programming [Chen 78]).

Three-version programming is today used on ultra-high-reliability computers, such as the Airbus or Space Shuttle navigation computers.

To reduce the price of diverse software, some have suggested using previous versions of the same program as alternates. The new versions should provide the same functionality, but could be more optimised than the old version and provide additional services. In fact, this is one of the most common uses of software sparing: a new version is installed, which provides some added functionality but remains untested. Upon detection of a software error, the old version can be installed back again. This works only if both versions work on the same data set and have the same service interface. If the data sets are different, then the switching from one spare to the other requires a redundancy of the data sets.

Although hardware is subject to design errors, diverse hardware designs are very rarely considered. An exception is the space shuttle computer, of which four processors are built and programmed by one manufacturer and the fifth by a different manufacturer.

Although conceptually different, "software" redundancy can be treated similarly to "hardware" redundancy for all practical purposes.

3.2.4 Time Redundancy

Time redundancy comes in the form of a "grace time" during which the outside world is not affected by the failure. Time redundancy allows for corrective actions and repetitions. How much time redundancy is available depends on the plant concerned. The optimum strategy for fault tolerance often depends on the amount of time redundancy available.

3.3 Error detection

Error detection is basic to fault avoidance, fault tolerance and integrity. Indeed, all fault-tolerant system require the previous detection of errors, except possibly systems which work by massive redundancy and ignore repair. In principle, complete error detection is sufficient to implement total integrity. This is required by fail-stop systems that would rather stop than output false data. Although fail-stop systems are not fault-tolerant (they stop operation in case of error rather than recovering) they are an important building block for fault-tolerant systems.

In principle, 100% error detection requires 100% redundancy of the resource that is expected to be erroneous. The quality of the error detection is basic to any fault-tolerant or fail-stop system. The error detection device must be matched to the kind of error it is intended to detect to be efficient. The quality of error detection is measured in terms of how many simultaneous errors it can detect.

One distinguishes between:

- **Initial checking**, which takes place prior to operation. It begins with the testing of the components at reception and before insertion in the boards and ends with the pre-operation check, for instance before launch of a spaceship. This checking ensures that the system is error-free before operation, an assumption which is always made in fault-tolerant systems. Initial checking is considered as a technique for fault-avoidance, rather than fault-tolerance.
- **Concurrent** or **on-line** error detection takes place during normal operation and checks the correct execution either by dedicated hardware circuits or by replication of hardware and software. Concurrent error detection is capable of detecting transient errors and permits one to build fail-stop units. The falling prices of hardware permits the inclusion of concurrent detection circuits at a low cost. However, parts of the machine that are not currently used cannot be checked by on-line detection.
- **Scheduled** or **off-line** error detection is executed by dedicated maintenance programs. A background task runs periodically to exercise every hardware unit of the machine. Off-line tests are shorter than initial tests and cannot rely on external help. Off-line tests need no redundant hardware but require test software and time redundancy. It is therefore appealing to implement off-line tests on existing hardware. Off-line detection has the advantage over concurrent detection that it can exercise the entire machine, and not only those parts which are currently used. It can, in particular, uncover lurking faults. On the other hand, off-line error detection is not capable of detecting transient errors and therefore cannot be used to implement integer systems.

Error detection is often coupled with **error confinement**. In a computer, the information can flow very easily and errors may propagate through the whole system and corrupt it if means are not taken to stop them. Therefore, the computer is divided into hardware and software confinement zones, which play the same role as firewalls in buildings or compartments in ships. An undetected error that is allowed to leak out a confinement zone can corrupt another confinement zone. Worse still, these data are treated there as legal data, since they came out of a protected zone.

The confinement regions are normally identical with the replaceable units (RU): ideally, an RU is self-checking and can confine errors within itself. We will consider hardware confinement regions in the next section.

3.3.1 Hardware Error Detection

A computing system consists of four kinds of components:

1. storage (RAM, disk)
2. transmission paths (buses and links)
3. computation units (processors).
4. auxiliaries (power supply, timers, etc...)

Auxiliaries are protected by special means. Power supplies are checked for correct voltage and frequency, timers are checked by timer bound counters, the temperature is monitored, etc...

In the rest of the computer, error detection requires redundancy in storage, transmission and computation. These components coincide with separate confinement zones (Figure 3-4):

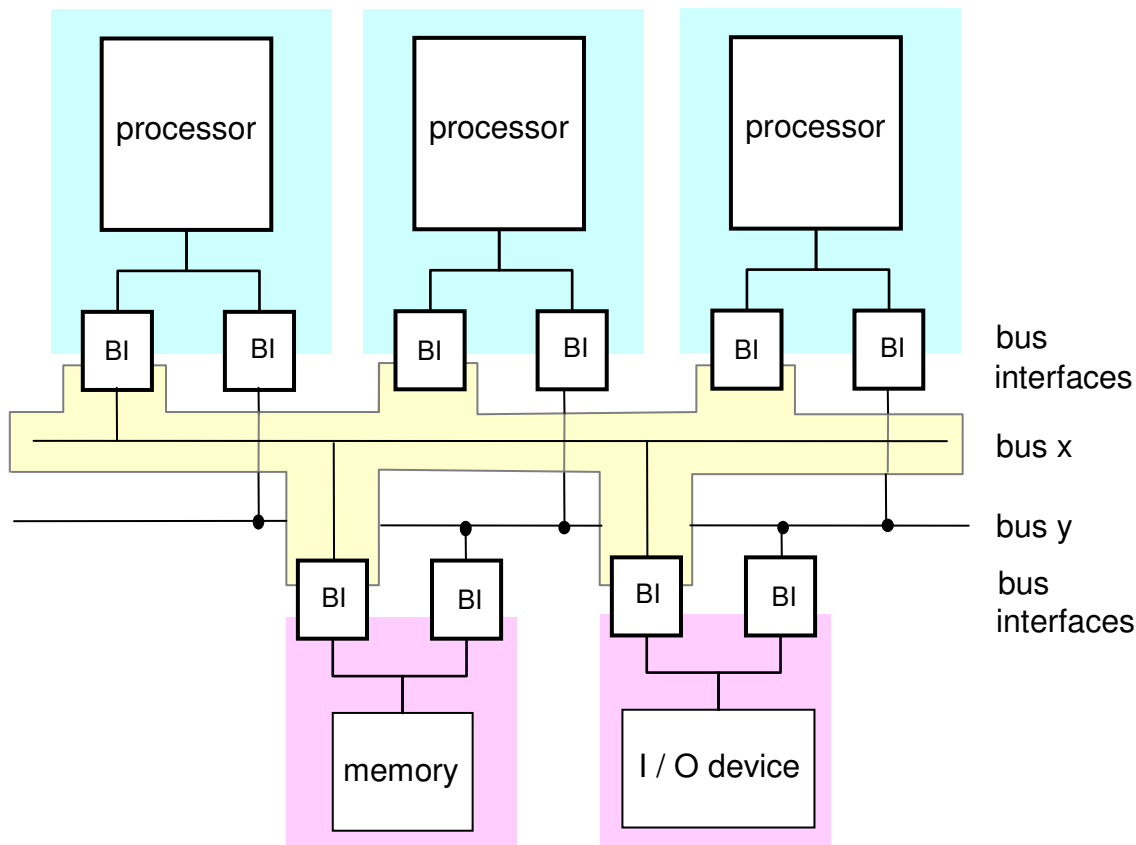


Fig. 3-4: Processor, Bus and Memory Confinement Zones

Errors in **transmission paths** are detected by coding. The code to be used depends on the expected error rate. Parallel backplane buses, which are relatively free of disturbances, are simply protected by parity. Short serial buses like MIL 1553 are also protected by parity. Serial links exposed to disturbances are usually protected by cyclic codes, which can detect a large number of errors and are tailored to a specific medium, for instance to account for header corruption, synchronization errors or certain burst errors.

Errors in **memory elements** are also detected by codes. The simplest code is parity, which cost only about 1/8 redundancy, and can detect single errors. Since memories are cheap, Hamming codes are also used which perform detection and correction of errors. These codes are stored along with the useful data, but require only a fraction of the logic of the useful data. For instance, to protect effectively a memory with a width of 32 bits, 7 additional bits need to be stored to detect every single and double error and correct every single error.

Errors in **processing elements** are more difficult to detect. The internal complexity is so high that it is relatively difficult to find efficient codes for error detection, since data are not only transported and stored, but also modified. The checking unit can become about as complex as the checked unit, but solutions exist as silicon becomes cheap. The most suitable method until now is to use duplication and comparison: complex elements are used in pairs and checked by letting them execute the same operations at the same time. An error detecting circuit then compares continuously the outputs of the replicated halves (Figure 3-5):

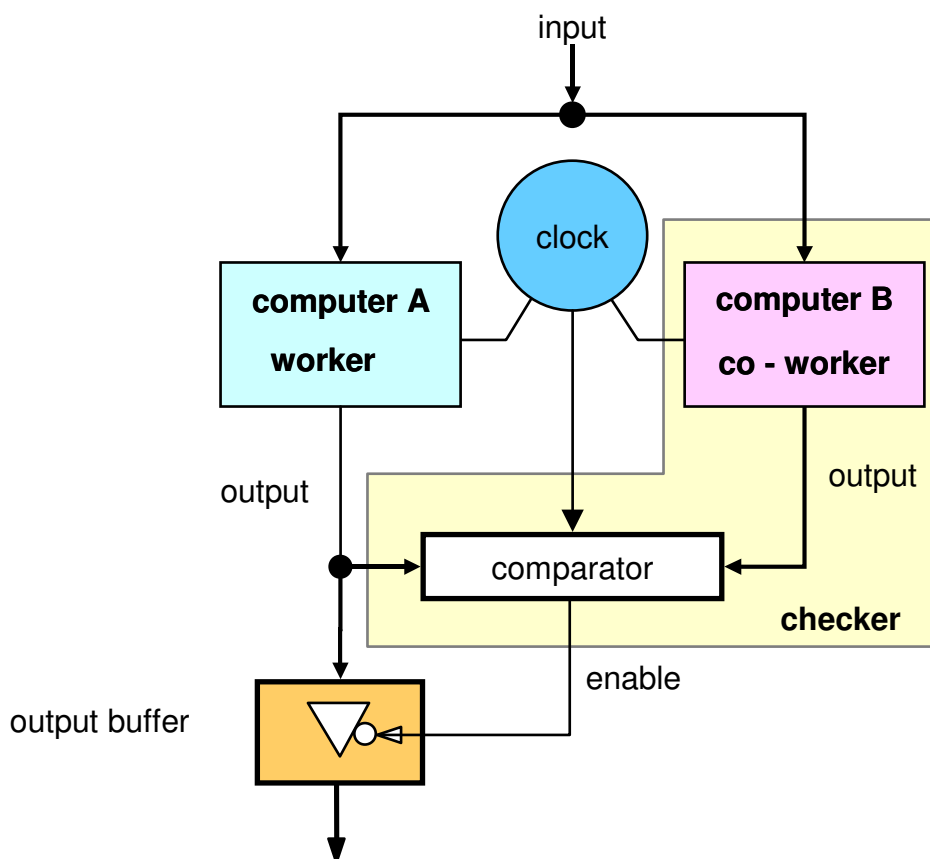


Fig. 3-5: Error Detection by Duplication and Comparison.

This method is called "worker/checker" or "duplication and match". It is used in numerous designs, such as in the ESS processors [Toy 78], in the COPRA computer [Meraud 79] or in Ericsson's AXE 10 [Ossfeld 80]. Integrated circuits already exist which let any chip operate as worker or checker, such as the iAPX 432 [Intel 82] and Advanced Micro Devices's series 29300 bit-slice processor (Figure 3-6):



Fig. 3-6: Worker and Checker Mode at the Chip Level.

The worker/checker mode assumes that the replicated units are initialised in the same state, that they receive exactly the same inputs at the same time, that their outputs are synchronized and that in general the circuits are completely deterministic.

The synchronization between the two units can take place at the processor clock level (micro-synchronism) by using a common clock or at the level of bus transfers (bus synchronism) or at the input/output level (I/O synchronism). The tighter the synchronism, the greater the probability of failures that affect both units at the same time in the same way. The problem gets more complicated when several clocks exist (one for the processor, one for the baud-rate generator, etc...), and especially when asynchronous events must be considered, such as interrupt signals. Then, complex synchronization circuits are required.

Fortunately, synchronization for error detection needs not to be perfect to allow 100% error detection. Synchronization errors will be flagged as a transient hardware error. Synchronization errors nevertheless impair availability. We will come back to this problem again when considering work-by synchronization (Chapter 4).

The common hardware error detection mechanisms are summarized in Figure 3-7:

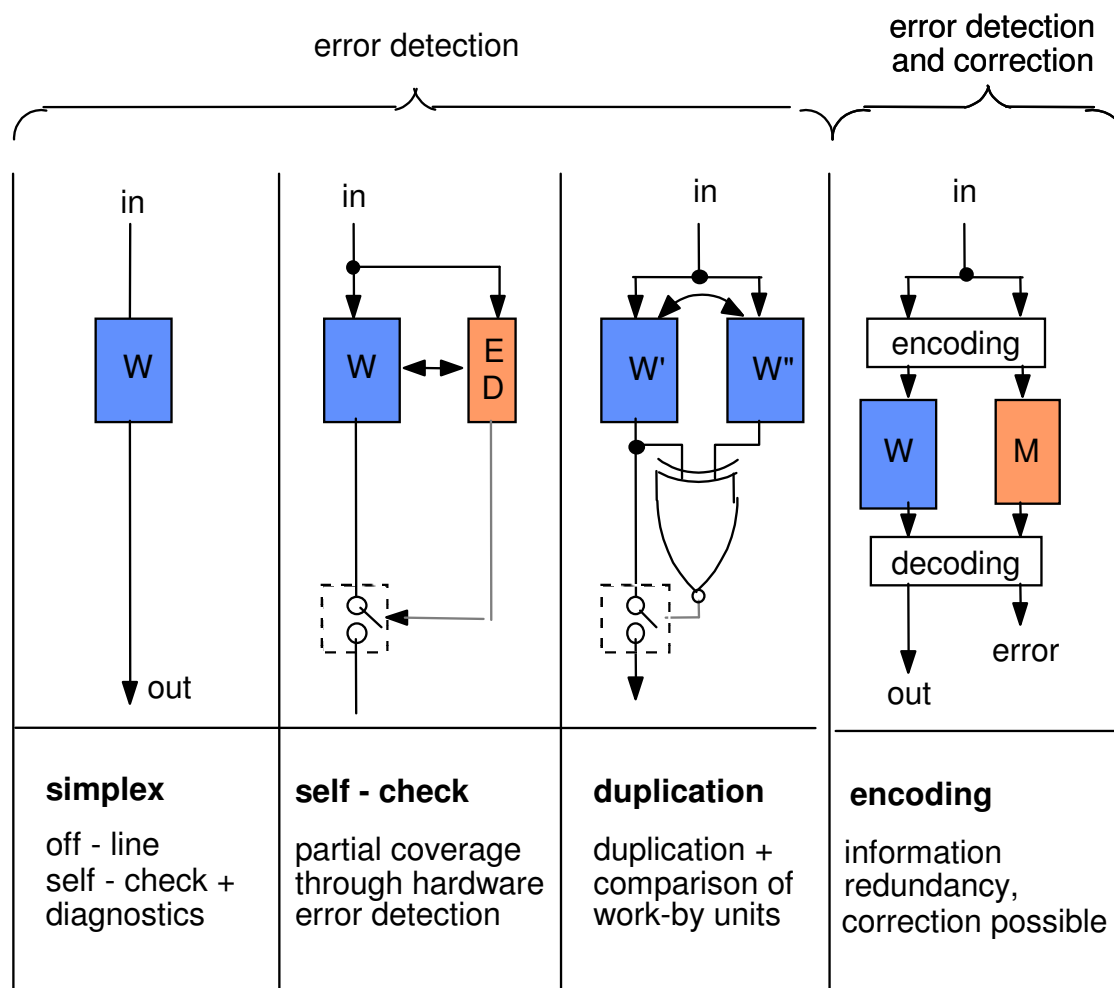


Fig. 3-7: Hardware Error Detection Architectures.

3.3.2 Software Error Detection

The detection mechanisms used against physical faults are insufficient to detect software faults. Software errors can only be detected by a redundant (alternate) design.

100% coverage of software errors requires at least two **alternate designs** in which independent programmers program the same application. A comparison program that should be 100% reliable then compares the results of the applications.

The comparison program does not simply compares data, but requires a great deal of intelligence.

Example:

if the programs generate real numbers, rounding errors occur if the floating-point operations are not exactly identical. The algorithms used in both programs must therefore be identical. Then, the same algorithmic bug in the mathematical routines can introduce a common mode error. The efforts of the IEEE Floating Point Standardization (IEEE Task 754) could bring a set of verified floating-point routines, but then the verification should also extend to the implementation.

The other solution is to let the programs diverge, and to smooth the results in the comparison module. As long as the controlled process is continuous, it is relatively simple to delay the results and build a plausible value. But it requires a great deal of intelligence to match for instance characters sent to a video display: one program could write "blower on" and the other "fan operates". Small details like void lines and uppercase/lowercase can cause errors and dependency on a correct and comprehensive specification is therefore high.

N-version programming is therefore not trivial, and suffers from the fact that, at the root, the requirement specifications are common for all programmers. In the SIFT project [Schwartz 83], the designer tried to overcome the specification hurdle by developing formal specification languages, which can be validated [Goldberg 84]. Using specification languages only shifts the problem to the detection of errors in the specifications. They do not solve the problem, but just displace it to a level that can be easier to manage.

Common mode failures exist not only because of bugs in the comparison program or in the specifications. People tend to do the same program errors in the same situation, because of common modes of thought possibly induced by the school they attended. Experiences with diverse software are described by [Chen 78, Avizienis 84].

Usual techniques that do not claim 100% error detection involve check within the program. The simplest means are provided by the compilers themselves, which intercept programming errors during the compile phase by checking the syntax as well as the consistency of predicates like variable types and procedure parameters.

In addition, compilers generate code to protect against out-of-bound array indexing or stack overflow, incorrect arithmetic operations and invalid pointer references at run-time.

These techniques have been developed further in object-oriented architectures, which check at run-time for the correct typing of data and limit the address space to the absolute minimum currently needed to run the program [Intel 81]. Software errors that cause pointers to be misplaced can be detected that way, but at a great expense in processing time. For that reason, some languages like PORTAL and CLU do not allow the use of pointers.

Further, an independent unit can monitor the program execution. This unit checks whether the program flows through legal paths and passes at determined checkpoints. The monitoring unit contains a model of the computation that is a form of software redundancy. For instance, the monitoring unit can check the running program against its description in form of a finite state machine or a Petri net.

Good programmers anyway include a large quantity of checking routines: validity and plausibility checks, acceptance tests, checking of predicates. Some estimate that about one-third of the application software in large projects is inserted to deal with exceptional situations.

None of these techniques can ensure 100% error detection, especially taking into account that the person who designs the tests is often the same as the one who wrote the program to be checked. Further, software checking is done after the values have been calculated, and therefore are much slower than on-line hardware error detection - but hardware detection is no replacement for software checks.

Therefore, software errors should be tackled by fault-avoidance techniques in the first place, and not by fault-tolerance techniques at run-time.

Once a design or software fault has been corrected, the fault is finally removed. But the correction may itself introduce another fault. This is why one hesitates in correcting complex software errors when they are recognized: it is sometimes safer to program around bugs than to correct them. An application of this philosophy is found in the Space Shuttle software.

3.3.3 Time Errors Detection

Timing errors reveal hardware and software faults, since the untimely execution of a program can be due to either kind of fault. Timing errors are detected by setting watch-dog timers, which are generally dedicated hardware devices. When the program starts a time-critical section, it triggers the timer that it should reset on leaving the section. If the program has not left the section before the time-out elapses, the timer generates an interrupt or an error signal.

The use of timers becomes problematic when the programs are non-deterministic, especially in hierarchical systems. Then, the timer has a certain probability of ringing, although no software or hardware fault occurred, just because of an infrequent temporary congestion. In reality, this congestion could also be an error, since a timing constraint has not been respected.

To control the complexity of hierarchical systems, the timers are sometimes tied to a certain level of nesting. For instance, a subroutine is allowed a certain time to execute. If that routine calls another routine, its timer is stopped, while the timer of the called routine runs.

The health of a system can also be monitored by a background check task that periodically checks the system. It resets the timer if the test is successful. If that task does not run in time or gets stuck for any reason, the timer triggers an error signal. Conversely, that task could also generate an "I'm alive" message for other modules at regular intervals. If the message is overdue, the other units will assume that a fault has occurred.

Timing errors also can manifest themselves as data errors, for instance in the case of buffer overflow.

3.3.4 Checking the Checker

Error detection is never perfect. First of all, some errors may remain undetected if their occurrence was not foreseen in the design. For instance, the simultaneous inverting of 2 bits fools a parity detector. Total replication does not guarantee 100% error detection coverage.

Other errors may not show up immediately. For instance, when a fault within a checked unit does not have an immediate consequence on its output, it therefore remains temporarily undetected. The time it takes between the occurrence of a fault and its detection is called **error detection latency**, or **latency**.

Latency may be reduced by periodically exercising all parts of a unit, in such a way that they affect the outputs.

A fundamental problem of error detectors is that the error detection circuit or algorithm may remain stuck in the "OK" state and that from that moment on a fault of one unit will no longer be noticed (lurking error). The non-function of an error-detecting device is equivalent to the loss of redundancy. Indeed, the reliability of the error detecting circuit determines the reliability of the whole plant.

Therefore, the checking circuit must also be exercised at regular intervals, by injecting false signals and monitoring the checker's response. This interval can be calculated knowing the probability of failure of the error detector and of the redundant parts, as we will see in Chapter 9.

Alternatively, the checker may be built using self-checking circuits, for instance in the **two-rail technique**. In this technique, each logical signal is carried by antivalent lines, one carrying the signal and the other its logical complement. Both lines go through separate circuits, like Figure 3-8 shows. An error is detected when the lines cease to be antivalent. A whole theory of self-checking checkers, also called **morphic checkers** has been developed [Duke 72, Carter 72].

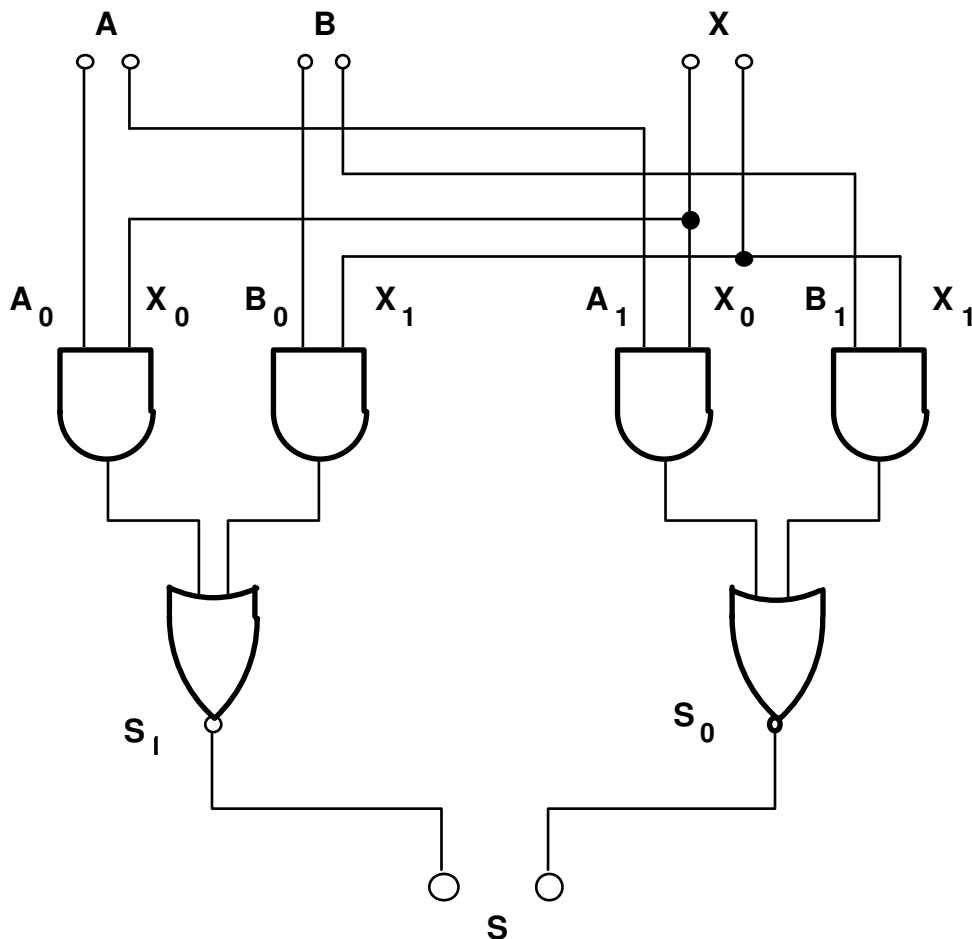


Fig. 3-8: Self-Checking by Dual-Rail Technique.

3.3.5 Checking the Redundancy

Another lurking fault results from the loss of a spare without notice. The system is running with a damaged spare, but the recovery logic does not know it. The situation differs according to the three forms of hardware redundancy we have seen:

- Replicated computing detects the faults in a unit that appear at the outputs. A situation can nevertheless occur in which one unit suffers a fault that does not show up at its outputs. The active unit may now fail, and if the recovery process makes use of the faulty part of the spare, the system fails. The solution is to let the spare expose frequently all its internal states and exercise the recovery procedure.
- Warm spares naturally exercise the hardware. Lurking errors may exist in the redundant storage (the saved state may be corrupted). Scheduled error detection programs also must check this. In particular, if the storage is capable of single error correction, the correction must be applied sufficiently frequently so as not to let a second error develop.
- Cold spares are the most difficult to check, since they are not in use. For instance, vacuum tubes and mechanical parts are shut down to augment their life. Lurking errors must be found by reactivating the unit from time to time and performing scheduled testing on them.

3.4 Fault-tolerance

Increases in availability and reliability rely on fault-tolerance, i.e. the ability to continue operations after the occurrence of a fault. By contrast, error detection allowed one only to build fail-stop systems, which were safe only when the plant had a safe side and which reduced availability.

There are two basic principles for fault-tolerance, fault masking and fault recovery.

Masking hides the faults by coding or replication and provides a no-downtime switchover. Masking requires that spares are permanently inserted or can be inserted within a negligible time.

Recovery involves an intelligent process to correct a fault and implies a non-negligible down-time in case of fault.

Both rely on functional spares, but at different degree of readiness.

3.4.1 Functional Redundancy

Fault-tolerance, and also error detection by duplication and comparison, relies on functional redundancy. Functional redundancy in a computer appears in form of spares or replaceable units (RUs), also called Line or Field Replaceable Units. A replaceable unit is generally a piece of hardware or software that is self-contained and has a defined interface towards the rest of the machine.

Earlier designs introduced redundancy at the component level. Siemens developed a logic family in the 60s for that purpose. Complete computer for safety applications were built by this technique, but they have found little application since.

Although smaller RUs tend to increase the reliability of the system, there is a limit to the gains in reliability that is explained in section 9.3.6. One problem is that the independency of fault between two small RUs is difficult to evaluate and certify.

Traditionally, replaceable units have been identical to the serviceable processors, memories, buses, links, and peripheral devices, which roughly correspond to the building blocks of a computer. Today, the tendency is clearly toward larger RU, e.g. complete computers.

At the same time, the RU are combined with error detection mechanisms to build error confinement zones. The ideal would be to have fail-stop RUs.

3.4.2 Spare Readiness

The architecture required to achieve fault tolerance depends in the first place on the maximum allowed recovery time, or time redundancy. As described in Chapter 1 one distinguishes computing element spares according to their degree of readiness:

- **Hot spare:** the spare hardware has exactly the same state as the working unit. The spare is either inserted permanently (massive and voting redundancy) or it can be inserted within a **negligible time**, by flipping a switch (hot-sparing redundancy). In that last case, it may be necessary to repeat the information transfer that took place at the moment of switchover. Therefore, an error can be perfectly masked. The switchover is either instantaneous or takes some ms.
- **Warm spare:** the spare is loaded with a valid state, typically obtained by periodic copies from the working unit's state. The insertion of the spare is bound with a **short loss of function**, due to the operations necessary to actualise the spare by redoing computations. In addition, the interactions of the working unit with the outer world must be monitored to avoid asking again for information already received or to send out again data already sent. Current commercial computers can perform a warm switchover in some seconds.
- **Cold spare:** switchover involves a **relatively long loss of function**. This is the case when the spare is void of memory, for instance when disk storage is lost and must be reloaded from tape, when the computer must be taken off-line and repaired or even is exchanged against a new unit. The spare must not only be loaded with a correct information, but also get itself acquainted with the situation it finds at restart (for instance it may have to poll all peripherals before starting work). In addition, there is a loss of the interactions with the environment that took place since the last saving point. A cold switchover takes from a dozen seconds to several hours, depending on the extent of the damage.

The main problem is how to maintain spares in an actualised state when they contain memory: it is not sufficient to provide replicated memory hardware; this hardware must contain the correct information as well.

- **Transmission links** and **busses** are memoryless. It is relatively simple to switch from a defective transmission link to another. Possibly, some registers and buffers in the interface must be reloaded or cleared.
- **Memory** is relatively simple to keep actualised. The classical solution is to write to duplicate memories at the same time. When the memory is large, this is not always economical. We will see other solutions in Chapter 6.

- **Processors, I/O controllers and complete computers** are more difficult to maintain actualised, since their internal state is extremely complex. There are two basic ways to maintain such elements actualised:

Replicated computing or **work-by** is used to implement **hot sparing**. The spare and the working unit execute the same function at the same time. Since computers are deterministic machines, their internal states and outputs should be identical, provided they receive exactly the same inputs at the same instruction and work in close synchronism. Thus, which unit is actually the working unit and which is the spare unit makes no difference: it depends on which unit is connected to the outside world.

State saving or **stand-by** is used to implement warm sparing and cold sparing. There is only one working unit, whose state must be saved at regular intervals, either in a dedicated storage (cold stand-by) or directly into a spare hardware (warm stand-by). Stand-by units may be used for other purposes while the working unit is error-free, or they can be maintained in a reduced failure rate state, for instance unpowered (cold redundancy).

3.4.3 Masking

Fault masking hides the fault from the outputs by proper coding or by replication of the computing units. There is no downtime bound to the fault treatment and as long as sufficient redundancy is present, the faulty case cannot be distinguished from the normal case. Fault masking is associated to the notion of massive redundancy, in which the work is executed by a large number of units working in parallel, or work-by techniques. A selection or a majority vote is taken on these units to choose the most probable output. One distinguishes:

- **coding redundancy**: the information is coded with an error correcting code. The algorithm can correct a limited number of bit errors.
- **massive redundancy**: the process builds a plausible value output by several replicated units.
- **voting redundancy**: a dedicated voter builds the most probable output among several units working in synchronism
- **sparing redundancy**: one unit does the work, the others are hot-spares which can be inserted in place of the faulty unit at any time.

The masking techniques are summarized in the following Figure 3-9:

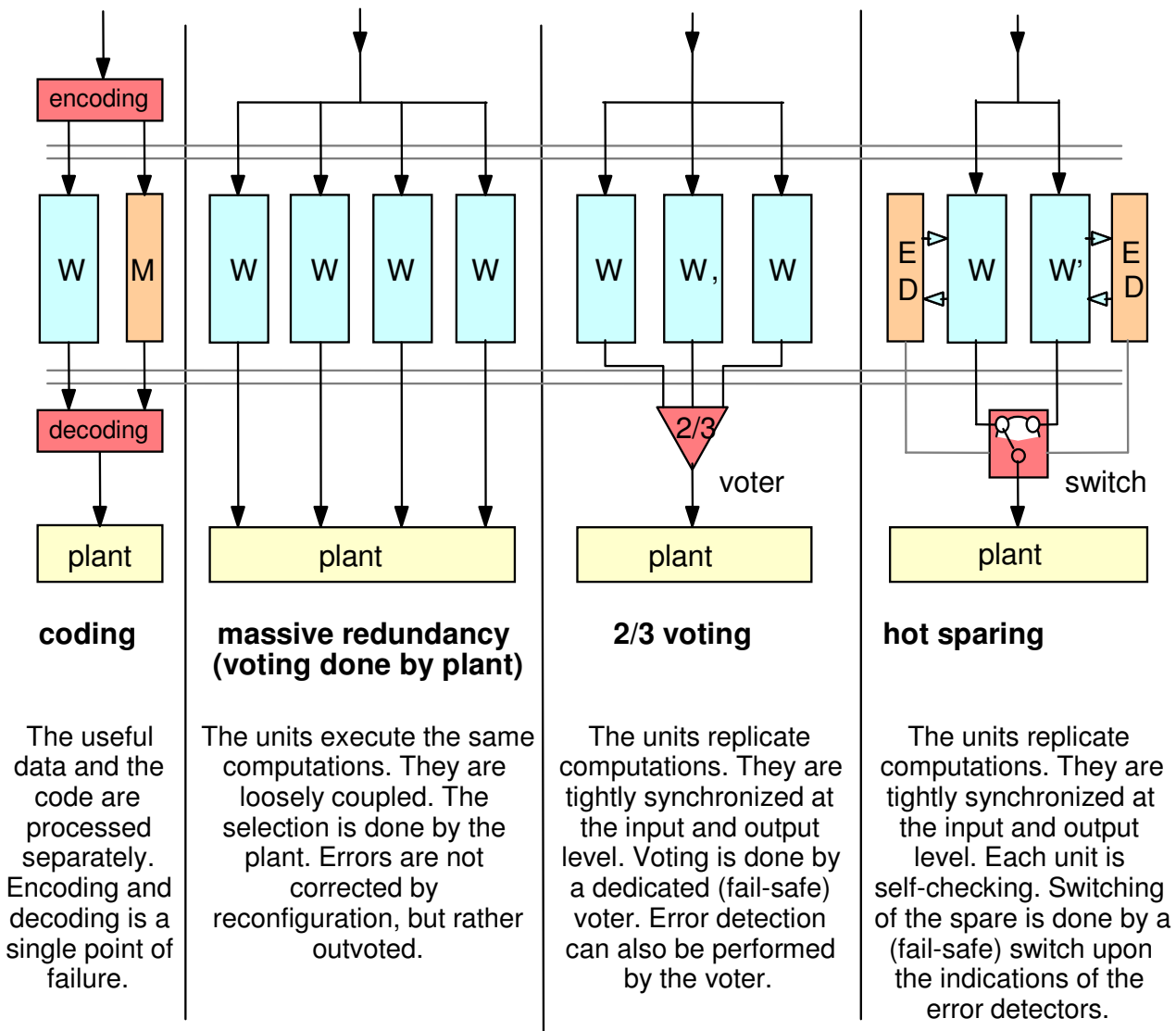


Fig. 3-9: Masking Architectures.

The redundant path is contained between the dotted lines. The components that are outside of the redundant path are single points of failure. They should themselves be made redundant if their reliability is insufficient.

3.4.4 Recovery

The other fault-tolerance technique is **recovery**. Recovery is associated to the notion of software redundancy and stand-by techniques. Recovery tries to correct the effects of a fault rather than to hide them. Recovery enables the amount of redundancy to be reduced with respect to masking at the expense of a higher complexity. At the extreme, only redundant storage is required. Common to all recovery techniques is the restoration of a valid state from which computations can be resumed, called a **recovery point**.

Two methods to reach such a recovery point exist: backward error recovery and forward error recovery.

Backward error recovery, or **retry** attempts to redo the failed task on the same or on other hardware. Retry consists of two phases: Rollback and Roll-ahead:

Rollback restores a recovery point that is either a copy of a previous, error-free state of the machine or a reconstructed state that may have existed some time before the fault. Rollback is done by:

- Reloading the lost storage parts,
- Correcting the affected storage,
- Including the effects of former computations and
- Restoring the environment as far as possible

Roll-ahead then redoes the failed task starting from the recovery point while:

- _ Correcting the effects of the first computation
- _ Ensuring no loss or duplicates of inputs and outputs.

To do both rollback and roll-ahead, enough information about the progress of the running task must be saved at regular intervals called save points in redundant back-up storage. This back-up storage can be a tape, a disk, a non-volatile semiconductor memory or the main memory of a spare unit.

forward error recovery or **compensation** attempts to use the erroneous state and build out of it an error-free state, which may never have existed before. In the simplest case it can just cancel the failed task. Forward error recovery is related to the techniques of exception handling. It is application dependent. Forward error recovery requires less redundancy than backward error recovery, but at the expense of additional complexity in recovery. Indeed, the complexity may be so high that artificial intelligence methods become appropriate.

We distinguish the following recovery architectures, to which both forward error recovery and backward error recovery may be applied:

- **Warm stand-by**, also called main/back-up or primary/secondary. A stand-by spare exists which has the capacity to perform, if necessary, the same work as the working unit. Normally the spare unit remains idle or performs other tasks. The working unit keeps the spare actualised by transmitting its own state directly to the memory of the spare at every save point. Upon detection of an error in the working unit, the spare is switched in place of the working unit. The recovery takes little time since the spare is already loaded with the current task state. Special care must be taken to treat interactions with the outer world that took place since the last save-point. For this purpose, the stand-by unit should divert a part of its computing power to log the execution of the working unit.
- **Cold stand-by**: There is no dedicated stand-by spare, or the spare is void of information. The state of the working unit is saved at each save point in a save storage (for instance a disk). In addition, the progress of input and outputs is recorded in a log (journal). When an error occurs, computations can be resumed on the faulty unit if the fault is transient. If the fault is permanent, the unit must first be repaired or a cold spare must be inserted. In both cases, the spare or repaired unit is reloaded with the state saved in the save storage to establish a recovery point and computation is resumed from that point on. Special attention must be paid to the interaction with the outer world. While the spare in warm stand-by could monitor all interactions with the outer world, cold standby suffers from a period of amnesia. A general solution does not exist. We will consider especially the application of this method to a database in Chapter 7.

These architectures are summarized in Figure 3-10:

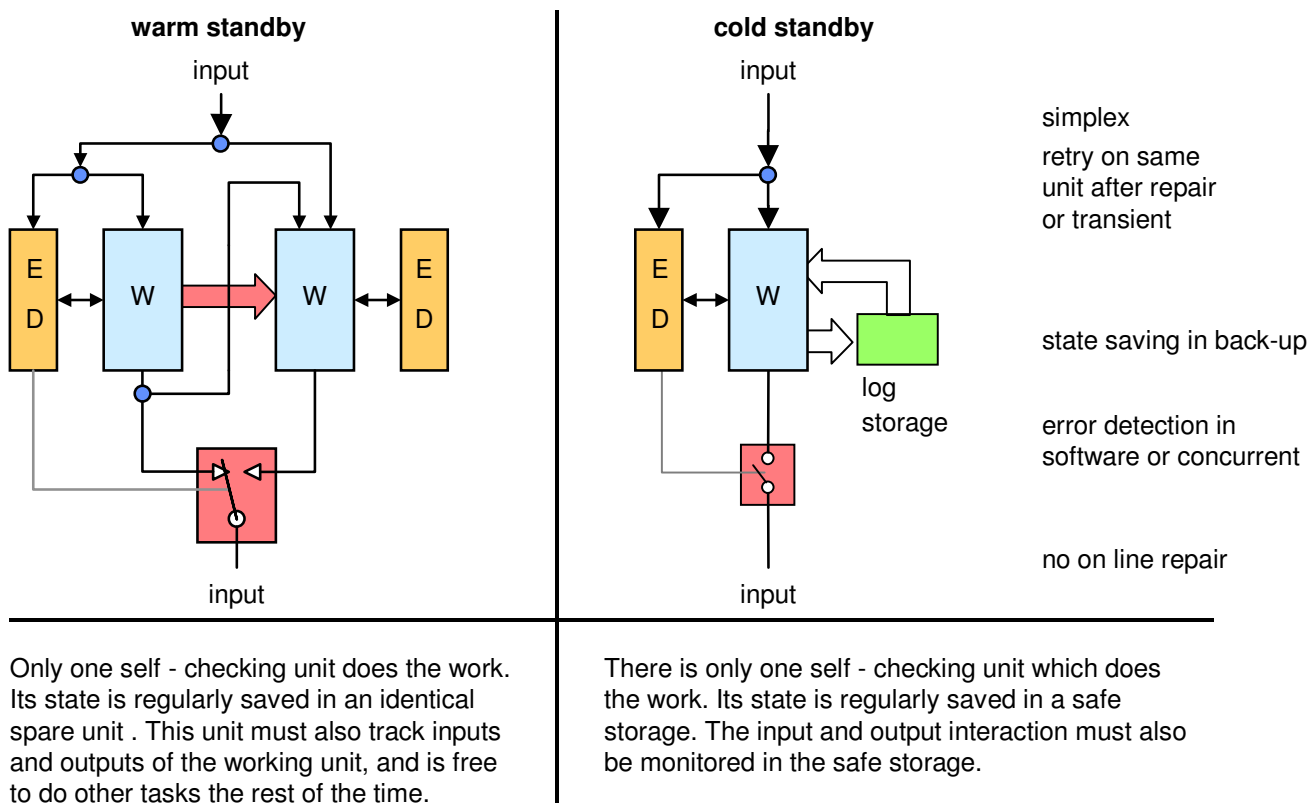


Fig. 3-10: Recoverable Architectures.

As in Figure 3-9, the part within the dotted lines is redundant. Recovery will be treated in Chapter 5, state saving and restoring in Chapter 6 and the special problems of databases in Chapter 7.

3.4.5 Hybrid Techniques

Masking is generally associated to the hardware fault-tolerance methods, while recovery is considered as a software fault-tolerance method. Indeed, the two methods are sometimes combined, for instance by using recovery techniques in case masking fails. The most common is however to use recovery techniques to reintegrate a repaired unit into a functioning set without shutdown of the working unit(s).

An example is Ericsson's AXE system [Ossfeld 80]. The AXE uses hot sparing normally: two work-by units are executing the same work at the same time in a synchronous manner. The outputs are continuously compared, and the two units work as a self-checking pair. Upon detection of a discrepancy, the units separate and each one runs diagnostic programs to determine if it is faulty. The faulty unit disconnects itself and the non-faulty unit keeps on. The faulty unit can then be repaired and checked. This mode is possible since the AXE computer aims at high availability and has a relatively long grace time.

To reintegrate the repaired unit as a spare, the working unit transmits its state into the repaired unit, as is done for recovery. One unit can also be disconnected for other purposes, like for testing new versions of the software.

3.4.6 Spare Pooling

Spare pooling, also called hybrid-parallel, is a very general form of fault tolerance, which uses masking as well as sparing techniques. The idea is that a fault-tolerant system can be configured out of a quantity of processors, memories and buses.

Each processor, memory or bus acts as a replaceable unit (RU). Fault isolation is either enforced by hardware (voter, self-checking units) or by software (acceptance programs). If the RUs are interchangeable, then in principle one or several of them can serve as spare for any other, thereby reducing the hardware for redundancy.

Figure 3-11 shows the case for 2/3 voting:

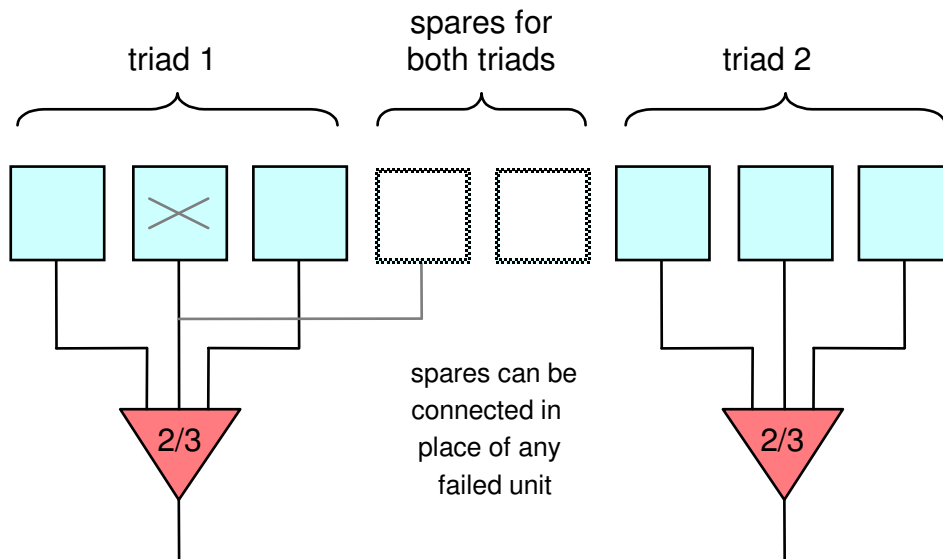


Fig. 3-11: Spare Pooling.

For two-of-three voting there are always at least three RUs, called triad working in parallel (examples are: SIFT, FTMP). For hot-sparing, there is always a pair of self-checking units (sometimes called a "quad") working in parallel ([Katzuki 78], iAPX 432 [Intel 81], STRATUS/32 CPS [Freiburghouse 82]).

The recovery procedure is similar to that of hot-sparing systems. Upon failure of an element the computation is kept on with the remaining redundancy until a natural ending point is found, such as the end of the current task or return of the current procedure. To restore full reliability, the triad, (respectively the quad) is dismantled, and a spare from the pool is inserted in place of the faulty unit or pair.

The reconfiguration phase is complicated and involves a constant assessment of the available resources and the redistribution of these resources after a failure. Reconfiguration requires that in all cases a part of the functionality is retained to run the reconfiguration routines.

Some designs leave the reconfiguration task to a highly reliable part which is dedicated to that task, and which is implemented by voting redundancy techniques, called the hard-core. Such is the JPL-STAR [Rennels 78].

Others distribute the reconfiguration task among all processors. Such multiprocessor systems have been implemented for high availability applications (Pluribus, iAPX432) and high reliability applications: FTMP [Hopkins 78], SIFT [Wensley 78, Goldberg 84], AFTC [Schmid 84]).

3.4.7 Graceful Degradation

Graceful degradation is a natural by-product of systems using stand-by spares and of spare-pooling since the spares normally can be performing tasks which they should drop to take over the work of the failed unit. Graceful degradation is bound to the notion of parallel programming: it requires that the functionality can be split into relatively independent sections, some of which can be dropped when necessary. Graceful degradation thus requires a complex reconfiguration phase which may be incompatible with real-time requirements.

3.5 A scenario of fault-tolerance

We now describe the sequence of operations involved in a typical fault- tolerant computing system. A general scenario appears in Figure 3-12:

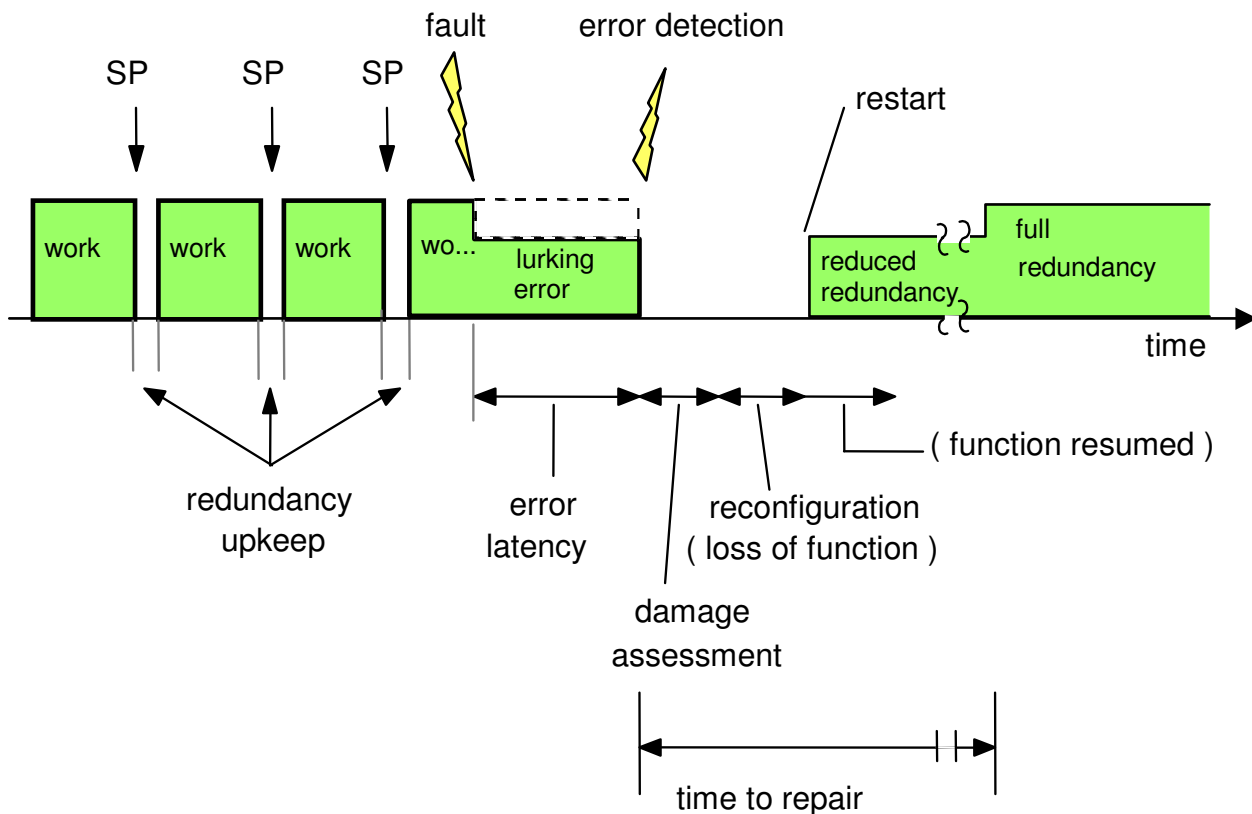


Fig. 3-12: Phases during the Reconfiguration.

These phases are:

Redundancy upkeep

Error detection and damage assessment

Reconfiguration

Repair

Teaching and reintegration

3.5.1 Redundancy Upkeep

During normal operation, the spares must be continuously kept actualised. This is done in work-by systems by synchronizing the concurrent execution and in stand-by systems by periodical state saving to a stable storage. Synchronization or state saving takes place at determined points in the execution and costs computing power. This is symbolized by the SP slots in Figure 3-12.

3.5.2 Error Detection and Damage Assessment

When a fault occurs, it can remain undetected for a relatively long time, until it is detected as an error. The signalling of the error triggers the closing of the corresponding confinement zone to prevent error propagation. Some implementations like the iAPX 432 cannot guarantee that the error will not leak out, but rely on an error reporting bus to catch up the error before it has any harmful consequences.

The error signal starts the damage assessment phase, during which the extent of the error is determined. Often, the damage assessment phase is delayed to allow for transient faults to pass by, since the recovery would otherwise be disturbed. Although the fault could be permanent, one does not know it at that time.

3.5.3 Reconfiguration

If the damage assessment indicates that the working unit has failed, this unit is phased out, and the spare is installed in its place to initiate recovery.

The reconfiguration strategy differs, depending on the degree of readiness of the spare:

- Masking systems and voting systems work with the spare already inserted. They select the most plausible output value and do not reconfigure as long as sufficient redundancy is present. Some masking systems rely on adaptive voting and configure out the faulty unit to increase reliability. Essentially, the duration of the reconfiguration phase for masking and voting systems is zero.
- In hot stand-by, the switchover operation is reduced to the flipping of a switch. The error can be masked if the operation of the switch is fast enough. Data that were transmitted at the moment of switchover could have been corrupted and may require retransmission.
- In warm and cold stand-by, for backward recovery techniques, a rollback phase is inserted during which the spare is loaded from safe storage with the previous good state. The rollback phase is followed by a roll-ahead phase during which the execution is repeated. These operations require from several seconds to some hours depending on the extent of the damage. - Forward error recovery requires a damage correction phase, selective reloading and undoing of possible faulty operations. The reconfiguration time depends heavily on the application.

3.5.4 Repair

With the return to operation, recovery is complete, but the system is not yet restored to its previous state. It is now in a reduced redundancy state. If the system uses dual sparing, there is no redundancy left and a new fault will bring the system down. To return to full reliability, there are several alternatives:

The simplest case only considers transient errors. The faulty unit can be declared good after extensive checking and reloaded with the correct information to serve as spare again;

If the error is persistent, then a spare unit from a pool can be taught to serve as a spare. If spares are not replaced, mission failure can result from spare exhaustion.

The failed unit may be on-line repaired by a maintenance team;

The failed unit may be off-line, repaired by a maintenance team at the next scheduled maintenance. This is the case for spare pooling.

3.5.5 Teaching and Reintegration

An aspect that has been often underestimated is the teaching and reintegration operation of a repaired unit. If the working unit(s) can be stopped for a while for reintegration of a repaired unit (at some expense of availability) then the problem is reduced to a dump from one memory to the other. If reintegration must take place while the working unit is running, i.e. on-line, the loading of the spare with the correct information involves a catch-up phase in which the spare is actualised by a mixture of loading from the working unit's memory and tracking of its execution to update obsolete information.

Here, the methods of teaching differ somewhat between masking systems with replicated computing and stand-by systems with state-saving. Teaching of a spare is a natural operation for recovery while it is an exceptional situation for masking.

The last step is the reintegration of the formerly failed unit as a spare into the system. The system has returned to the same reliability level as existed before the fault.

3.6 References

- [Avizienis 84] A. Avizienis,
"Fault Tolerance by Design Diversity: Concepts and Experiments",
IEEE Computer, Vol. 17, No. 8, pp. 67-80, August 1984
- [Avizienis 78] A. Avizienis,
"Fault Tolerance: the Survival Attribute of Digital Systems",
Proceedings of the IEEE, Vol. 66, No. 10, pp. 1109-1125, October 1978
- [Carter 72] W.C. Carter, A.B. Wadia, D.C. Jessep Jr ,
"Computer Error Control by Testable Morphic Boolean Functions",
FTCS-8, 8th International Conference on Fault-Tolerant Computing, Toulouse , June 1978

- [Chen 78] L.Chen and A. Avizienis,
"N-version programming: a Fault-Tolerance Approach to Reliability of Software Operation",
FTCS-8, 8th International Conference on Fault-Tolerant Computing, Toulouse, June 1978
- [Duke 72] K.A. Duke,
"Detect Errors in complex logic",
Electronic Design 21, pp 88 ff October 1972
- [Laprie 85] Laprie, J.C.,
"On Computer System Dependability: faults, errors and failures",
Proceedings of the IEEE COMPCON, Silver Spring, 1985
- [Intel 82] Intel Corporation,
"iAPX 432 Interconnect Architecture Reference Manual",
Order No. 172487-001 Intel Corporation, 3065 Bowers Avenue, Santa Clara, California 95051 1982
- [Fitch 84] D.J. Fitch, A.M. Guercio, K.W. Johnson, G.T. Surratt,
"DMERT: A Fault Tolerant Environment for Diverse Applications",
FTSC-14, 14th Fault-Tolerant Computing Symposium, Orlando, Florida, pp. 336-340, June 1984
- [Freiburghouse 82] R. Freiburghouse,
"Making Processing Fail-Safe",
Mini-Micro Systems May 1982
- [Goldberg 84] J. Goldberg, W.H. Kautz, K.N. Levitt, R.L. Schwartz, M.W. Green, L.B. Lamport, P.M. Melliar-Smith, C.B. Weinstock,
"Development and Analysis of the Software Implemented Fault-Tolerance (SIFT) Computer",
NASA Contractor Report 172146 SRI International, Menlo Park, California 94025 1984
- [Hopkins 78] A.L. Hopkins Jr., T.B. Smith III, and J.L. Lala,
"FTMP-A Highly Reliable Fault-Tolerant Multiprocessor for Aircraft",
Proceedings of the IEEE, Vol 66, No 10, pp 1221-1239, October 1978
- [Meraud 79] C. Meraud, F. Browaeys,
"A new line of Ultra-Reliable, Reconfigurable Computers for Airborne Aerospace Applications",
AGARD - Symposium on advances in guidance and control systems Ottawa, pp. 62-1 ff, May 1979
- [Ossfeld 80] B. Ossfeld & I. Jonsson,
"Recovery and Diagnostics in the Central Control of the AXE Switching System",
IEEE Transactions on Computers, Vol. C-29, No. 6, pp. 482-491, June 1980
- [Katsuki 78] D. Katsuki, E.S. Elsam, W.F. Mann, E.S. Roberts, J.G. Robinson, F.S. Skowronski and E.W. Wolf,
"Pluribus - An operational Fault-Tolerant Multiprocessor",
Proceedings of the IEEE, Vol 66, No 10, pp. 1146-1159, October 1978
- [Schmid 84] H. Schmid, J. Lam, R. Naro, K. Weir,
"Critical Issues in the Design of a Reconfigurable Control Computer",
FTSC-14, 14th Fault-Tolerant Computing Symposium, Orlando, pp. 36 ff, June 1984
- [Schwartz 83] R.L. Schwartz, P.M. Melliar-Smith, F.H. Vogt & D.A. Plaisted,
"An Interval Logic for Higher-Level Temporal Reasoning",
SRI International NASA Contractor Report 172262
- [Toy 78] W.N. Toy,
"Fault-Tolerant Design of Local ESS Processors",
Proceedings of the IEEE, Vol 66, No 10, pp. 1126-1145, October 1978
- [Wensley 78] J.H. Wensley, L. Lamport, J. Goldberg, M.W. Green, K.N. Levitt, P.M. Melliar-Smith, R.E. Shostak and C.B. Weinstock,
"SIFT: The Design and Analysis of a Fault-Tolerant Computer for Aircraft Control",
Proceedings of the IEEE, Vol 66, No 10, pp. 1240-1255, October 1978

4 Encoding and workby

Encoding and workby are two techniques that rely on the parallel operation of redundant hardware for the sake of the same operation. Both techniques allow detecting and overcoming errors. For this reason, they are treated in the same Chapter.

4.1 Masking

Masking techniques intend to prevent faults or their consequences from leaving a faulty device and leaking to the outer world. Masking provides full integrity and punctuality. As long as sufficient redundancy remains, the faulty case is concealed from the external observer. Masking relies on replicated computation, in which several units cooperate to perform the same work simultaneously. This operation mode is vital for massive redundant, voting and hot-sparing systems. The same synchronisation techniques are used for error detection by duplication and comparison, which is the base for fail-stop systems, as we have seen.

Two basic masking techniques exist:

- **coding redundancy** is a branch of the error detection techniques. It complements the information path and storage elements by redundant bits. It is an efficient technique that requires little redundant hardware.
- **workby** lets a number of synchronized computing elements work in parallel on behalf of the same operation. The redundant units may be loosely or closely synchronized. The units are matched, so they all should be in the same state at the same time. A choice is made among the outputs of the redundant units by a device, which may be a switch, a voter or a selector. We use the generic term of voter for the selecting device. A voter may itself be replicated. Workby requires at least a duplication of the hardware, plus additional elements for the voting. The number of redundant units may be quite large (≥ 3) and one speaks then of **massive redundancy**.

Classical forms of workby are: massive redundancy, 2/3 voting (**TMR**), 2/4 (**quadding**) and 1/2 self-check (**dual self-check**).

The section 4.2 treats the coding techniques in detail. Section 4.3 explains the workby techniques. The last section 4.4 focuses on the techniques for synchronisation and matching for workby operation.

4.2 Error detection and correction by coding

4.2.1 Coding Redundancy

Coding serves the dual purpose of error detection and of error correction. The basic principle is shown in Figure 4-1:

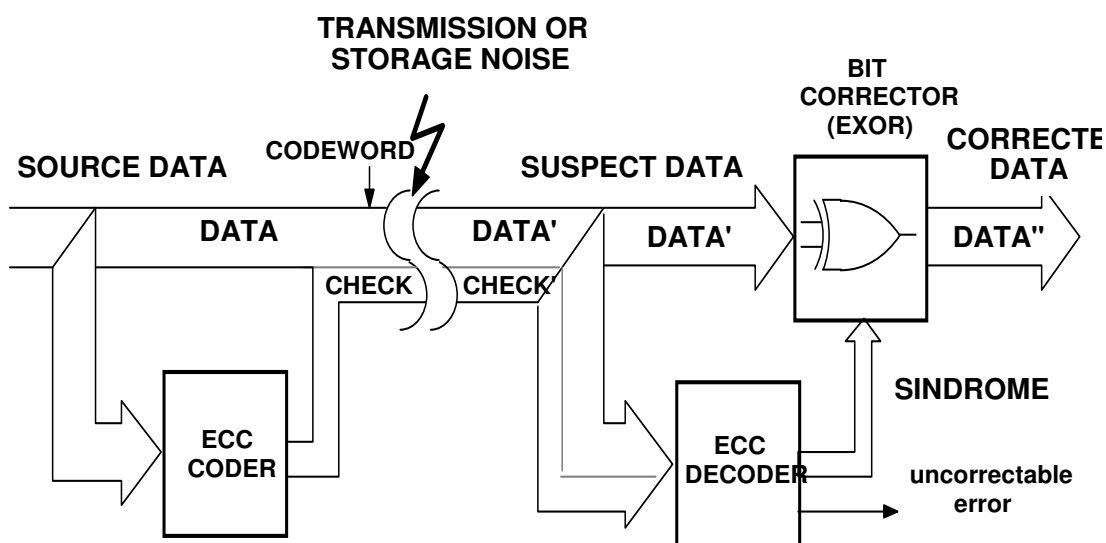


Fig 4-1: Data Path for Error Detection and Correction

Redundant **check information** is deduced from the original, **useful data** by an ECC-coder. The check bits are normally less numerous than the useful data bits. The data and the check bits, which form together a **codeword**, are transmitted, respectively stored in a medium subject to noise. When the codeword is read, a certain number of bits may have been inverted by some noise or failure. The original data is reconstituted from the (possibly modified) codeword by an **ECC** decoder. A circuit that inverts faulty bits corrects erroneous data.

Some simple codes only perform error detection. Error correction is then done by some other mean, like retransmission. In that case the correction circuit is missing, but this case will be left to the next Chapters.

Since the error detection/correction algorithm is applied whether errors are present or not, the correction of an error within a device is hidden from the outside world. This makes coding a suitable technique for masking errors.

The largest application of Error Correcting Codes is found in increasing the reliability of semiconductor memories, which suffer from spurious bit inversions. Indeed, the reliability increase is appreciable: in a 32-bit memory extended by a 7-bit Hamming Code for SEC-DED (single error correction, double error detection), the reliability improvement factor (RIF) for a mission of 1000 hours is 43 [Levine 76].

Note : the reliability improvement factor or RIF is the ratio between the unreliabilities before and after the application of fault-tolerance (see Chapter 9).

Coding schemes are also applied to information transmission, although one is more normally interested in detecting errors than in correcting them (correction can be done by retransmission). In cases where retransmission is too costly or impossible (space probes, long transmission delays), error correcting codes are also used.

Coding schemes are also used to mask errors in computing elements. But here also, one is more interested in detecting errors than in correcting them.

We will explain briefly the fundamental coding schemes.

4.2.2 Parity

The best-known coding scheme for error detection is **parity**. Parity is not an error correcting code: it only allows detecting errors. However, suitable extensions of the parity scheme can also correct errors. Parity consists in appending a **parity bit** to a **data word** of k bits which is transmitted as $(k+1)$ -th bit to form a $(k+1)$ -bit **codeword**. The parity bit tells whether the number of "1"s in a codeword, i.e. its **hamming weight**, is odd or even.

We define that an **odd parity bit** is chosen such that the Hamming Weight of the resulting codeword (including the parity bit) be odd, i.e. the parity bit has the value "1" if the number of "1s" in the symbol (excluding the parity bit) is even and a "0" if that number is odd. Similarly, an **even parity bit** sets the number of ones in the codeword (including the parity bit) to an even number. The even parity bit is calculated as the exclusive OR (XOR) value of all symbol bits. Figure 4-2 shows the building of even parity for an 8-bit message word.

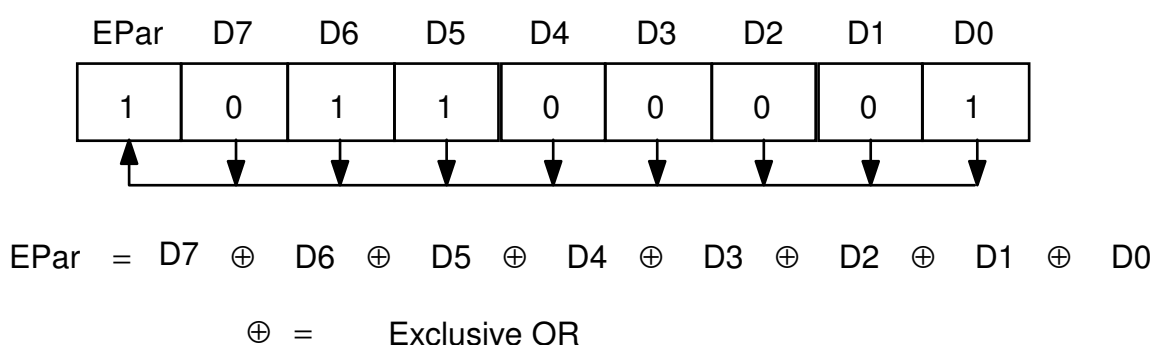


Fig. 4-2: Even parity (HW = 4).

Note: Unfortunately, the reverse definition of parity appears in the datasheets of some integrated circuits: "odd" is taken in the sense there that the sum of the "1"s excluding the parity bit is odd, which is just the opposite of the commonly accepted definition.

Note that the parity bit is generally considered as being the bit with the highest weight. This is because serial transmissions usually transmit the least significant bit first, and parity is then appended to the message.

In general, the parity kind is chosen so as to force a difference between transmission and the lack of transmission. If a transmission is binary, i.e. if it has only two distinct levels, the quiescent level (passive level) cannot be

distinguished from one of the logical levels. Then the parity is chosen so that at least one bit is active when the dataword consists only of passive levels.

Example:

if the passive level is "0", the parity should be odd. If the passive level is "1", the parity should be chosen as even if the number of bits in the dataword is even or as odd if the number of bits is odd.

The parity bit does not provide 100% error detection. In fact, if any 2 bits in the codeword containing the data and the parity bit are inverted, the parity is again correct, although a double error occurred. Therefore, the Hamming Distance of parity is only 2. Obviously, it is not reasonable to protect a 512-block file with a single parity bit. The length of the word that a parity bit protects effectively depends on the acceptable residual error rate.

The residual error rate R_{er} can be calculated as follows: Let E_r be the probability that one bit is in error. The probability that a particular bit is not in error is $(1-E_r)$. The probability that any of the n bits is in error is equal to $E_r n$, and that no bit is in error is $(1-E_r)^n$.

The probability that a given bit is in error and no other is $(1-E_r)^{n-1} \cdot E_r$ and, since there are n bits, the probability that exactly one bit is in error is $n \cdot (1-E_r)^{n-1} \cdot E_r$. Therefore, the probability that the codeword has either no error or exactly one error is:

$$R_{er} = \underbrace{1}_{\text{NOT}} - \underbrace{(1-E_r)^n}_{\text{no error}} - \underbrace{n \cdot E_r \cdot (1-E_r)^{n-1}}_{\text{exactly one error}}$$

This expression can be approximated to $R_{er} = n \cdot (n-1) \cdot E_r^2$ for $E_r \approx 1$.

Example:

The error rate is 10^{-5} ; if the protected word size is 8 bits ($n = 9$), the residual error rate is $72 \cdot 10^{-10}$.

But if the protected data block has a length of 512 bits, $n = 513$ and the R_{er} approximates to $2.6 \cdot 10^{-5}$ which means that the probability that an undetected error takes place in the block is higher than the bit error rate, and makes parity quite useless. Therefore, parity is used to protect small data items, typically 7 or 8 bits wide.

In parallel buses, parity is commonly used. DEC's SBI has one parity bit for all 32 lines of information. IEEE 896, MULTIBUS II, VERSABUS and NUBUS have one parity bit for every 8 information lines.

4.2.3 Longitudinal parity

We consider now the transmission of a sequence of words such as a file. In some transmission schemes it is not possible to transmit a $(k+1)$ -th bit for each word, since the channel is not wide enough. For instance, most asynchronous serial channels allow the transmission of an 8-bit wide word. If one uses the 8th bit as parity, only 7 bits remain as useful information. This is what the ASCII transmission standard recommends. However, if one transmits a binary file consisting of 8-bit words (for instance an object file), the chunking of 8-bit words into 7-bit units is not a good practice. One prefers then to transmit 8-bit words and to protect them additionally by a parity word every M -th word. Then parity is built over all bits belonging to the same column (Figure 4-3):

word 1	0	0	0	1	1	1	0	0
word 2	1	0	1	1	0	1	1	1
word 3	0	0	1	0	0	1	1	0
word 4	0	0	0	0	1	1	0	0
word 5	1	1	0	0	0	1	0	0
word 6	1	1	1	0	1	0	1	0
word 7	0	1	0	0	0	1	0	0
word 8	0	0	0	1	1	1	1	0
parity word	1	1	1	1	1	1	1	0
word 9	x	x	x	x	x	x	x	x

Fig. 4-3: Longitudinal, even parity over 8 bytes.

The parity is built column-wise, hence the name "longitudinal parity". Note that longitudinal parity indicates in which column the error took place. In a parallel transmission that is as wide as the individual words, if a column is repeatedly in error, this suggests damage on a particular line. In a serial transmission, it is unlikely that the same column is repeatedly affected.

4.2.4 Single error correction

Longitudinal parity can be used to correct errors as well, when combined with horizontal parity. If a single error occurred, then one can detect precisely which bit is in error since its position is known in term of row and column. Then this bit can be inverted again and the error corrected.

Hamming (around 1950) [HAM50] extended the principle of error correction found in the combination of longitudinal with horizontal parity to correct single errors. **error correcting codes** (ECC) are often employed to increase the reliability of memories. Error correcting codes are not common in parallel buses although they become attractive in 32-bit or wider buses. The biggest advantage of error correcting codes is that the overhead of retransmission can be avoided.

The method used by Hamming for **single error correction** (SEC) is quite simple. To a data word of k bits, r Hamming check bits are concatenated to form a n -bit codeword. Let's suppose, to see how it works, that a method exists for correcting errors in that codeword. By applying a proper algorithm to this code word, we should be able to generate an error report, called a **syndrome**, which indicates precisely the position of the erroneous bit so that we can invert it (we expect only one error to be present at a time). If, for instance, the syndrome length is 4 bits, we can check with it a codeword that is $16 - 1 = 15$ bits in length. The missing combination is needed to tell that there is no error.

Now, let us put down this 15-bit codeword $W_1..W_{15}$, without caring which bits are data and which are check bits. Below each position 1..15 we write the binary value of the syndrome that should be generated in case of error in that position (Figure 4-4).

codeword :		1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Syndrome bits	S3	1	1	1	1	1	1	1	1
	S2	.	.	1	.	1	1	1	1	1	1	1
	S1	.	.	1	1	.	.	1	1	.	.	1	1	.	.	1
	S0	.	1	.	1	.	1	.	1	.	1	.	1	.	1	1

↑
↑
 no error position 6

Fig. 4-4: Syndrome S0..S3 and bit positions.

The zero position corresponds to "no error". To make things more apparent, the zeros have been changed to dots. Consider, for instance, that after the building of the syndrome, we find $S = 0110$. This means that the bit in position 6 is incorrect and should be inverted.

Conversely, if bit 6 of the codeword is in error, then this should produce a "1" in the syndrome bits S1 and S2 and a "0" in S0 and S3. Now let us see how to generate such a syndrome.

Consider the last row (S0) in Figure 4-4. Suppose that the sender uses the check bits of the codeword as parity bits in such a way that the parity over all bits of the codeword which have a "1" in the first syndrome row is odd. That is, if we build the parity over all odd positions W1, W3, W5, W7...W15, S0 will be zero. Obviously, one of these positions contains a parity bit, we will see which one later.

At the destination, we can recompute the parity over the odd positions and it should still be zero if no error occurred in them. But if the computed parity is one, we know that one of the bits in the odd positions is in error, but we do not know which one. If an even location is wrong, this will not affect S0.

Similarly, let us arrange that the parity over all bits which have a "1" in the third syndrome row S1 (W2, W3, W6, W7, W10, W11, W14, W15) be even. We do the same for all bit positions which have a "1" in the second row S2 (W4..W7, W12..W15) and for all bit positions which have a "1" in the upper row (W8..W15).

Consider that if bit 6 in Figure 4-4 is in error, then this will affect the parity of the second and third row, S1 and S2, but not S0 and S3. Therefore, the vector S0..S3 is 0110 and points to position 6 where the error took place.

As explained above, there must be in the codeword one parity bit for each row that completes the parity of its row to "0". These parity bits are called the HAMMING BITS H3..H0. There are obvious positions for the Hamming bits: every time a Hamming bit is in error, it should point to its own position. Therefore, the positions for the parity bits is where there is only one "1" in the column, that is, at positions 1, 2, 4, and 8. The rest of the codeword is filled with the data bits, as shown by Figure 4-5:

Codeword position	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
	H0	H1	D0	H2	D1	D2	D3	H3	D4	D5	D6	D7	D8	D9	D10
S3	1	1	1	1	1	1	1	1
S2	.	.	.	1	1	1	1	1	1	1	1
S1	.	1	1	.	.	1	1	.	.	1	1	.	.	1	1
S0	1	.	1	.	1	.	1	.	1	.	1	.	1	.	1

H0 = parity over D0, D1,D3, D4, D6, D8, D10
 H1 = parity over D0, D2,D3, D5, D6, D9, D10
 H2 = parity over D1, D2,D3, D7, D8, D9, D10
 H3 = parity over D4, D5,D6, D7, D8, D9, D10

D0 data bit
H0 hamming code bit

Fig. 4-5: Single Error Correction.

When the information is received, the receiver builds the parities over all codeword bits including the Hamming bits. The resulting values of S0..S4 yield as a syndrome the position of the faulty bit. The syndrome is then fed to a logic circuit that inverts the faulty bit. This logic consists simply of an XOR gate placed in each data line, which complements the corresponding bit, and an address decoder, which selects one of the XOR gates, as will be shown in Figure 4-9.

It is quite easy to find how many Hamming bits are required for a given data length. A syndrome of length r points to 2^r positions, one of which is needed to signal that there is no error, that is, the codeword has a length of $2^r - 1$ bits, of which r bits are Hamming bits. Therefore, the number of data bits is $2^r - 1 - r$.

Each Hamming Code is identified by the relation of code bits to data bits, as is shown in table 4-6:

Check Bits	Codeword Bits	Data Bits	Hamming Code Name
r	$n = 2^r - 1$	$k = n - r$	(n, k)
3	7	4	(7, 4)
4	15	11	(15, 11)
5	31	26	(31, 26)
6	63	57	(63, 57)
7	127	120	(127, 120)
8	255	247	(255, 247)

Table 4-6: Hamming Codes for SEC.

The corresponding codes are known as **hamming (n, k) codes**. Note that the efficiency of the code, i.e. the relation data bits/code bits, increases with the codeword length. Therefore, it is more efficient to protect a long word than a short one, but the limit is given by the acceptable residual error rate, like for parity.

The length of the data field (11, 26, 57, etc.) is unusual in the computer world, where word sizes are multiples of 4 or 8. This means that, in practice, some data positions are not used. If there is only an 8-bit data word to protect, then we can ignore the three positions D8..D10. If some positions are not exploited, we are able not only to correct all single errors, but also to detect some double errors: those which yield syndromes pointing to the unused positions. Except for this case, a double error will fool SEC, since it lets the logic correct the wrong bit.

Alternatively, the SEC Hamming codes can also be used to detect all double errors, but then they cannot correct them. Every time a syndrome appears which is different from 0, a single or double error occurred. Therefore, one should be careful since some manufacturers advertise their circuits for "single error correction, double error detection" (SEC/DED), meaning that they can do either, but not both. The following section will show how both SEC and DED can be made.

4.2.5 Single Error Correction, Double Error Detection

SEC codes can be used for **single error correction and double error detection** (SEC+DED) at the expense of one additional check bit. With the previous SEC scheme a double error manifests itself as a seemingly correct syndrome which points to the wrong position.

Suppose that a first error is present, and the syndrome points (correctly) to location 6 (0110). A second error would change the syndrome into any of the following just by changing one bit: 1110, 0010, 0100, 0111. Note now, that an error that inverts one bit changes the number of "1"s in the word, which is its Hamming Weight. If the weight was even, an error makes it odd, if it was odd, it is made even.

So we now make the additional restriction on syndromes, that they all have the same Hamming Weight. A second error would then change the HW of the syndrome and this would be detected as a second (uncorrectable) error. To do so, we must sacrifice all bit positions that have an even Hamming Weight, as Figure 4-7 shows:

Codeword position	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
	H0	H1		H2			D0	H3			D1		D2	D3	
S3	1	1	1	1	1	1	1	1
S2	.	.	.	1	1	1	1	1	1	1	1
S1	.	1	1	.	.	1	1	.	.	1	1	.	.	1	1
S0	1	.	1	.	1	.	1	.	1	.	1	.	1	.	1

H0 = parity over D0, D1, D2
H1 = parity over D0, D1, D3
H2 = parity over D0, D2, D3
H3 = parity over D1, D2, D3

D0

data bit

H0

hamming code bit

unused

Fig. 4-7: Valid syndromes for SEC+DED.

Only four data bit positions remain. Of course, we choose to remove the bit positions that have an even weight so as not to remove the Hamming bits.

If the syndrome is now for instance 11 (binary: 1011), changing one of its four bits will yield one of the invalid bit positions (3, 9, 15 or 10), which will be flagged as a double error. However, a third error can fool SEC+DED. This means that 4 check bits are required to perform SEC+DED on a data word of 4 bits, and in general, the relation holds that $2r \geq 2 \cdot n$.

We need therefore 3 check bits to protect one bit, 4 check bits for 4 bits, and so one as table 4-8 shows:

Check Bits	Codeword Bits	Data Bits	Hamming Code Name
r	$n = 2^{r-1}$	$k = n - r$	(n, k)
3	4	1	(4, 1)
4	8	4	(8, 4)
5	16	11	(16, 11)
6	32	26	(32, 26)
7	64	57	(64, 57)
8	128	120	(128, 120)

Fig. 4-8: Hamming Codes for SEC+DED.

As for the above case of SEC, the data size of these codes is unusual in the computer world. Therefore, some positions remain unassigned. For instance, 5 bits are required to protect an 8-bit word with 3 unassigned positions, 6 bits are required for 16 bit words with 10 unassigned positions and 7 bits for 32-bit words with 25 unassigned positions. Most of the subtleties of the hardware design apply to the correct choice of unused combinations so as to minimize the logic.

Figure 4-9 shows a complete decoder circuit for SEC+DED built with standard TTL circuits [ALT79]. The data lines (D1..D16) and the Hamming bits (H0..H5) are applied at the input to the parity generators which build the syndrome bits S0..S5. The syndrome is decoded by the '138 to select one of the 16 data inverters which form the correction circuit. Double errors are detected by building the parity over the syndrome word, which is even when there is no error or a double error.

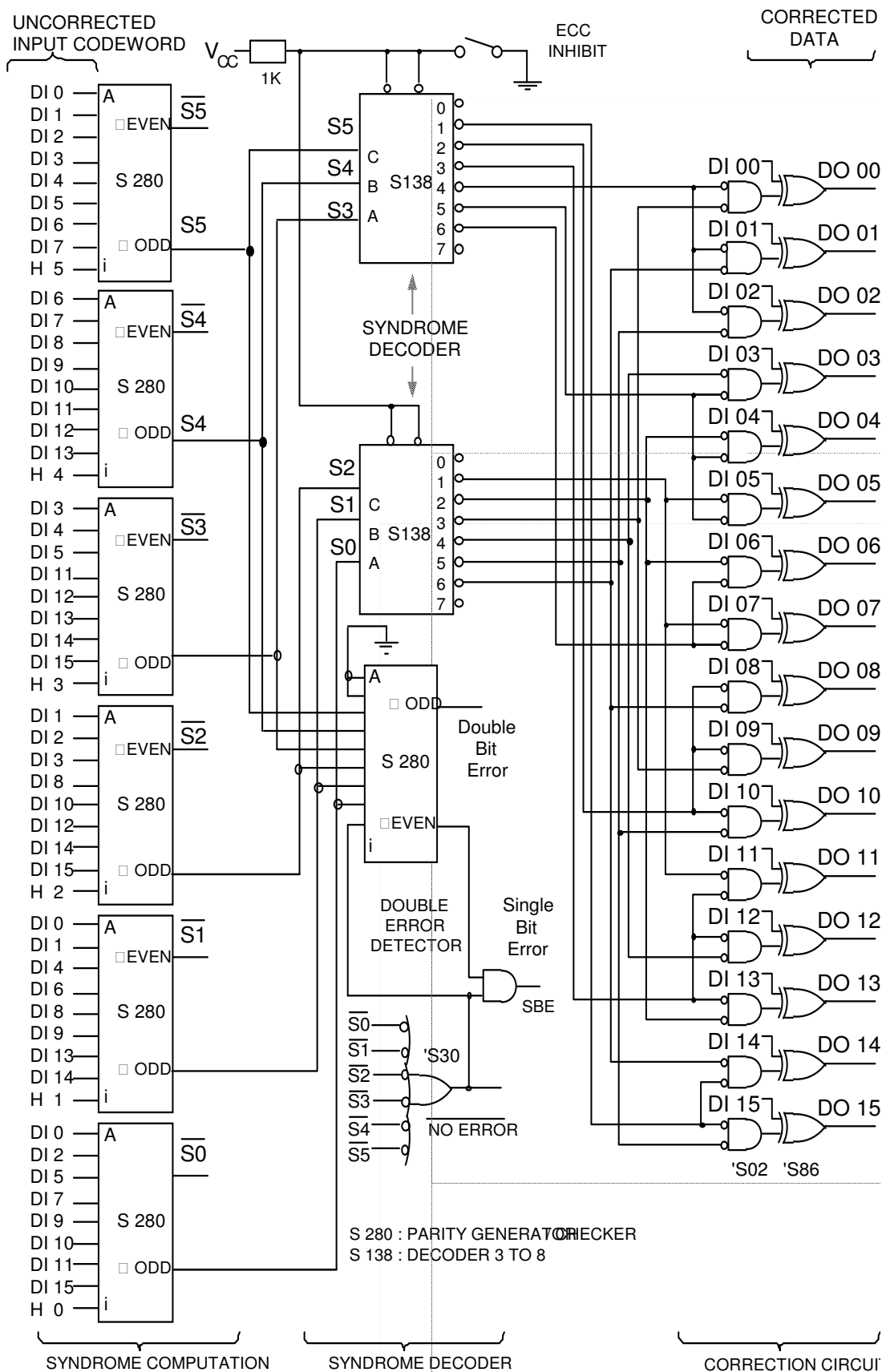


Fig. 4-9: SEC+DED circuit with integrated circuits [Alt 79].

There are numerous integrated memory controllers in the marketplace which offer at the same time coding and refresh functions for dynamic memories.

Figure 4-9 shows the syndrome-building matrix of the INTEL memory controller for SEC+DED on 16 bits [ALT79].

Bit	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22
Position :	H0	H1	H2	D0	H3	D1	D2	D3	H4	D4	D5	D6	D7	D8	D9	H6	D10	D11	D12	D13	D14	D15
	1	2	4	7	8	11	13	14	16	19	21	22	25	26	28	32	35	37	38	41	42	44
S0 .	1	.	.	1	.	1	1	.	.	1	1	.	1	.	.	.	1	1	.	1	.	.
S1 .	.	1	.	1	.	1	.	.	.	1	.	1	.	1	.	.	1	.	1	.	1	.
S2 .	.	.	1	1	.	.	1	1	.	.	1	1	.	.	1	.	.	1	1	.	.	1
S3	1	1	1	1	1	1	1	1	1	1
S4	1	1	1	1	1	1	1
S5	1	1	1	1	1	1	1

Fig. 4-10: Syndrome Generator Table for a 16-bit Data Word.

Note that positions 31 (011111), 47 (1001111), 55, 59, 61, and 62 are missing in Figure 4-10. These positions have been dropped since their Hamming Weight is 5 and consequently these bits would need to be routed to 5 different parity generators instead of 3 for the other combinations. If only weight 1 (check bits) and weight 3 are considered, then a 20-bit word can be protected. The unassigned combinations are 49, 50, 52 and 56.

The manufacturers differ in which other combinations they drop to reduce the number of data bits to 16. Some, for instance, drop positions 21, 22, 41 and 42 and use combinations 49, 50, 52 and 56 instead. This is what the manufacturers call "modified Hamming Codes", and great care must be taken when interfacing the chips of different manufacturers that they have the same syndrome generator. Such errors tend not to appear to the designer, since most of the time, the circuit works error-free.

Since a double error cannot anymore be corrected, measures are taken to correct every single error as early as possible. The classical method is scrubbing, i.e. the cells are read individually during the normal refresh and corrected if necessary. Furthermore, every read operation must also be a read-modify-write.

It is not sufficient just to detect and correct data errors. A read or a write of a correct data to an incorrect memory location is just as dangerous as a write of incorrect data to the correct address. The solution is to store an address code along with the data. Therefore, write operations must first read the previous data to check the address code before writing the new one into it. This causes an interesting problem: since at initialisation time, memory is void, its checksum is incorrect. A first solution is to load the memory with dummy words before the program starts. A better one could just tag the memory as void during initialisation, by using an illegal Hamming bit combination (there are always some of them since 7 Hamming bits cover more than 32 bits). That way, one could detect when variables are read which have never been written: a cheap mechanism to catch software errors.

If the same technique is used consistently in a system, covering processors and buses as well as memories, one can correct any single bit error. More elaborated coding schemes can correct double or triple errors at the expense of an increase in word width.

However, the problem arises when the information is not only stored or transported, but also modified by a processor. Techniques exist to transform a code into another that respects the computation performed on the data. It is today feasible to build a whole processor using coding with only about 30% additional gates [Horninger 85]. Coding is therefore far more economical than duplication.

The problem of coding is the close coupling between the redundant elements. A common mode of failure can affect several bits at the same time and lead to a false correction. Mechanical coupling is also an obstacle to modularisation. It is not possible to remove a faulty part and keep the rest working: so no on-line repair is possible. This is possibly the stricter limitation of coding.

4.2.6 Example of a processor coding system: Philips's (4,2)

It is relatively difficult to extend the Hamming correction scheme to a complex device as a processor, since information is not only transported and stored, but also modified. Further, the tight coupling of the working and the coding circuits is an obstacle to modularisation. An attempt to overcome this problem has been made by Philip's (4,2) concept [Krol 82], shown in Figure 4-10:

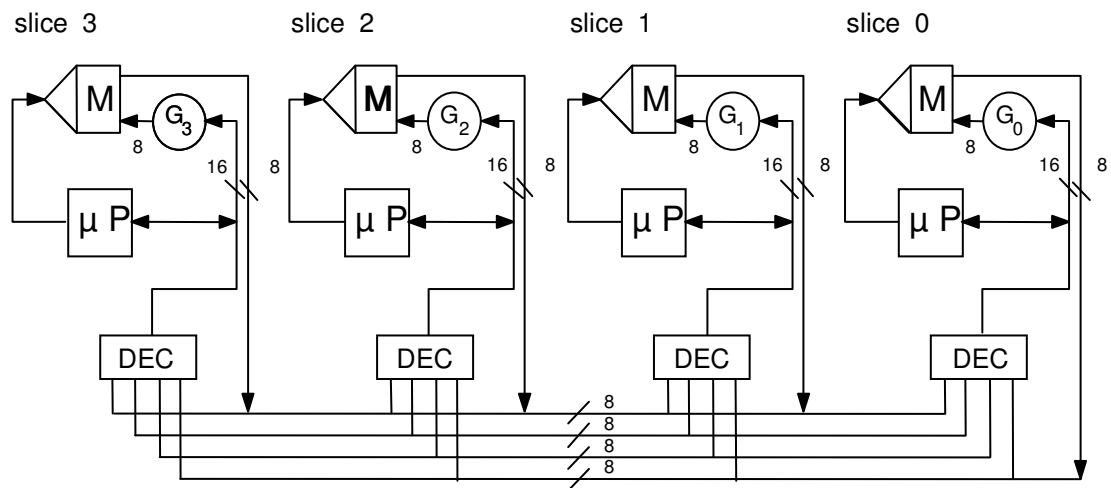


Fig. 4-11: The (4,2) Fault-Tolerant Computer (taken from [Krol 82])

Compared to a non-redundant processor, this concept uses four processors and four memories, each memory being only half as wide as the simplex memory. The processors are synchronized in lock-step. The 16-bit output produced by each processor passes through a code generator that produces an 8-bit code. This code is stored in the 8-bit wide half-memory associated to the processor. The coding is such that the system can tolerate the loss of any of the processor/memory pairs.

4.3 Work-by techniques

4.3.1 Massive Redundancy

The simplest method for fault-tolerance is to let a sufficiently high number of computing units work in parallel. The units should be synchronized so that they all output the same or a similar result at the same time if they are not faulty. The plant then takes a plausibility or majority vote. If the plant process is continuous, the synchronization can be slackened. There is no necessity for error detection, except for maintenance purposes.

Consider a reliable computing system for moving the control surfaces of an aircraft (Figure 4-12):

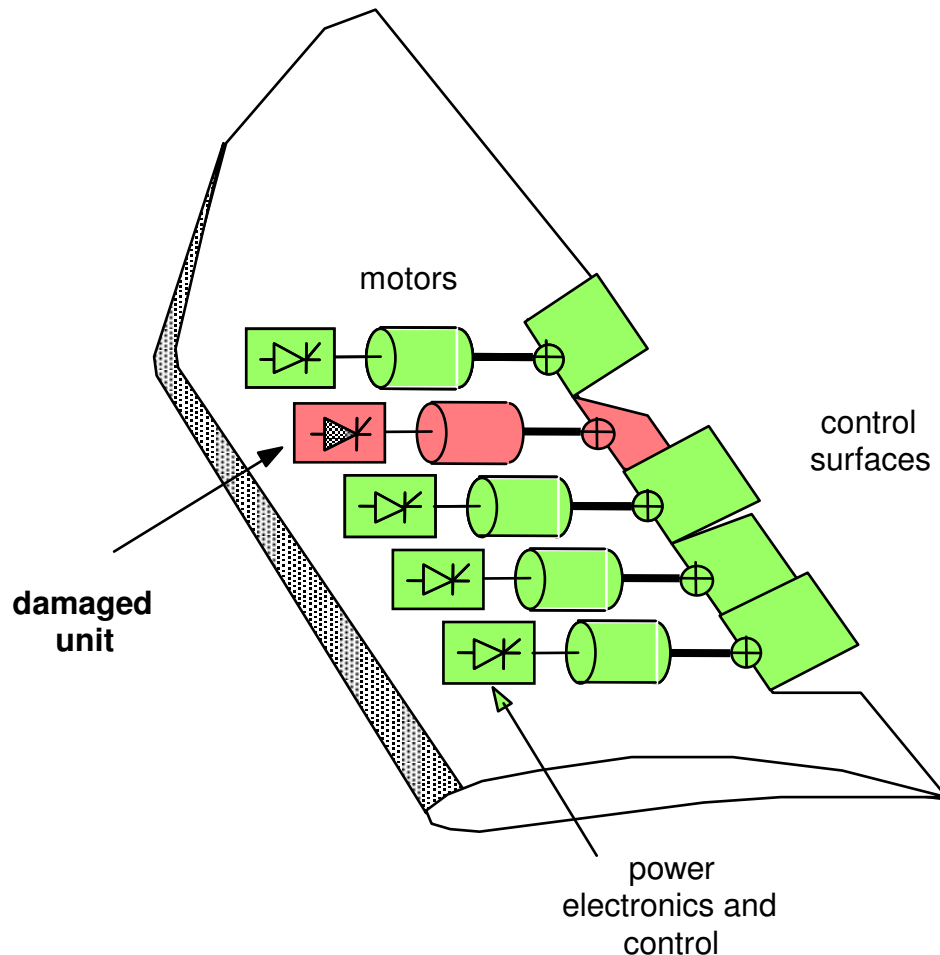


Fig. 4-12: Massive Redundancy.

Five computers in parallel control the wing. Each one is doing in parallel the same work as the other four, responding to the same inputs and (normally) generating the same outputs. Instead of installing a switch to decide which computer should move the control surface, each computer is given a portion of the control surface to move. So, even if one computer fails completely and systematically in the wrong direction, there will be four other computers to compensate for its wrong action. The result will possibly be an increased drag, but the aircraft will be still able to land safely.

Massive redundancy is the method with the highest reliability, but also the highest hardware expenses. It requires at least three units. Further, it is not generally applicable, since only certain kinds of plants have inherent voting possibilities.

In massive redundancy, the computers may be independent and not synchronized, as long as the process is continuous, for instance in the case of a temperature regulator. But as soon as the process becomes non-synchronous, and the program contains branches, discrepancies among the computers may arise because of internal state differences, without any fault involved.

Consider an algorithm that should maintain an aircraft at a height of 10'000 m. Some of the computers may compute a value of 9'990 m, others a value of 10'010 m. A set of computers will move the elevator upwards and the rest move it downwards.

Therefore, massive redundancy requires in most cases that the replicated units be in an identical computing state as long as no error is involved.

4.3.1.1 Example of massive redundancy: the Space Shuttle Flight Computer

The Space Shuttle's computer is an example of massive redundancy. Its function is to guide the Shuttle during launch, ascent and re-entry, to perform the navigation and guidance and to monitor the payload operations on orbit. The Shuttle's computer consists of five work-by processors. Four of the five processors have been developed by IBM and build the PASS (Primary Avionics Software System). The fifth computer, called BFS (Back-up Flight System) contains only the software for the most critical flight phases (ascent and reentry). The BFS can be switched on by the crew upon a failure of the other four as an ultimate measure against a generic software fault. The BFS has been developed by an independent firm (Rockwell) from a subset of the specifications and is an example of diverse programming. Although it participates in the inputs and computations, its outputs are ignored except for test purposes.

The five processors are loosely synchronized (within 150 μ s). Each monitors a separate bus to which sensors and effectors are connected. A mechanical device, which is essentially a steel bar to which the motors driven by each processor are coupled, is the voting element. There is no voting on the results by the processors themselves. Faulty processors are not disconnected when they suffer a failure, since the idea of letting a processor switch off itself or another processor was considered too risky. The only serious situation would be a "two/two split" in which two processors would fail in the same way, but this situation has never been observed (Figure 4-13):

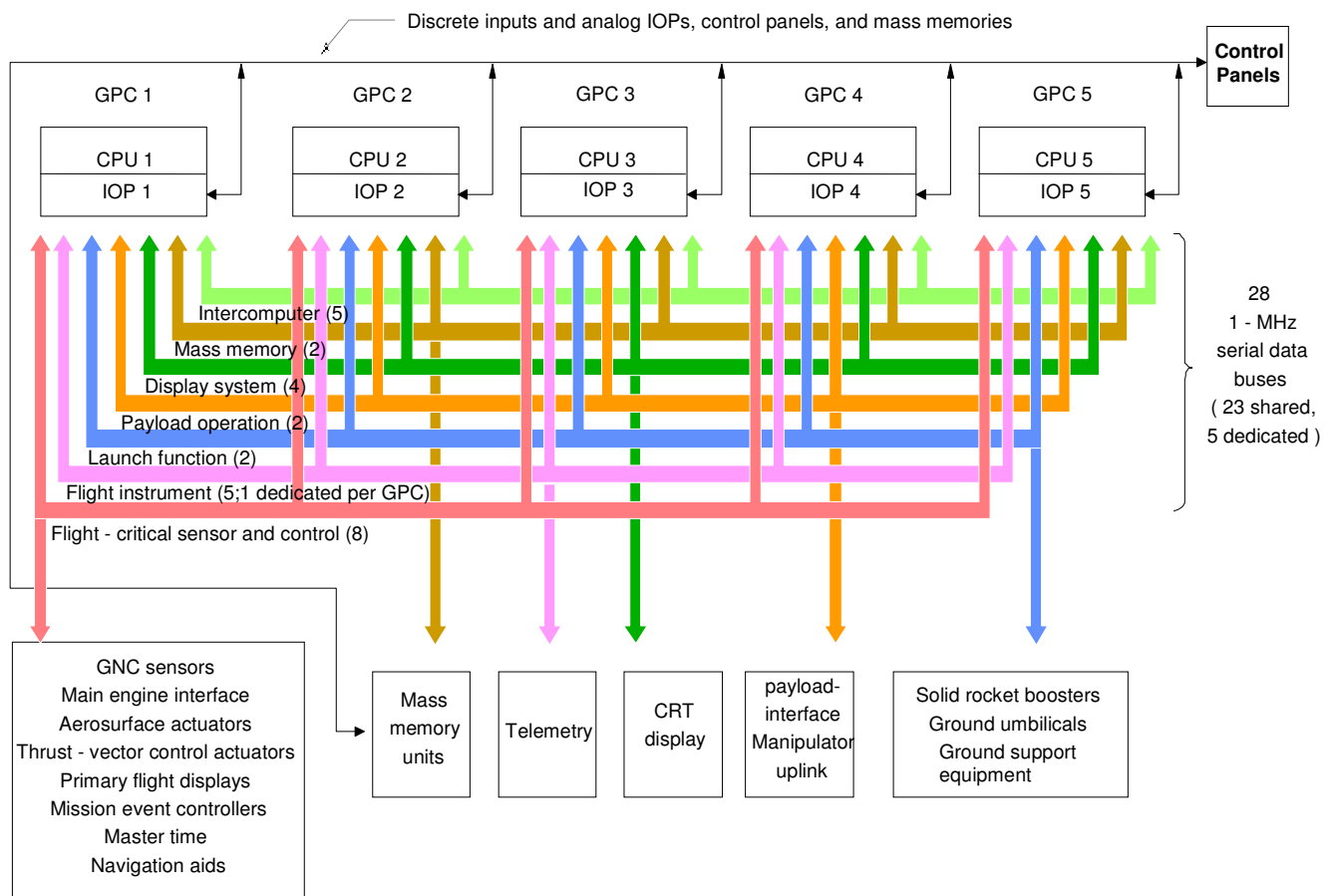


Fig. 4-13: Space Shuttle Computer.

The software and hardware design of the Space Shuttle's computer is discussed in detail in [ACM 84] by several authors. For the coming discussion, it is interesting to know that most of the problems in the Space Shuttle computer were caused by faulty synchronization between the computers, not by actual failures of the units.

4.3.2 Triple Modular Redundancy and NMR

In voting redundancy, the majority vote is taken by a dedicated unit, called the voter rather than by the controlled plant. The result of the voting is output to the plant. The minimum configuration for this mode of operation is 2-out-of-3, or 2oo3 redundancy, also called Triple Modular Redundancy or TMR. This technique can be generalized to N units under the name of N-modular redundancy or NMR. The Space Shuttle's avionics computer is not really an NMR system, since no dedicated unit does vote its outputs.

In NMR, the faulty units are ignored, and there is no necessity to detect errors for continued operation. However, errors are naturally detected by the voter and signalled for maintenance and logging. Therefore, it is also a method for error detection.

TMR is probably the oldest masking technique for computers: it was foreseen by Von Neumann and implemented for the first time back in 1954 on the SAPO computer. This computer operated with vacuum tubes and magnetic drums. Prof. Antonin Svoboda at the Academy of Sciences in Prague, Czechoslovakia developed it.

TMR consists of three identical processors (a triad) that execute the same computations in parallel under control of a common clock. The inputs are synchronized, the outputs are voted upon by a two-out-of-three voter circuit which takes a majority vote (Figure 4-14): The hardware redundancy is higher than 200%, since in addition to the three redundant units, a (safe) voting system must be implemented.

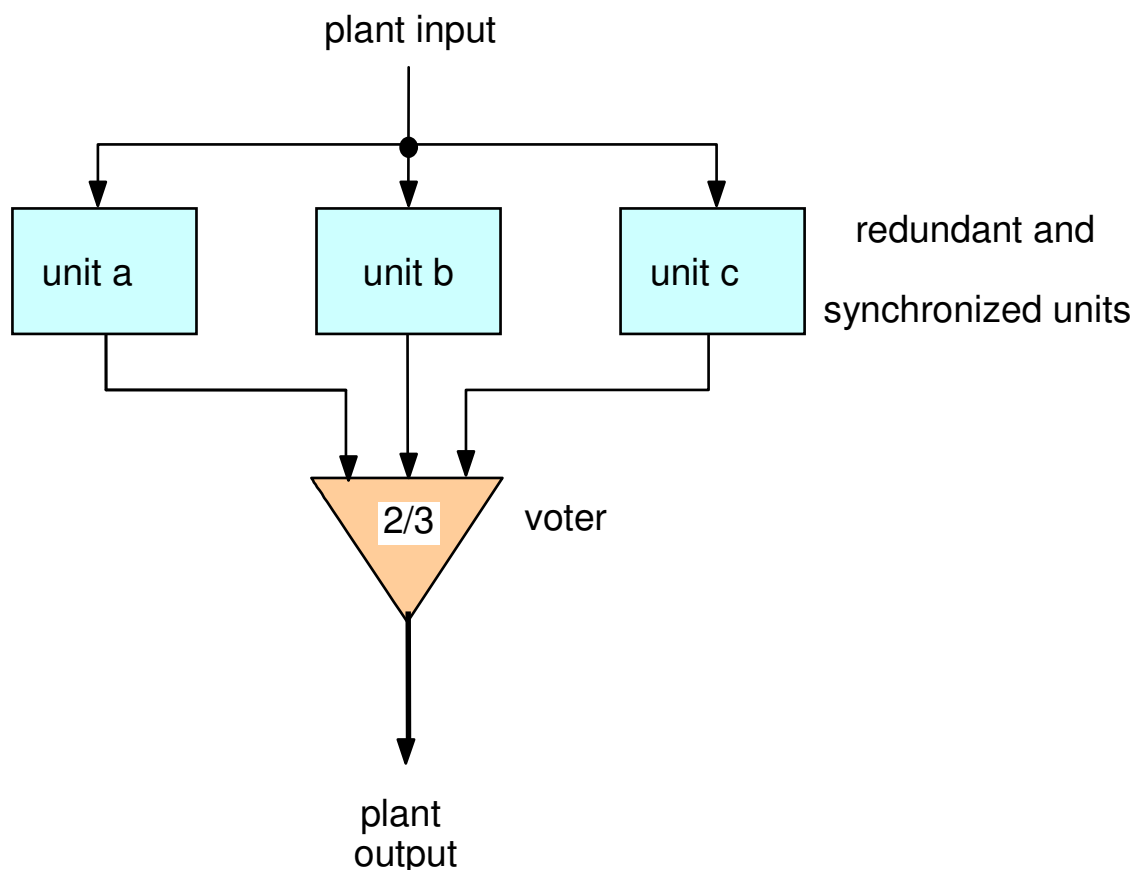


Fig. 4-14: Triple Modular Redundancy.

The voter is a hard-core component: it determines the reliability of the whole system, since a failure of the voter leads to a total outage. To obviate failures of the voter, the voter itself must be triplicated (Figure 4-15).

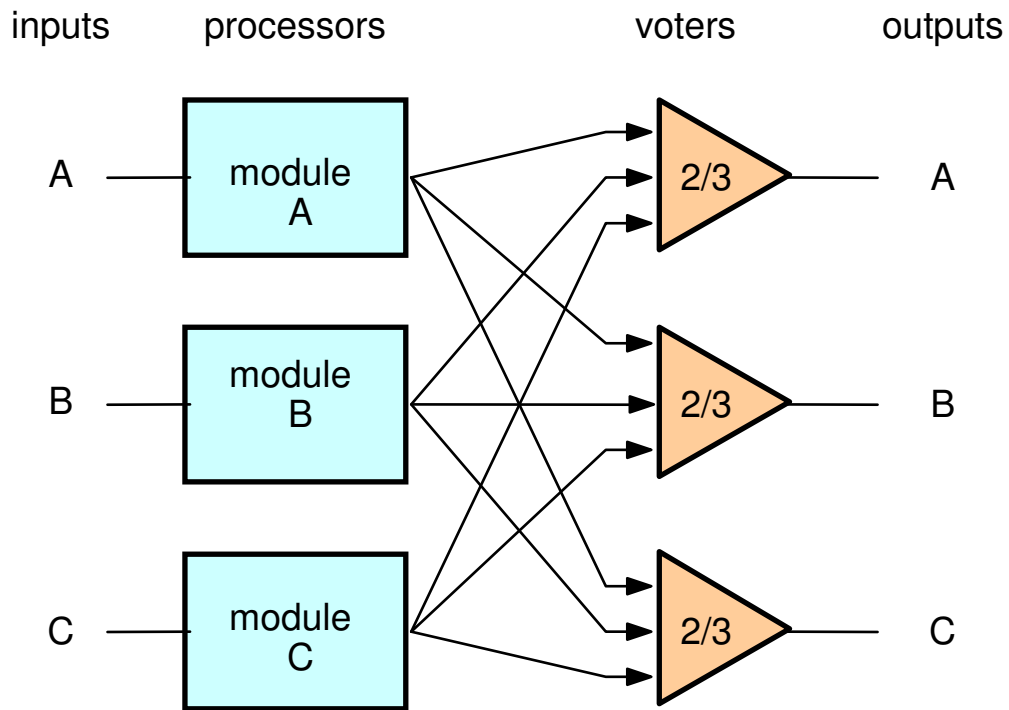


Fig. 4-15: TMR system with triplicated voters.

Of course, the output of the voters must be tied together at some place before the plant, thereby introducing a single point of failure, but in some plants it may be possible to stretch the voting out into the plant, like for massive redundancy.

TMR is based on the assumption that the three computers will be in the same state if no error occurs. This is achieved by letting the three computers execute the same programs at the same time in close synchronism. TMR exists in various flavours: Static TMR and Repairable TMR, which itself subdivides in On-line-Repairable TMR and Spare-Pooling TMR.

In **static TMR**, a first error is overcome and the operation continues. A second error leads to a total outage. Generally, a second error causes the whole TMR system to shut down to give at least a fail-stop behaviour. In some cases, the faulty unit is passivated after a second error to reduce the probability of **complot** (same fault in two units which votes out the correct output).

The MTBEF (Mean Time Between Element Failures) of Static TMR is more than three times higher than the MTTF of the individual processors, as we shall see in Chapter 9. The reliability of TMR, given by its MTTFM (Mean Time To Mission Failure), is quite low when the mission time exceeds the MTTF of the redundant modules, as Figure 4-16 shows.

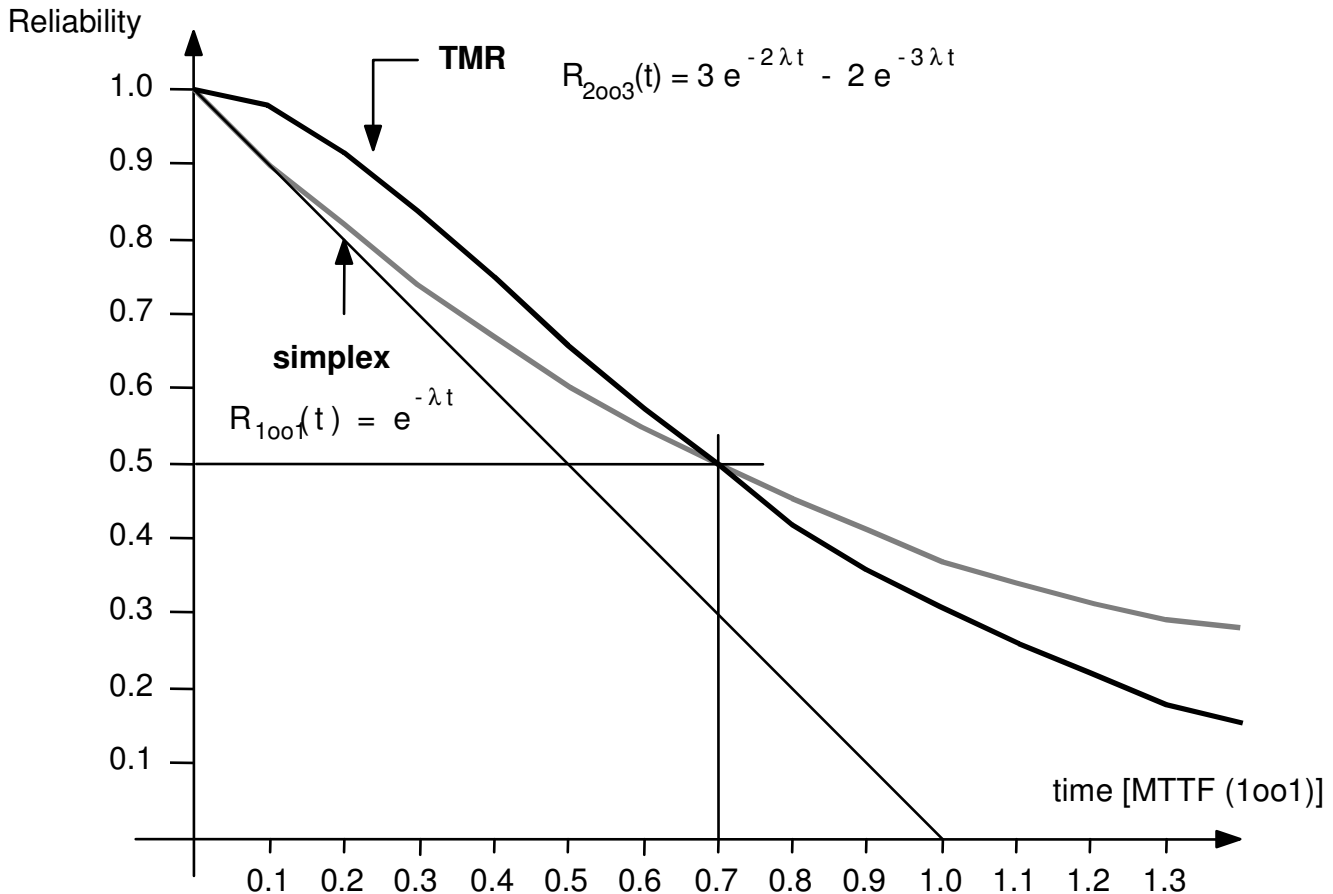


Fig. 4-16: Reliability of TMR in Function of Time.

Therefore, Static TMR is only considered for short mission times. It has been used successfully in the control computer of the Saturn V rocket. It is also used for the control of rail vehicles to increase the availability of fail-stop systems: the first fault can be survived, the second fault causes a fail-stop.

Dynamic TMR is better suited than static TMR for longer missions (long relative to the MTTF of the parts). Dynamic TMR assumes that the failed unit can be replaced and the reliability restored by reconfiguration. There are two ways to achieve this, on-line-repairable TMR and spare-pooling TMR.

- **on-line-repairable TMR** assumes that on-line repair of the failed modules is possible during operation. The maintenance personnel may execute the repair without stopping the computer (or at a point in time where function is not required). A failure only occurs when a second unit fails while the first unit is being repaired.
- **spare-pooling TMR** (also called Hybrid Redundancy) replaces the on-line repair by an automatic insertion of a spare taken from a spare pool. We will discuss two examples in the next subsection.

A particular problem is induced by **transient errors**. If an error leaves the memory of one unit in an incorrect state, this error can remain undetected for a long period of time, since only the outputs, not the internal states are compared. Now, if another error occurs in another replicated unit, it may combine with the first one and the voter will be confronted with three different outputs. A fail-stop system would then stop operation, but a reliable one would fail, since it would have no means to tell which of the three RUs still contains the correct state. It is therefore necessary to expose these lurking errors as much as possible at the output, for instance by checksumming the memory at regular intervals.

The difficulty in dynamic TMR is the **reintegration**: the repaired unit must be brought exactly to the same state as the remaining good units (taught) and synchronized with the running units. The teaching requires a breach in the symmetric world of the synchronization. Without considering teaching, all units are identical, individual and isolated. Teaching requires a communication channel between a good unit and a repaired one, or at least a special copy program in every unit that must run in the background.

Several techniques have been developed for it. One consists in separating the processors in a memory and a (memory-less) processing unit (PU) with a voter, such as in Figure 4-17:

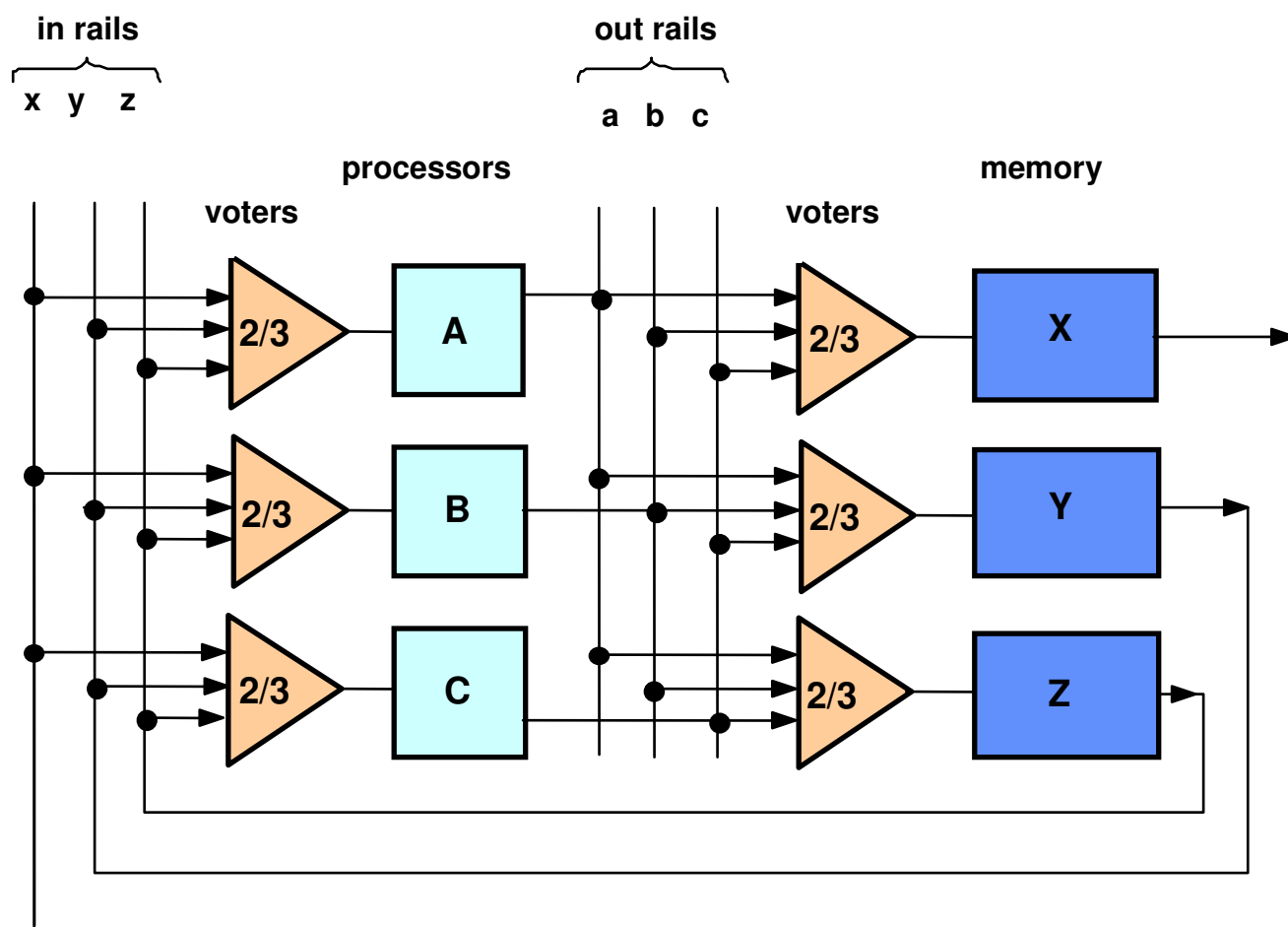


Fig. 4-17: Self-learning of a TMR System.

If a memory fails, then the new memory is actualised by every write operation of the processing units. If the PU intends to read a datum from a memory that has not yet been written into by a former operation, then the voter will transmit the value of the two experienced memories and outvote the random information in the uplearned memory. It will not signal an error during the teaching phase, though. Of course, this method yields only exponential teaching: it is still possible that after a very long time, variables remain that have not yet been written into. As long as this is possible, the voter cannot be used for error detection. Therefore, a part of the processing power of the PU must be spared for a cyclic update program, which reads and writes every memory cell as a background task.

The teaching of a failed and repaired PU is easy since these elements are assumed to be memoryless. This assumption does not completely hold, since each PU has an internal state. However, a PU can be forced to dump its contents and reload another context under program control. The normal interrupt system is sufficient for it. After repair of a PU, an interrupt request is sent to all PUs. The good PU and the fresh PU dump their respective contexts into memory and start with a new context, which is now the same for all PUs since it has been voted upon.

The TMR technique is particularly economical in terms of hardware, although the above diagrams may seem complicated. In principle, one can take any three off-the-shelf processors, with no special mechanisms for error detection, synchronize them and let the voter(s) choose the output. In reality, a great deal of investment is required to provide correct synchronization and teaching (in on-line repairable TMR). The voter itself becomes a critical point in the design.

4.3.2.1 An example of On-Line Repairable TMR: August Systems

August Systems series 300 [Wensley 83] is intended for industrial process control. It consists of three identical processor units (containing CPU, memory and I/O controller) called Control Computer Modules (CCMs) which work in parallel and in synchronism (Figure 4-18):

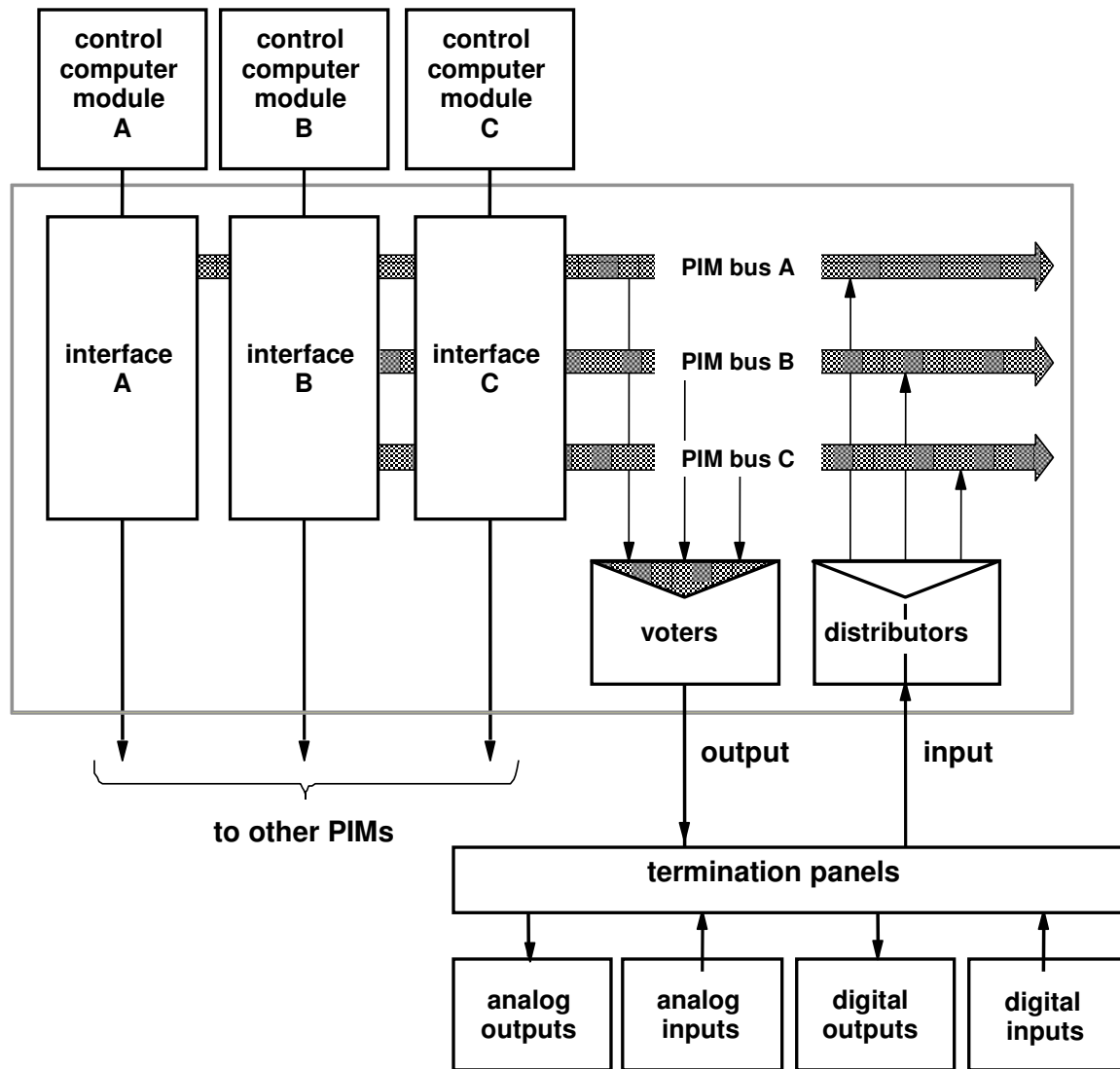


Fig. 4-18: August Systems' 300.

The outputs of the three CCMs are sent simultaneously over three independent buses (PIM Buses). The peripherals are in charge of voting the data. The input data are either replicated on the three buses by triplicated sensors or by a distributor unit.

The processors should be exactly in the same state at the same time. This implies that they work on the same set of input variables, even if there are slight differences between the values read on the three buses. The differences can arise either from a transmission error on a PIM bus or from small differences in the values generated when three independent analog sensors are used.

In fact, it is far more important to synchronize correctly the inputs than to synchronize the outputs: if there is a discrepancy in the input data, the internal states of the three CCMs would drift apart although no error is present, and the voter would be presented with three different output values.

A typical cycle used in a process control loop is shown in Figure 4-19:

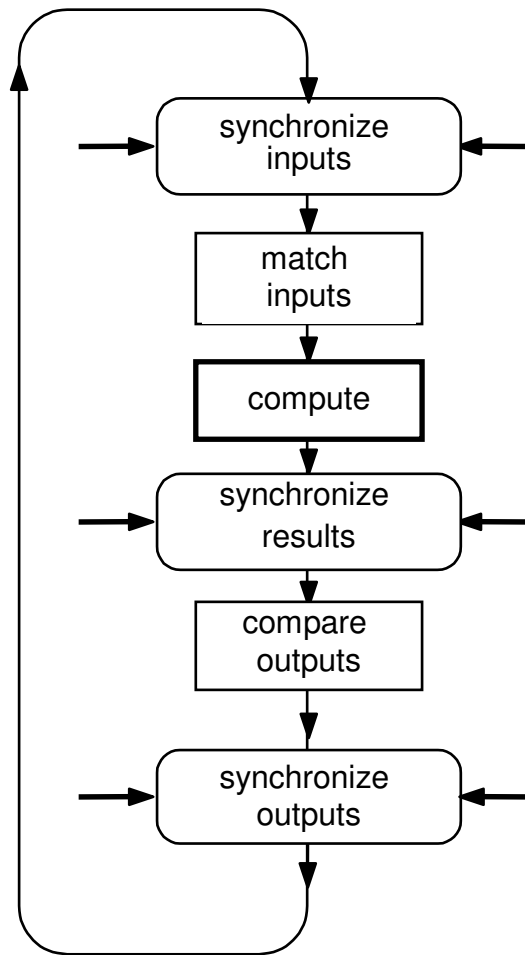


Fig. 4-19: Control Loop Cycle.

The input values are first read over the PIM buses. Then the CCMs exchange the values they have read and vote on them by software to reach a common value that will be used by all three for the computations. Since the when the three input values can be (legally) different, a median value algorithm calculates this common value. The CCMs communicate directly with one another through three dedicated buffer areas for synchronization. A CCM can read the memories of the other two CCMs but cannot write into them.

After the new output value has been computed, a second vote takes place before the values are released to the PIM. This is not absolutely necessary since the plant is supposed to vote also on the PIM buses, but the output synchronisation simplifies the voting by the plant.

The actual problem of this architecture is the on-line repair and warm start. The units must be mechanically and electrically constructed so that they can be removed and reintegrated while the remaining units stay on-line. In addition, a provision must exist to teach a freshly reinserted CCM while the other two CCMs continue normal operation. The reinserted unit tracks the state of one of the good units and traces itself in the current state. This step requires a great deal of care since the state is continuously changing. Some hints on the implementation can be found in [Wensley 83b].

4.3.2.2 An Example of Spare Pooling TMR: SIFT and FTMP

The FTMP [Hopkins 78] and SIFT [Goldberg 84, Wensley 78] are examples of spare pooling TMR or hybrid TMR. Both computers have been built under a NASA contract to develop a new generation of airborne computers that could fly fuel-efficient, but instable aircraft. The reliability of these computers should be as high as that of wing, i.e. about 10-10 /hour, since a loss of computer would cause the loss of the aircraft. FTMP and SIFT are competitors for this job. The FTMP (Fault-Tolerant MultiProcessor) has been built by Charles Stark Draper Laboratories (Cambridge, Massachusetts). The SIFT (Software Implemented Fault Tolerance) has been built by Stanford Research International (Menlo Park, California). Their architectures are very similar: both are multiprocessors, both use triplication and voting (TMR) and both use spare pooling to restore reliability after a failure (Figure 4-20):

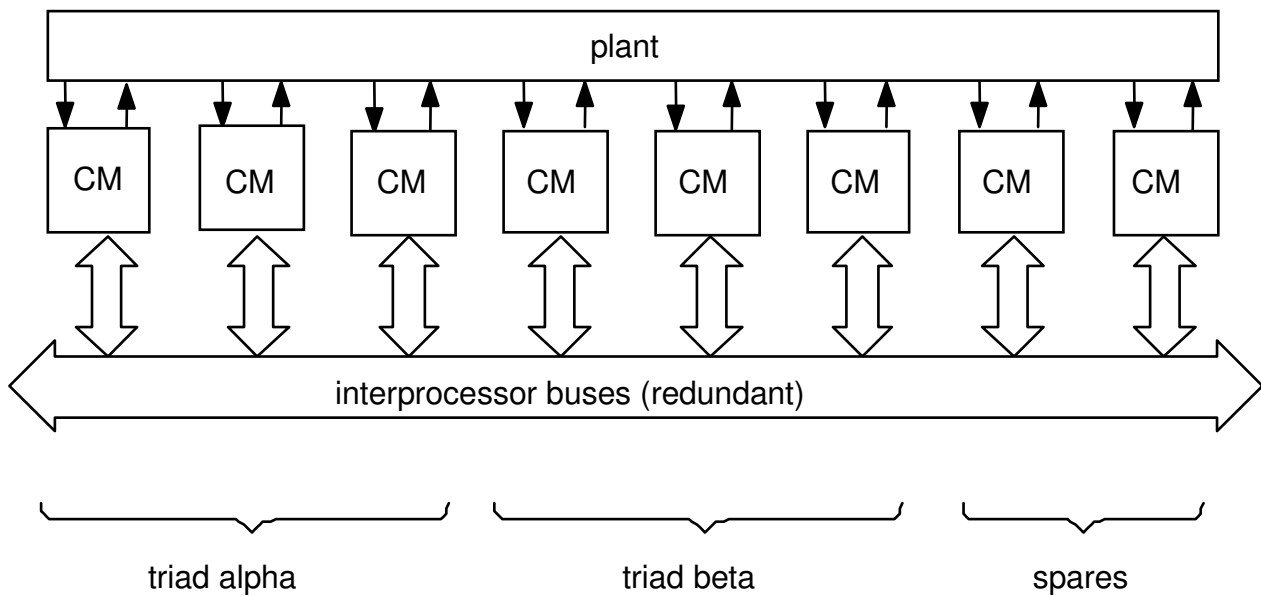


Fig. 4-20: A Generic View of SIFT and FTMP.

The computer consists of units called LRUs (line replaceable units). Each LRU contains a processor, a private memory, a local memory, an I/O interface and bus interfaces. An LRU is also a unit for fault-containment. The LRUs are located physically in different places of the aircraft to reduce common mode errors (due to mechanical or electrical damage). A processing unit consists of a group of three LRUs, called a TRIAD. The LRUs of a triad are synchronized at the execution level and at the input and output level. They execute the same operations simultaneously. The receiver of the information, which can be from the same or from a different triad, vote on the results. When a LRU fails, the fact is discovered and signalled by the receiver. The injured triad will keep on until the current task is terminated. Then another triad will reconfigure the injured triad and replace the failed LRU by inserting a spare from the pool in flight. This is analogous to a car rolling on three wheels until it finds an opportunity to stop and insert the spare wheel.

The FTMP supports 10 computers, allowing it to build 3 triads and keep one spare. The SIFT engineering prototype consists of 8 LRUs. It supports two triads with 2 spares.

In that mode, they operate as a multiprocessor with three and two processors respectively. At the first failure, the spare is taught by one of the intact LRUs and inserted. When no more spares are available, the failed triad is dismembered: the computing power is reduced, but two spares are now available to resist a third or a fourth failure. Therefore, SIFT and FTMP support graceful degradation.

The interconnect bus is also redundant and consists of a set of serial buses. Both SIFT and FTMP are connected to the plant by the standard MIL 1553 serial bus. This part is application dependent and redundancy of the I/O system is not specified. Each LRU has just its own MIL 1553 interface.

SIFT and FTMP differ on their approach to synchronization and matching. While SIFT (Software Implemented Fault-Tolerance) uses software to vote among the results, FTMP does the voting in hardware.

The structure of an FTMP module is shown in Figure 4-21:

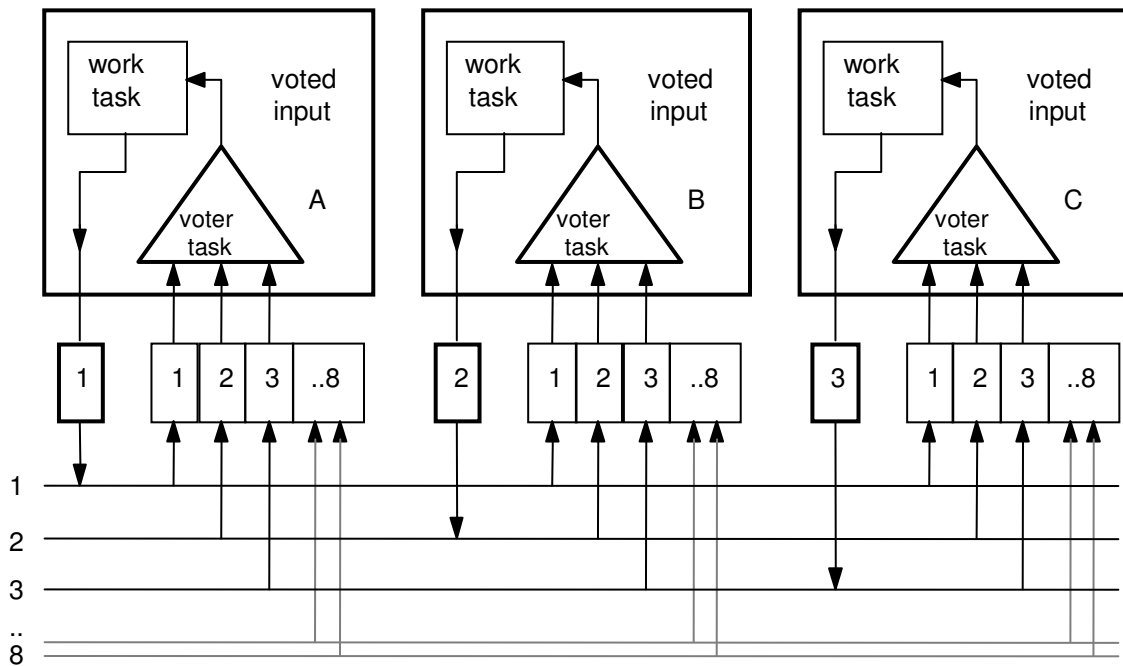


Fig. 4-22: SIFT's Broadcast and Voting.

Although all LRUs receive the broadcasted data, those for which it is not intended disregard it, but nevertheless vote on it to detect errors.

The software comparison scheme of SIFT reduces system costs, since the BGUs, CCUs and multiplexers/voters of FTMP are critical parts for which custom ICs are required. On the other hand, the speed is impaired by the lack of hardware support.

Voting and synchronization in SIFT takes some 100 μ s, i.e. about 100 times as long as hardware voting. SIFT's processors are synchronized within 50 μ s. FTMP's LRUs are synchronized within some 50 ns, since the clock is global to all LRUs (we will return to the clock below). Loose synchronization has at least one positive aspect: the resistance to flash (burst of electromagnetic energy) is greater in loosely coupled system, since there is a risk that two tightly synchronized units could be affected in the same way by the flash and outvote the good unit. But in general, loose coupling only increases the voting time.

The biggest problem of SIFT has been that it has not met the expectancies with respect to the real-time requirements. A comparison study done by the FTMP team [Smith 84] and the SIFT report itself [Goldberg 84] lead to the conclusion that a loose synchronization and voting scheme can hardly fulfill the real-time requirements of a flight control system.

However, the bulk of the work of SIFT has not been done on the hardware reliability, but in the field of the software correctness proof and formal specifications. In that respect, the FTMP and SIFT projects are complementary rather than competitors.

4.3.3 Dual Workby

Dual Workby is the minimum configuration for workby operation. It is also referred to as **synchronous-and-match**, **hot sparing** or **hot-standby** (we prefer to use "Stand-by" when recovery techniques are used). It relies on a pair of identical, synchronized units. This arrangement can be used for two different purposes, for error detection (by duplication and comparison), or for persistency (by spare switching) as Figure 4-23 shows:

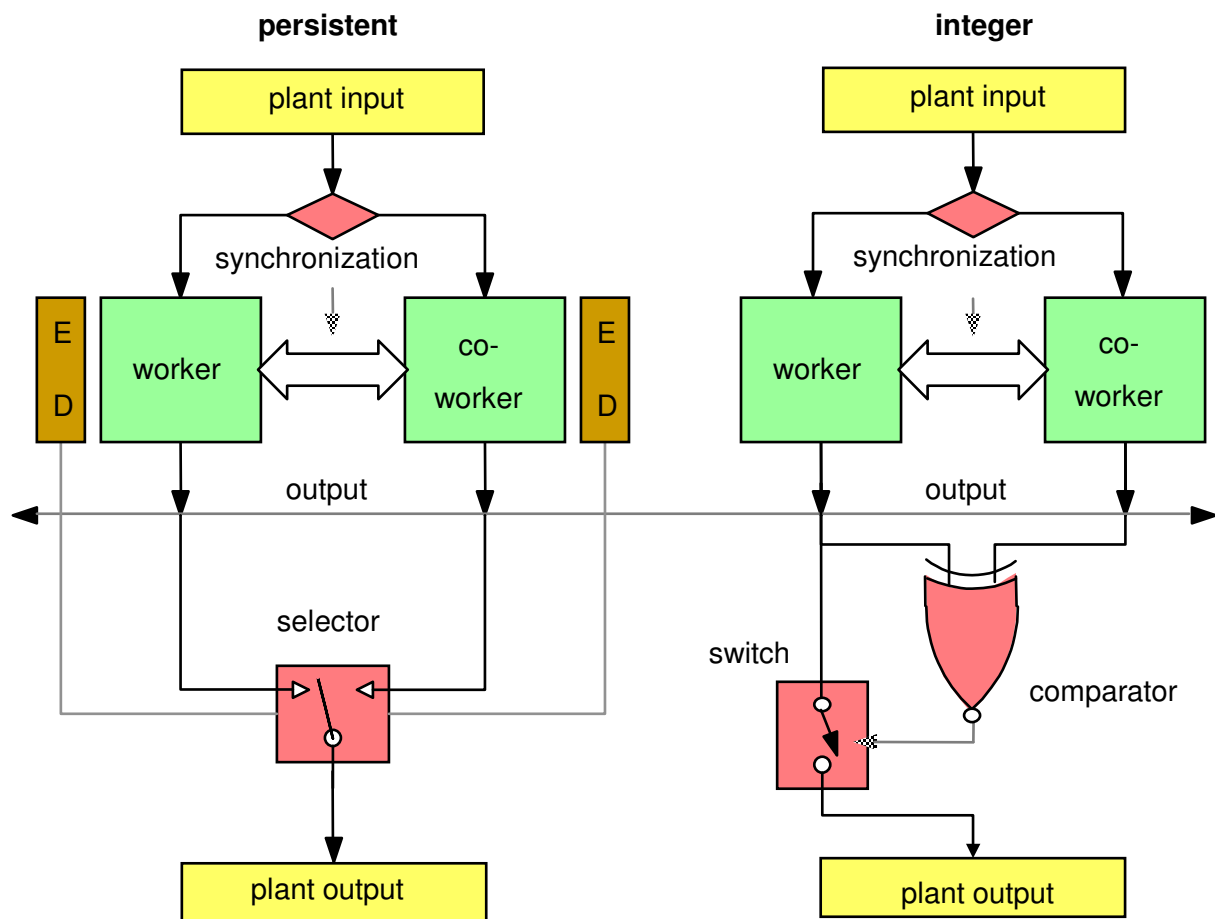


Fig. 4-23: Dual Workby.

In both cases, the operation above the dotted line is the same: a worker unit and a co-worker, which are completely identical and deterministic units, are performing the same work at the same time. Their executions are synchronized, and the input data they receive are matched, so one can rely on the determinism of computer execution to maintain them in the same state. Which of the two units is connected to the plant and which is redundant is only decided by what is below the dotted line.

For error detection, a simple comparator (which may be synchronized with the execution clock) detects if a discrepancy occurs. If this is the case, an error is signalled and the error signal may be used to sever the output to guarantee a fail-stop behaviour.

Although both units are replicating computations, comparing their results with a comparator can only tell that an error occurred, but could not tell which unit is faulty. Therefore, an independent redundancy is required to decide which unit is faulty. This redundancy is not drawn on Figure 4-23.

For fault-tolerance, a selector device chooses between one or the other unit. Since both units are in the same state, the switching can be instantaneous and mask the error completely. This, however, requires that an error is immediately detected upon its occurrence, and that the selector takes action immediately. So, the additional redundancy for error detection is crucial here, because unlike TMR, the masking depends on the error detection capability.

The dual-workby solutions differ in the way errors are detected:

- quadding, or 2/4, which relies on the duplication of a fail-stop pair,
- dual self-check (DSC) which relies on a pair of self-checking units, whereby the amount of redundancy for error detection is smaller than one, and
- Dual Compare and Diagnose (DCD), which relies on off-line error detection.

The selector or the comparator/switch are, like the voter in the previous example, a single point of failure, and they should be made highly reliable, possibly by replicating them.

4.3.4 Quadding

A 100% error detection can be achieved by quadding, i.e. making each worker and co-worker a workby pair for the purpose of error detection. This means that the fourfold amount of hardware is required to perform the function. The two redundant units, active and back-up, also called worker/mirror are interchangeable, for instance by means of a switch (Figure 4-24).

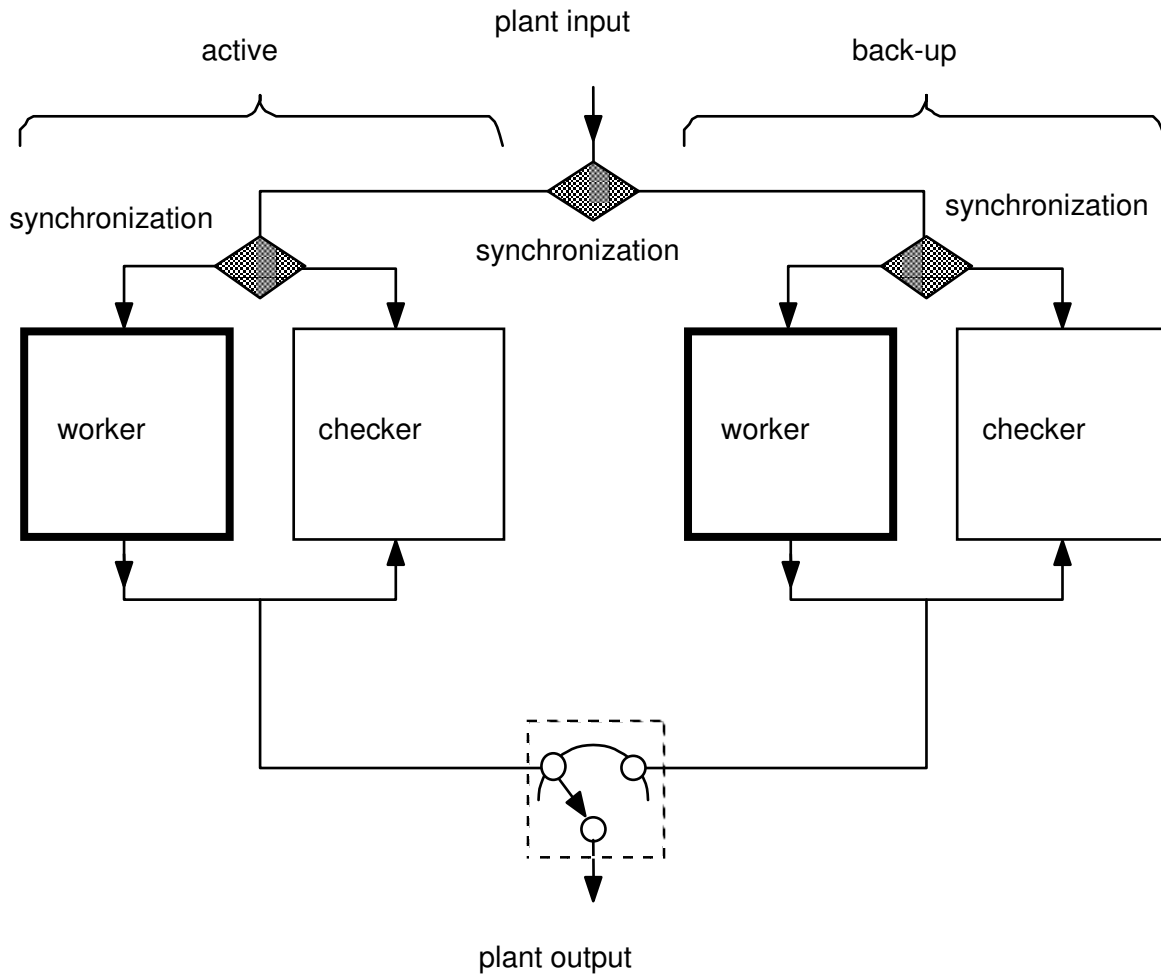


Fig. 4-24: Quadding.

Quadding represents a higher hardware investment as 2/3. Accordingly, its reliability is lower: While the MTTF of a 2/3 is only 5/6 (0.833) of a simplex system, the MTTF of quadding is 6/8 (0.75) as will be shown in Chapter 9. In addition, quadding requires two stages of synchronism: a first level to keep the two pairs synchronized, and a second level to keep the synchronism within a pair. The reason why quadding is often used is that it provides a higher modularity than 2/3 voting. In addition, no voter is required, and therefore hardcore (single point of failure) is reduced. The switch can be embedded into the plant much simpler than a voter.

4.3.4.1 A First Example of a Quadding Multiprocessor: the iAPX 432

The concept of quadding can be extended to multiple processor systems, as exemplified by Intel's iAPX 432 architecture [INTEL 82, Johnson 84]. The iAPX 432 is a multiprocessor system in which processors and memories are interconnected by a distributed crossbar consisting of two bus arrays, the processor buses ACD and the memory buses MACD. At the crossing points of the two buses, the BIUs (Bus Interface Units) do the switching. The memories are connected by MCU (Memory Control Units) which perform addressing, refresh, error code generation and controls interleaving (Figure 4-25):

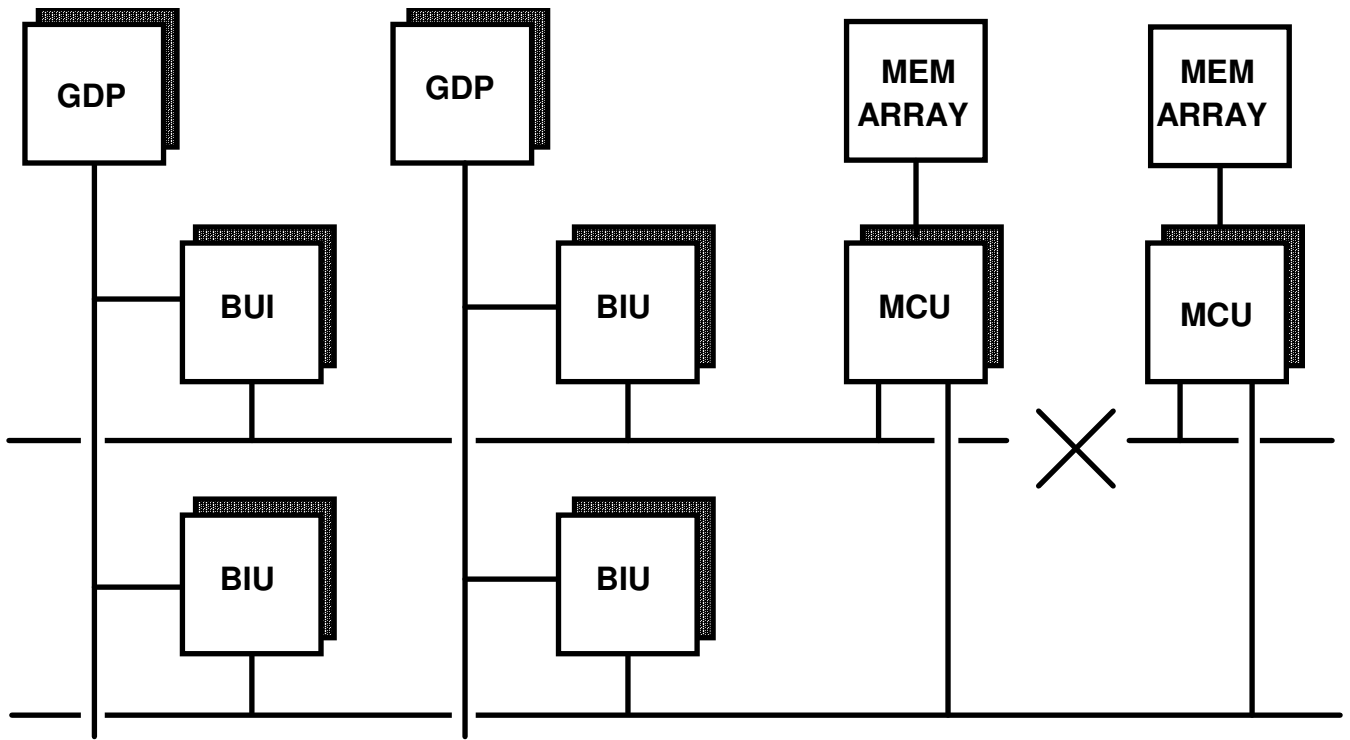


Fig. 4-25: iAPX Interconnection Architecture.

A processor has no local memory: its state consists of a set of registers. The state of a task therefore resides in memory except for the register context, making it easy to swap tasks between processors to share the load among the existing processors.

For integrity, each processor is duplicated at the chip level and consists of a self-checking processor pair (duplication and comparison). For fault-tolerance, each self-checking processor is covered by another failstop-pair in work-by mode. Therefore, a faulty unit can just be switched off at any moment and the work-by unit will take over. The switching is at the charge of the BIU (Bus Interface Unit), which is itself duplicated on each processor.

If the fault takes place during a bus transfer, the shadow cannot take over "on-the-flight" and it is necessary to retry the bus transfer. In any case, a quiescent period must first elapse to let transient faults settle. This period lasts from 16 μ s to 2 s according to the application.

This simple method requires a sophisticated error reporting system. It consists of a duplicated serial bus, running alongside the parallel buses ACD and MACD, which transmits the reconfiguration information to the BIUs. It is also used to catch-up errors which leaked out of a confinement zone because of a bus driver failure.

Repaired processors can be reintegrated easily in the working set, since they do not hold the task's state, except for the register contents. Reintegration is done when the context of the processor is irrelevant, i.e. at the next task switch for that processor.

Reintegration of memory is more difficult. Sometimes it is only necessary to clear the memory, which takes about 40 ms for a 256 KB array. If this is not possible, then the content of the good memory is poured into the repaired one. This copy operation takes about 800 ms for a 256 KB array. It is necessary to stop the operation of the computer for about 1 s, since otherwise, consistency problems would arise (see above August Systems's series 300). Thus, this scheme can only be applied when the mission involves frequent quiescent phases during which the computing can be stopped (scheduled repair).

The reliability of the iAPX 432 is therefore limited by the probability of a second fault in the same memory pair before a quiescent point is reached. The probability of a second processor failure before a task switch is made is far lower, and further, processor insertion takes much less time.

4.3.4.2 Another Example of Quadding in Multiprocessors: STRATUS/32

Another example of quadding is STRATUS's /32 system [Freiburghouse 82]. The architecture is similar to the iAPX 432 (Figure 4-25). The number of interprocessor buses is limited to two, and there can be many memories connected to them. The processor units are fail-stop. They include local memory, which is lost when a processor fails. Tasks can be executed with several levels of redundancy, either on one fail-stop processor, on one processor

with a back-up or on a pair of work-by processors. Each of the boxes in Figure 4-26 represents a unit consisting of two halves, forming a fail-stop unit.

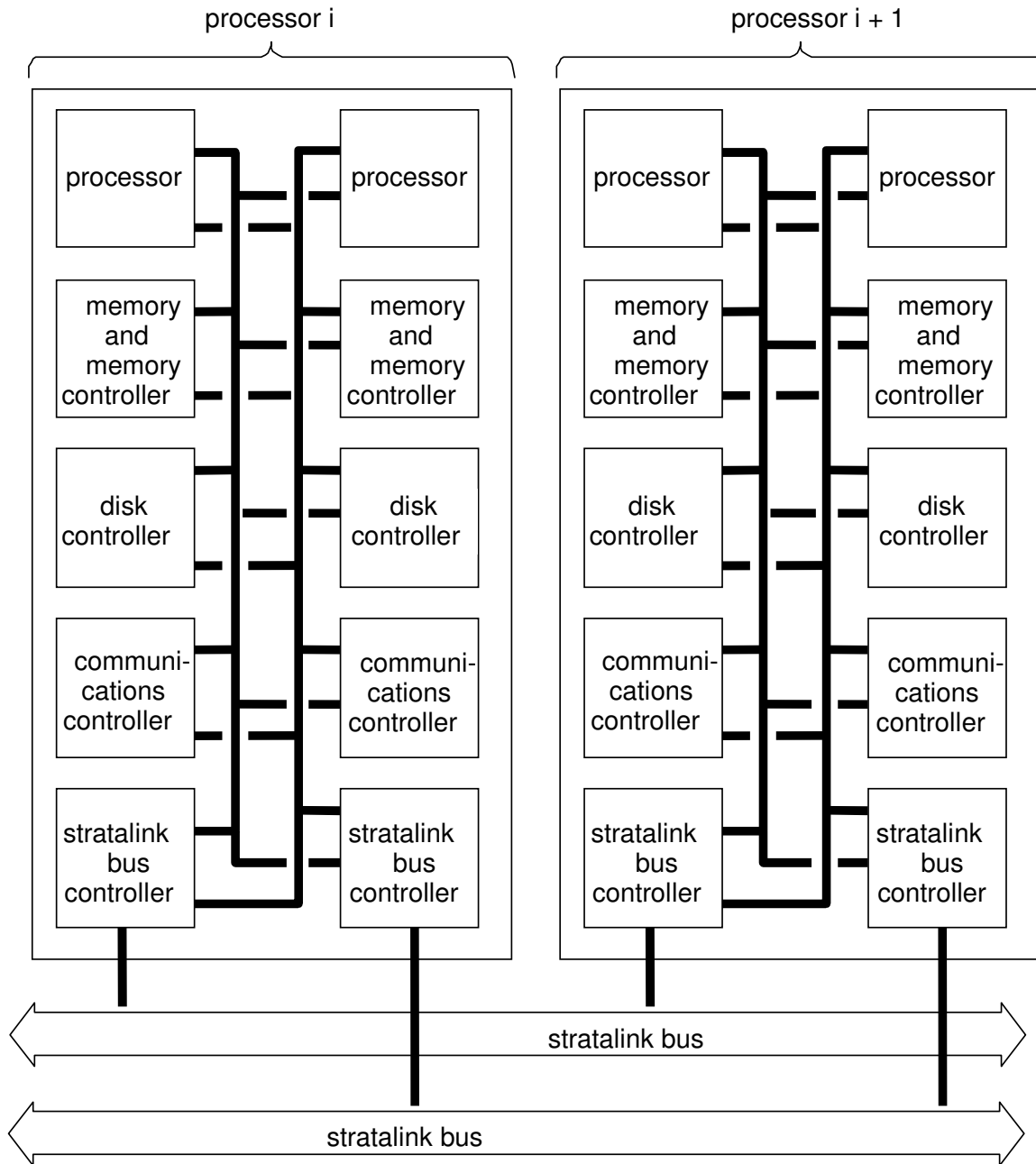


Fig. 4-26: The STRATUS/32 Architecture.

STRATUS/32 considered in detail the work-by operation of disk units. Since disks particularly and mechanical devices in general are non-deterministic elements, their inputs must be carefully synchronized by techniques similar to TMR's to achieve consistency of the inputs for both replicated units. Figure 4-27 shows the structure of a disk controller.

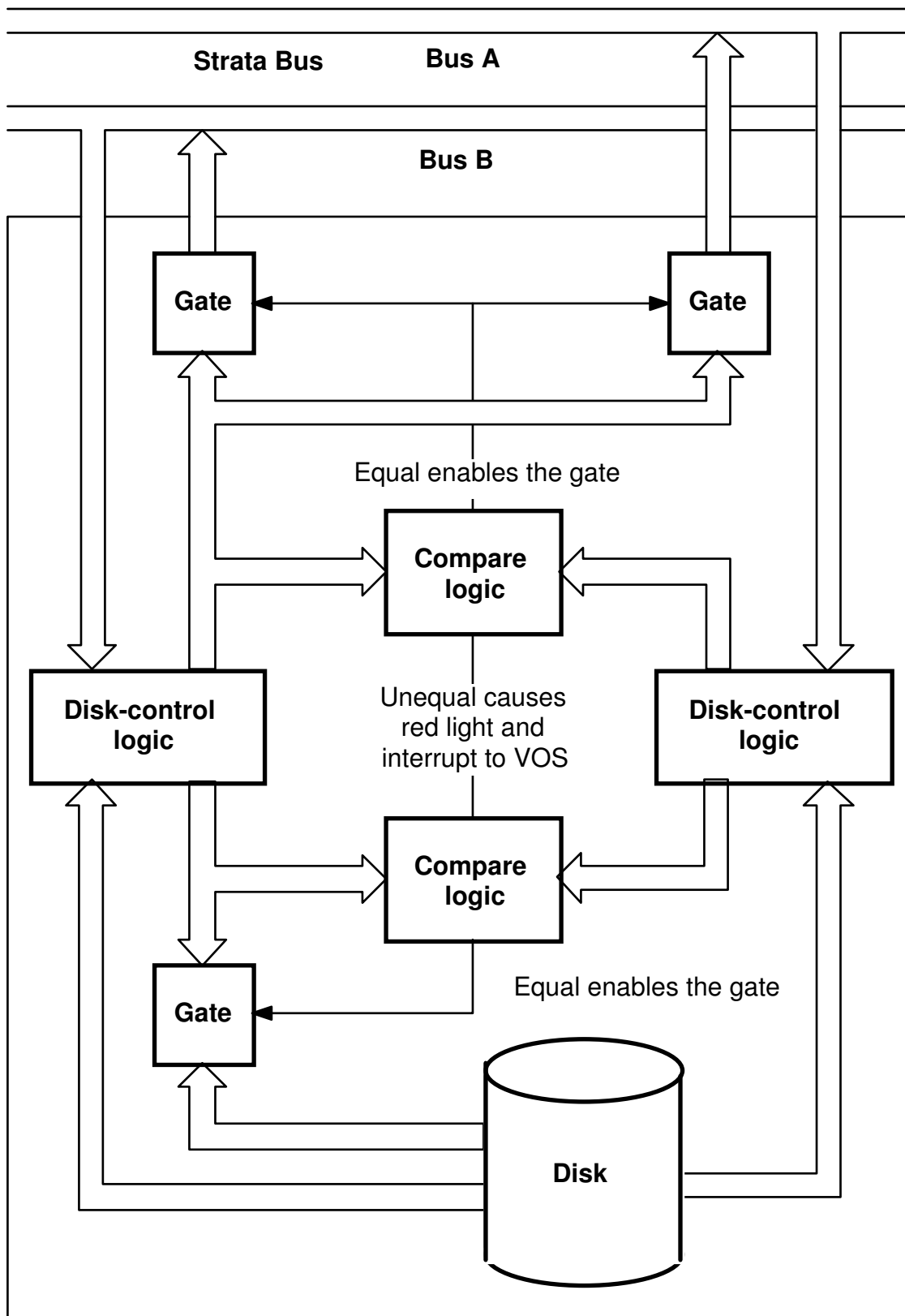


Fig. 4-27: The STRATUS/32 Disk Control Unit.

4.3.5 Dual Self-Check

The largest disadvantage of quadding is that a fourfold hardware is required to provide integer as well as persistent operation. This overhead does not only cost hardware and power consumption, but also unreliability. The redundancy can be reduced if the error detection is not done by duplication and comparison, but by coding instead.

Indeed, coding provides plausibility checking at a much lower price than duplication and comparison. A Hamming Code for single and double error detection requires only 6 check bits for 32 bits of useful data, see Table 4.6.

Figure 4-28 shows the principle of Dual Self-Check:

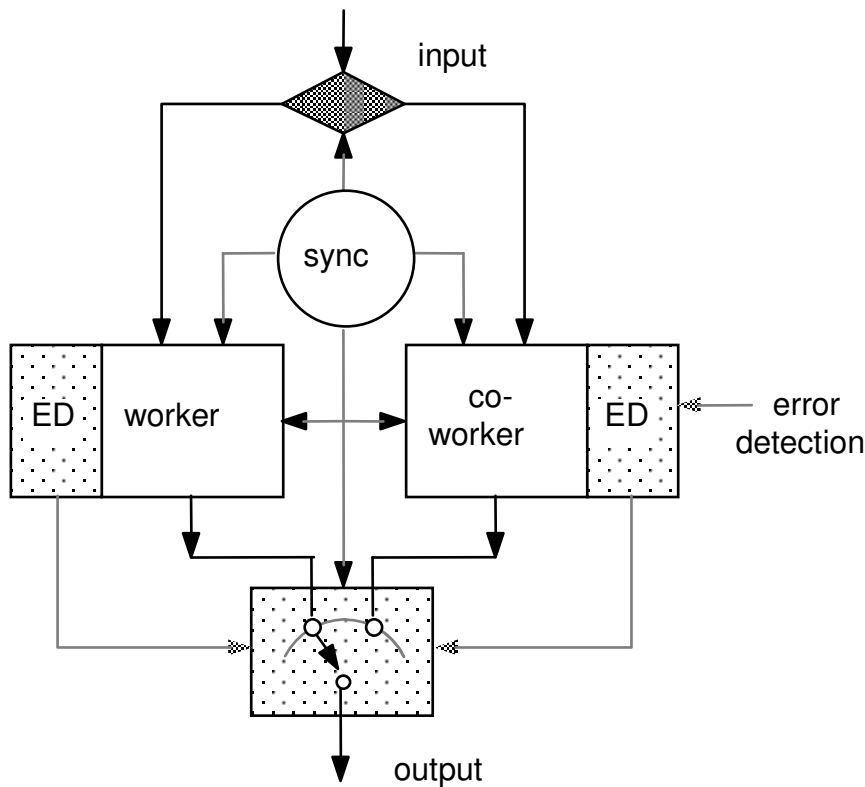


Fig. 4-28: Dual Self-Check.

One can show (Chapter 9) that Dual Self-Check becomes more interesting than TMR when the amount of error checking hardware does not exceed 50% of the functional hardware. This goal can be achieved by providing ECC for all RAMs, parity on buses, CRC on serial transmissions and duplication and comparison for complex logic. One can then achieve 100% error detection coverage, which is necessary for integrity.

DSC is used in numerous industrial controllers, such as Siemens's AS220 H and Simatic S5-135H. However, self-checking hardware is quite difficult to built, since the design effort is practically doubled. Therefore, a tendency exists to reduce the coverage to save design efforts at the expense of reliability and integrity. At the end, the reliability will depend entirely on the error detection coverage (see Chapter 9).

4.3.6 Duplication, Comparison and Diagnostic (DCD)

In many cases, complete error masking is not required, but integrity primes over persistency. Then, the amount of error detection logic can be reduced to plausibility checking. Figure 4-29 shows the principle:

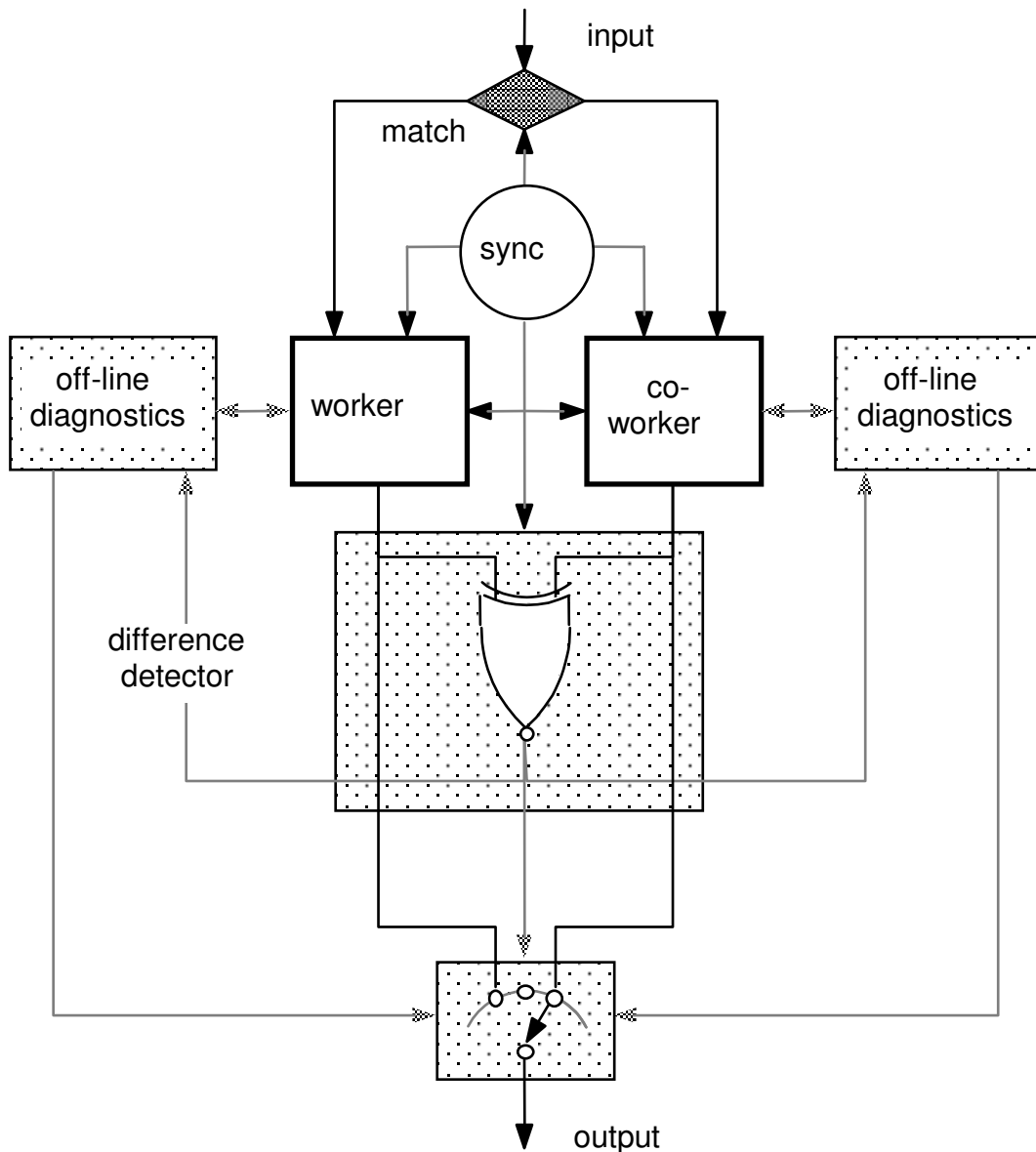


Fig. 4-29: Duplication, Comparison and Diagnostic

The workby units are operated in parallel and their outputs are compared. The output of the comparator does not act on a switch, but triggers a diagnostic program. The outputs are stopped and each unit performs a health check on itself (and possibly on the other unit). If a faulty unit is found, it is disconnected and work continues on the remaining unit. In this post-failure mode, a second error would lead to false output. Such a reduced-safety mode is normally only permitted when a human operator is supervises and during a limited time.

A problem of DCD is that in some cases, it may not be possible to declare one of the units faulty, for instance because the error was only transient. Then, an arbitrary choice is made, and the unit declared faulty is reloaded with the state of the unit declared as good. This is necessary, since the states of the units may differ and a second error could occur within a short time. Again, such a reload may be prohibited in high-safety applications.

The dependability of DCD is limited by the probability of picking up the faulty unit within an acceptable period of time, as Chapter 9 will show.

4.3.6.1 An example of a DCD structure: the AXE 10

Numerous examples of dual workby computers exist today, among them are the ESS [Toy 78], and AXE 10 [Ossfeld 80] which are used in telecommunication switching. Their configuration is shown in Figure 4-30:

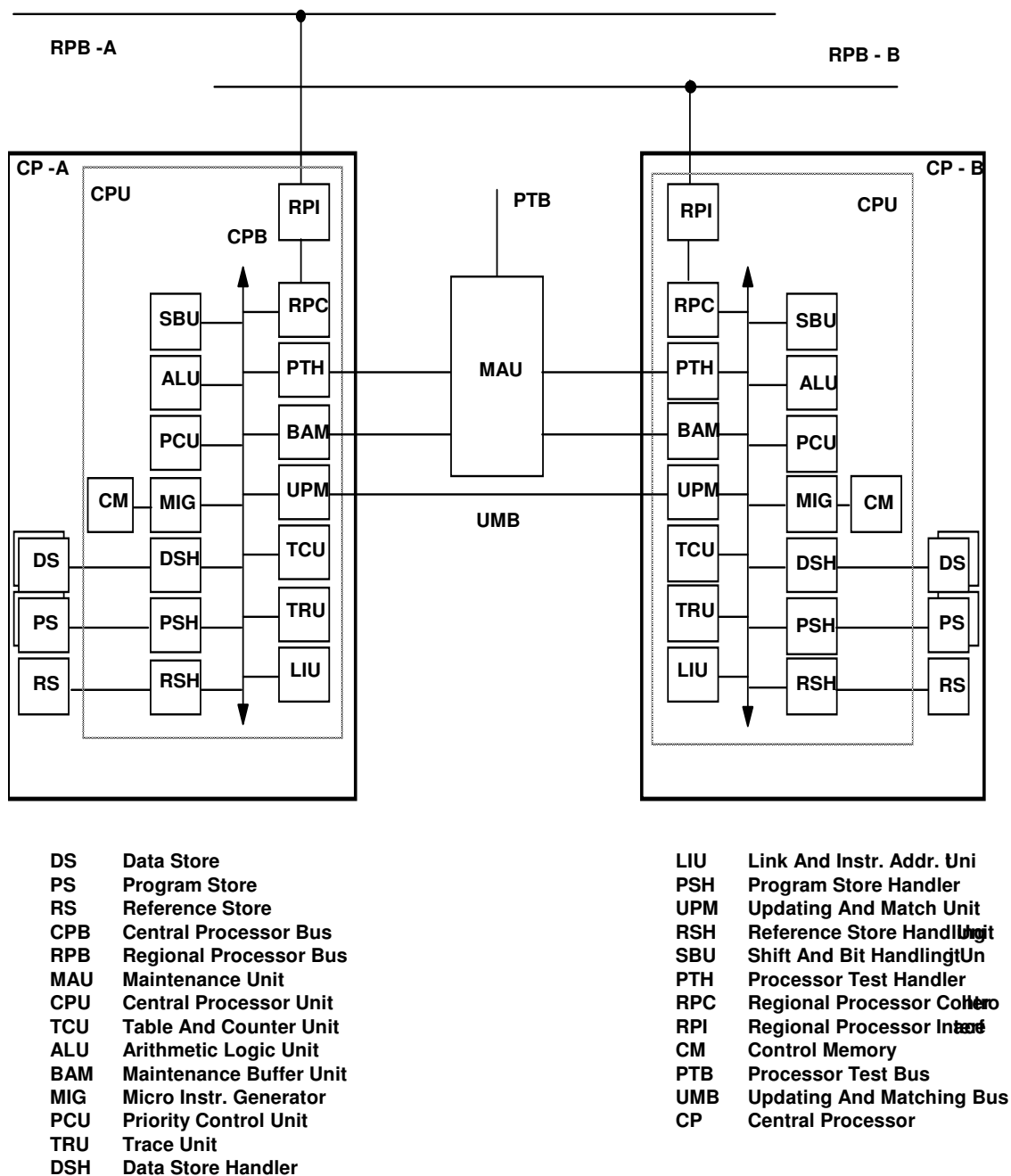


Fig. 4-30: Duplication, Comparison and Diagnostic Computer (AXE 10).

Originally, this concept was introduced for error detection: the outputs of both processors are compared. A mismatch indicates an error, but not which unit is faulty. Therefore, upon detection of an error, the units had to undergo self-check to decide which unit is faulty. The temporary loss of function is tolerable in telecommunication switching.

The computations are resumed on the non-faulty unit, if one can be found. Since off-line tests cannot detect transient errors, it is likely that no error will show up. In this case, the computation is resumed on a randomly chosen unit. The other unit can now be taken off-line for repair and update.

The separated operating mode is also useful to perform software updates: the work-by unit is loaded with a new version of the software and inserted in place of the working unit. If there are problems with the new software, one can switch back to the old version. This supposes of course that both versions operate on the same data structures.

The next step was to make the duplicated processors self-checking: each has on-line error detection circuits and, for the memory, error correction circuits. Each processor is considered a fail-stop unit. Therefore, the question of which unit is faulty in case of discrepancy does not arise any more: switching is instantaneous. This shows that a DCD structure can be upgraded to a DSC structure starting from the same design and improving the error detection capabilities of the hardware.

4.4 Synchronization and matching of work-by units

4.4.1 Synchronism and Determinism

All the previous methods (massive, N/M voting, dual workby) assume that the working units are all exactly in the same state. This assumption is also made for error detection by duplication and comparison. A similar requirement exists for coding.

The method to keep all units in the same state is to let them perform the same work simultaneously on the same inputs. This is based on the assumption that the units are absolutely identical and deterministic. Any non-deterministic element within a unit is a source of discrepancy and must be carefully matched.

Non-determinism may arise from analog elements, like watchdog timers, mechanical devices or variable delays in memory access. Although digital circuits tend to be deterministic, synchronization circuits are a potential source of non-determinism. Arbiters, for instance, may decide differently depending on slight variations of the order of arrival of signals.

The existence of an error recovery technique in the units can cause a synchronization problem if one unit is correcting an error and the other(s) are not. Among other non-deterministic components are the synchronization circuits:

Workby requires in addition that the inputs of the redundant units be identical and simultaneous. This requirement cannot most of the time be fulfilled without external help: due to slight variations in delay, interrupts may be responded earlier in one unit than in the other, and let the execution flow of the programs diverge. The redundant processors often receive their data from redundant input devices, so as not to introduce a single point of failure. In that case, discrepancies between the input data are expected even as a normal case.

We will consider the following problems of synchronism:

- Non-determinism in the units, Execution synchronization
- Non-determinism in the inputs, Synchronizing and matching the inputs

The "SYNC" circle and the "sync/match" diamond in the above diagrams remind that the requirement exists.

4.4.2 Synchronizing the Execution

All replicated units should have a common time reference or they will have difficulty in agreeing on a common time frame for inputs and outputs. The replicated units may be synchronized at the processor clock level (instruction synchronism), at the bus access level (bus synchronism) or only at the input/output level (I/O synchronism). Although in theory only a synchronism at the input/output data level is required, a closer synchronism is recommended to provide a smooth switchover.

There is no such thing as an absolute time reference [Lamport 78]. One could think of implementing one by broadcasting the clock to all units, for instance by a radio signal. This could not prevent one unit from receiving that signal some nanoseconds before another. Thus, the common time reference depends on the distance between the units. It is theoretically not possible to achieve a closer synchronism than within the time light needs to travel from one extreme of the system to the other. Since propagation delays are in the order of 5 ns/m, and drivers/receivers add delays to it, synchronism cannot be maintained tighter than about 50 ns if the units are in the same board and 100 ns if they are on different boards and 1 μ s if they are in different crates within the same cabinet.

To show that the problem is not trivial, consider what would happen if the outputs of a triplicated clock were applied to a 2/3 voter (Figure 4-31):

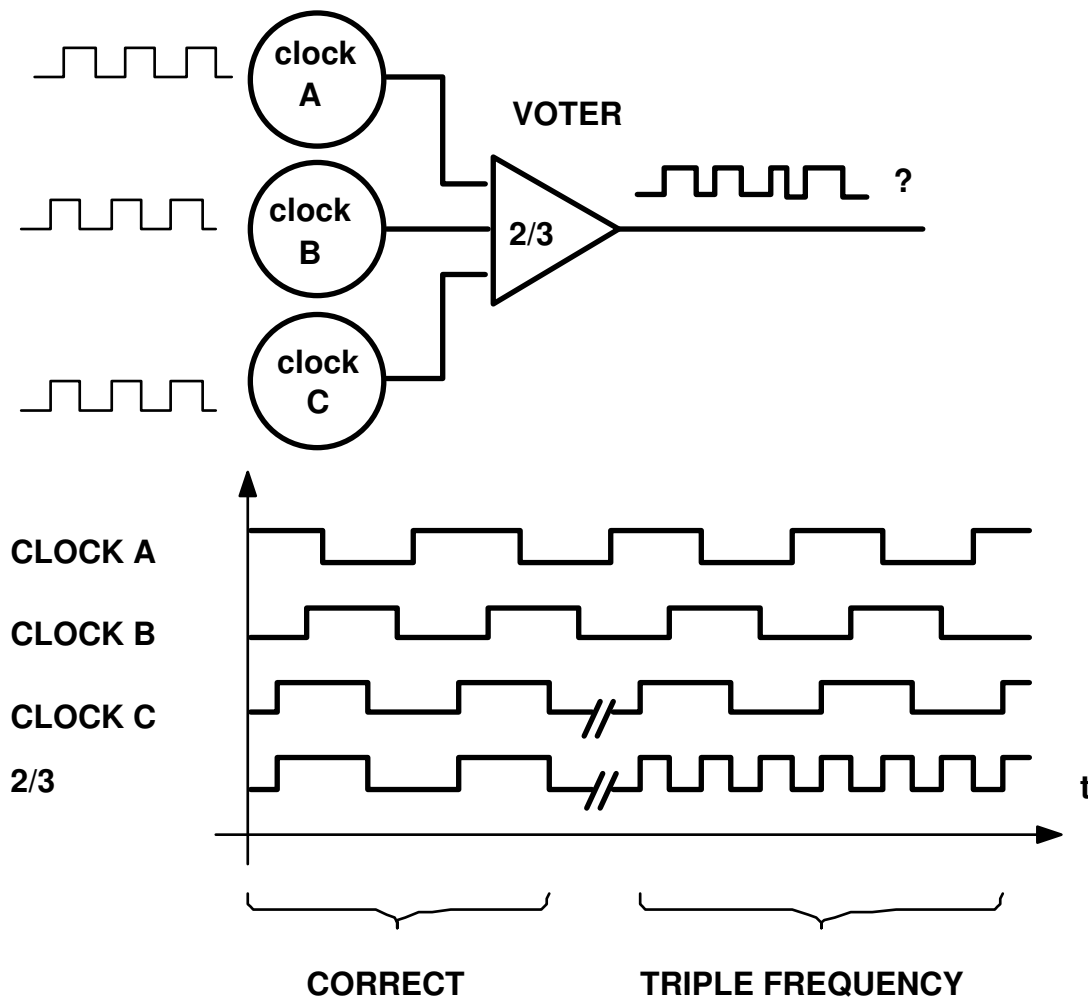


Fig. 4-31: Running a Triplicated Clock through a 2/3 Voter.

Although no input is faulty, the circuit behaves as a frequency tripler.

There are classical solutions to the synchronisation problem like crystal oscillators modulated by phase-locked loops. Hardware solutions have been developed which require at least four clocks to reach a common time reference under all situations [Smith 81, Hopkins 78, Kessels 84]. The solution used in the FTMP is shown in Figure 4-32:

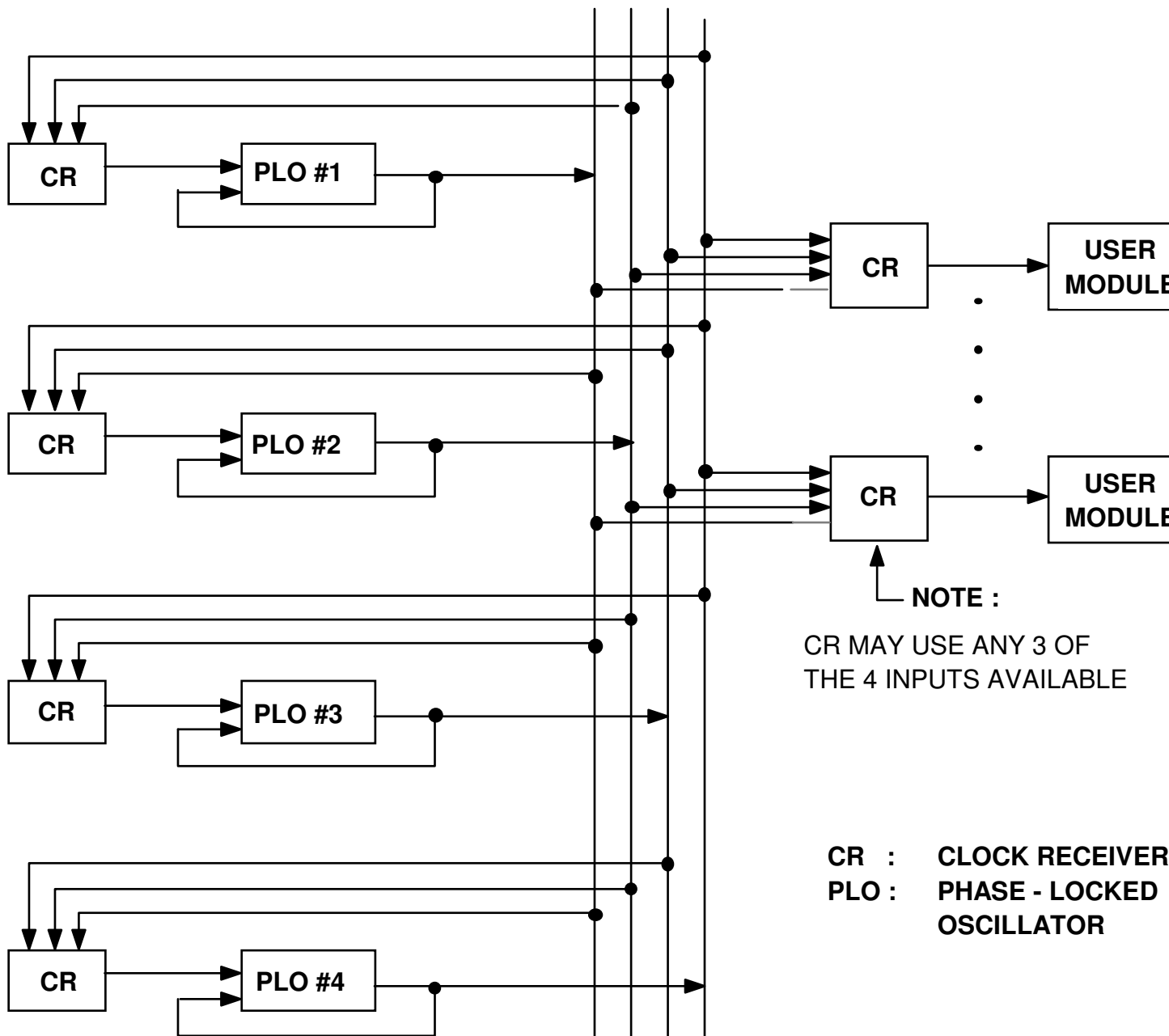


Fig. 4-32: FTMP's Fault-Tolerant Clocking Scheme.

The above circuit can ensure that all machines use the same clock frequency, but not that they all see the same "absolute" time - keeping in mind that absolute time exists only in one place.

A common time reference can only be achieved by communication between the units and agreement on a common value. Since communication takes a certain time, one cannot expect a common time reference to emerge which is shorter than one message exchange. Further, the synchronisation must also work in spite of failure of any of the participants.

From this, it is clear that closely coupled elements can be synchronized better than loosely coupled elements. In fact, all synchronization methods that involve communication protocols suffer from inefficiency.

We will therefore treat the building of common time like the matching of input data and discuss both topics together.

4.4.3 Matching the Inputs

To achieve exactly the same internal state, all replicated units should receive the same data at the same time, even in the case when they receive different data. The only exception which could be acceptable is when the process data are continuous and the algorithms used would let the output converge to the same value, but this would exclude even the use of simple integral control in a control loop. Analog TMR devices are rarely used in practice.

Thus, inputs must be synchronized such that all units start from the same input data vector, even if some units received different inputs.

Figure 4-33 shows an example of a TMR system in which the three units receive their input data from different sensors.

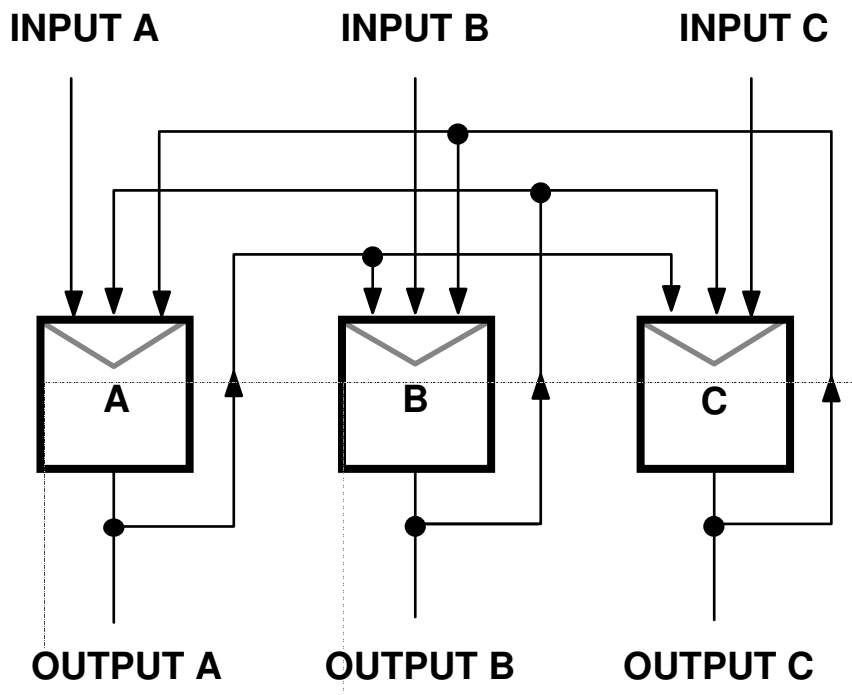


Fig. 4-33: Synchronizing the Inputs of a TMR System.

This synchronization implies that the units exchange the data they have received to reach a common agreement. Exactly the same problem occurs when the units try to agree on a common time.

4.4.4 The Byzantine General's Problem.

The problem of reaching a common input data or time reference value in spite of faults is an interesting investigation subject [Pease 80, Lamport 82, Frison 82, Moore 84, Krishna 84]. It is called "interactive consistency" or "source congruency" depending on whether the author is with the FTMP team or with the SIFT team.

It appears that it is theoretically impossible for three computers to agree on a common value, and that at least 4 are required for this. The proof for it is known as the "Byzantine Generals' problem" [Lamport 82] but it was already described in a less didactical way earlier.

A city has been attacked by three Byzantine armies. The city can be caught only if all three armies decide to march at the same time. They can also decide to retreat, but then they must all do it at the same time. Unfortunately, Byzantine generals are well-known as venal and it is likely that one of them has been bought by the city inhabitants with the aim of destroying the other generals' armies, for instance by letting one army attack or retreat without moving the others. Therefore, an algorithm must be found through which the mutually suspicious generals can reach a consensus on whether to attack or to retreat.

The attack or retreat command is given by general A which communicates it to generals B and C. Since generals B and C are mutually suspicious, they exchange the command they received to see if they agree. One general will assume that the command is correct if it received it from two different persons in the same way.

The situations which can occur are shown in Figure 4-34:

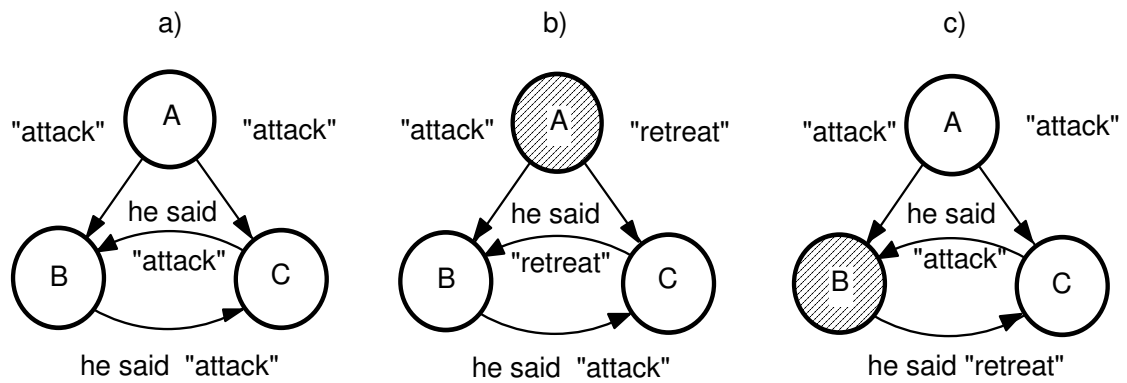


Fig. 4-34. The Byzantine Generals Problem

No traitor: General B and C receive the same command and they march on.

General A is a traitor: he sends the command of attack to B, but the command to retreat to C. B and C exchange their messages and find them in contradiction: they cannot take any action.

General B is a traitor: he received the command to attack from A, but transmitted the retreat command to C. C himself has received the command to attack from A, so C has received two contradictory commands and cannot take any action.

For general C, the cases 2) and 3) are not distinguishable: he cannot tell which one of A or B is the traitor and therefore cannot take any decision.

One can generalize the problem by building groups of generals, and to consider each group as a super-general. This way, one can see that to solve this problem you need at least $3t + 1$ participants to cope with t traitors. A more formal proof can be found in [Pease 80].

There are however two ways to solve this situation with only $3t$ participants:

1. **Encryption.** General A should encrypt his message such that B cannot falsify it, for instance, by writing the command on security paper with his seal and signature. Situation 3) can no longer occur, since B cannot falsify the command it received from A. The worst he can do is not deliver it. So, if situation 2 occurs, C knows that A is the traitor. Several techniques for data encryption with public keys have been developed which could serve this purpose.
2. **Atomic broadcast.** If the message system is such that the command is transmitted simultaneously to all participants, then A cannot send a different message to C and B, and situation 2) cannot occur any more. In addition, since C and B broadcast their commands, the traitor will be immediately uncovered. This transmission requires that the message is either received identically or not at all by all participants. Some networks such as Brown Boveri's PartnerBus support in hardware the construct of atomic broadcast, by a token passing mechanism. In other networks, one must implement it by software. Atomic broadcast requires that the message be encoded with an error detecting code such as a CRC. Then, the probability is extremely small that any participant recognizes a false message as good. An additional complication comes from retransmission: a good message retransmitted at an inopportune time is also a false message - the traitor could store a valid "attack" message and deliver it in place of the "retreat" message. Therefore, it is important that the messages be stamped with a **sequence number** - and that each participant receives exactly one copy of each message. The daytime is not adequate as a time stamp, since there would be no control of lost messages, and since the message exchange is used precisely to establish a common daytime.

The problem can be complicated by considering random transmission delays, time-outs and unreliable messengers, but at the end, it is worth asking oneself how likely it is that a malicious error will appear in a faulty computing system. A software error in all the replicated units is much more likely.

REFERENCES

- [ACM 84] Special Issue on Computing in Space,
Communications of the ACM , September 1984
- [Davies 78] D. Davies & J.F. Wakerly,
"Synchronisation and Matching in Redundant Systems",
IEEE Transactions on Computers, Vol. C-27, No.6, pp.531-539 , June 78
- [Freiburghouse 82] R. Freiburghouse,
"Making Processing Fail-Safe",
Mini-Micro Systems , May 1982
- [Frison 82] S.G. Frison & J.H. Wensley,
"Interactive Consistency and its Impact on the Design of TMR Systems",
FTCS-12, 12th. Int. Symp. on Fault-Tolerant Computing, Santa Monica, pp. 228–233 , June 1982
- [Horninger 85] K.H. Horninger, H.P. Holzapfel,
"Fehlertolerante VLSI-Prozessoren",
GI-Fachgespräch 'Themengebiete FTCS-16', Munich 1985
- [INTEL 82] Intel Corporation,
"iAPX 432 Interconnect Architecture Reference Manual",
Order Number 172487-001, 1982
- [Kessels 84] J.L.W. Kessels,
"Two Designs of a Fault-Tolerant Clocking Scheme",
IEEE Transactions on Computers, Vol. C-33, No. 10, pp 912-919, October 1984
- [Krishna 84] C.M. Krishna, K.G. Shin,
"Synchronization and Fault-Masking in Redundant Real-Time Systems",
FTCS-14, 14th. Int. Symp. on Fault-Tolerant Computing, Orlando, pp. 152-157 , June 1984
- [Krol 82] Th. Krol,
"The '(4,2)-Concept' Fault-Tolerant Computer",
FTCS-12, pp. 49-54 , June 1982
- [Lamport 78] L. Lamport,
"Time, Clocks and the Ordering of Events",
Communications of the ACM, Vol. 21, No. 7, pp. 558-565, July 1978
- [Lamport 82] L.Lamport, R. Shostak, & M. Pease,
"The Byzantine Generals Problem",
ACM Transactions on Programming Languages and Systems, Vol.4, No.3,
pp. 382-401, July 1982
- [Levine 76] Len Levine & Ware Meyers,
"Semiconductor Memory Reliability with Error Detecting and Correcting Codes",
IEEE COMPUTER, Vol. pp. 43-49 , October 1976
- [Moore 84] W.R. Moore, N.A. Haynes,
"A review of synchronisation and matching in fault-tolerant systems",
IEE Proceedings, Vol. 131, Pt.E, No.4, pp. 119-124, July 1984
- [Pease 80] M. Pease, R. Shostak, & L. Lamport,
"Reaching Agreement in the presence of faults",
Journal of the Association for Computing Machinery, Vol.27, No.2, pp. 228-234 , April 1980
- [Smith 81] T.B. Smith,
"Fault-Tolerant Clocking Scheme",
FTCS-11, 11th Int. Symp. on Fault-Tolerant Computing, Portland, pp. 262-264, June 1981
- [Smith 84] T.B. Smith,
"Fault-Tolerant Processor Concepts and Operation",
FTCS-14, 14th Int. Symp. on Fault-Tolerant Computing, Orlando, pp. 158-163, June 1984

- [Wensley 83] J.H. Wensley,
"Industrial-Control System does Things in Three for Safety",
Electronics, pp 98-102, January 27, 1983
- [Wensley 83a] J.H. Wensley,
"An Operating System for a TMR Fault-Tolerant System",
FTCS-13, 13th Int. Symp. on Fault-Tolerant Computing, Milano, pp. 452-455 , June 1983
- [Wensley 78] J.H. Wensley et al.,
"SIFT: The Design and Analysis of a Fault-Tolerant Computer for Aircraft Control",
Proceedings of the IEEE, vol. 66, no. 10, pp. 1240-1255, October 1978

5 Recovery to previous state

5.1 Principles of recovery

In the previous Chapters, we considered the masking of faults by coding, massive redundancy, voting and hot-spares. Masking ensures that no fault leaks to the external world and that the duration of the outage is essentially equal to zero. The conditions to achieve masking are total error coverage and a close synchronism between replicated units.

In some cases, such a close synchronization is not feasible, for instance when the units are distributed over a wide area and interconnected by a network. A loose synchronization can also be desired to reduce the coupling of the units and therefore avoid common modes of failure. Further, the amount of hardware redundancy of masking systems is high - at least a triplication of the computing elements is required in addition to special elements like voters and self-checking comparators. And finally, the masking methods cannot deal easily with software faults, since diverse (replicated) software is costly and since the nature of software faults makes total error coverage unlikely. This is indeed the Achille's heel of masking systems: errors which manage to leak out of them cannot be corrected anymore.

The alternate to masking redundancy is **recovery**, also called **dynamic redundancy**. Recovery techniques intend to correct errors rather than masking them.

The basic strategy of recovery after an error is detected consist of:

- _ Reloading the lost storage parts,
- _ Undoing all effects of the fault in the storage which survives and
- _ Continuing computation from that state.

Operations that the failed unit may have executed before it crashed and especially its interactions with the plant must also be considered.

The two technique of recovery are **compensation** of errors and **retry** of a failed operation:

- _ **Retry** (backward error recovery) is quite a natural technique for transmission links: if a transmission fails, for instance because a message is lost or corrupted, then retransmission is attempted on the same hardware. Practically all protocols in data networks use retry as a fault-tolerance mechanism. Retry is particularly simple in transmission lines because these devices do not have memory: retry uses only time redundancy. When the element has a memory, such as a computer or a data base, retry requires that the element be restored to a previous state before repeating the operation. This restoring requires that a copy of the previous state have been saved in anticipation of a failure.
- _ **Compensation** (forward error recovery) requires active correction of the errors and their consequences by a compensation program. Compensation is a common practice in book-keeping for instance, where wrong entries may not be erased, but must be compensated for in the opposite column by a storno entry. This is an intelligent process that is application dependent. Compensation is only possible for software errors, as the extent of the damage must be known.

Fault tolerance by recovery techniques is attractive for three main reasons:

Recovery allows continuing the operations on the same machine when the error is transient. This is interesting since about 90% of the errors are transient. The amount of redundant hardware is modest, especially when compared with TMR systems.

Recovery is also helpful when error coverage is not complete, that is, when wrong data can leak out of a processor. This is often the case when dealing with software errors.

Recovery applies to typical problems of consistency in databases as a consequence of voluntary and expected cancels. Here, the cancel may be due to the peculiarity of a consistency algorithm, for instance to remove deadlocks or version conflicts.

On the other hand, recovery has some disadvantages with respect to masking:

1. Recovery is conceptually more complicated than masking, whose only real challenge is synchronisation.

2. Recovery diverts computing power at run-time to save periodically the execution state in prevision of a failure. Although this saving is in principle only necessary for retry, compensation also requires a minimum of valid state.
3. Recovery involves a certain disruption of function. During that disruption, interaction with the environment could take place. It is therefore difficult to realize hot spares (spares which are inserted within negligible time) with recovery. Recovery is used for warm sparing (short loss of function) and most of the time for cold sparing (relatively long loss of function and amnesia about interactions).
4. Recovery techniques can be considered as being one abstraction level on top of masking techniques, which are essentially hardware methods. Recovery techniques are used at different levels in the hierarchy of a computer, beginning at levels near to the hardware (instruction retry) up to high software levels within the application (database recovery). In fact, masking and recovery appear as complementary rather than competing techniques, especially in the sense of multiple lines of defence.

For instance, volatile CPU errors and page faults can be handled by retry, the whole CPU can be built as a masking, dual work-by system and the database built on this computer can be protected by recovery methods. Finally compensation methods are the high school of recovery: a meaningful compensation requires a good knowledge of the application.

The topic of recovery is divided into several sections:

1. Basic architectures, warm stand-by and cold stand-by
2. Basic scenario of recovery: forward and backward recovery
3. Methods used for state saving and restoring (Chapter 6)
4. Recovery in databases (Chapter 7)

5.2 Recoverable architectures

Recovery relies on only one working unit. Redundancy for recovery comes in form of warm or cold STAND-BY units, i.e. units which do not participate in the actual computations, but which can be inserted within a reasonable time to replace the failed unit. In the meantime, the stand-by unit can contribute to performance by doing some other job while the system is error-free.

In contrast to work-by, which maintains the (hot) spare actualised by replicating the computation, stand-by maintains the spare actualised by transferring parts or totality of the state of the (unique) working unit at regular intervals in a safe place. When an error occurs, this state is loaded into the spare, which is connected in place of the working unit and resumes the interrupted computations.

The actualisation of the spare is not a continuous process: the saving of the working state takes place at determined places in the execution of the program, which are called **save-points**. The save-points cannot be put arbitrarily close to one another since the communication overhead would be too large if every modification of the state of the working unit would be communicated to the spare. This necessarily causes the state of the spare to lag behind the worker's state by at most one save-point.

The two main stand-by techniques, warm stand-by and cold stand-by are described in the next subsections. In both cases, one assumes that the working unit and the spare or back-up storage are fail-independent, i.e. it is unlikely that both would fail at the same time.

5.2.1 Warm stand-by

Warm stand-by, also called **main/back-up**, or **primary/secondary** appears to have the same architecture as dual work-by. Both use a second unit that is functionally identical to the working unit. The difference is that the stand-by spare is not performing the same computations as the working unit, but can do some other useful work. One can however only take advantage of this free computing power when graceful degradation is feasible.

The working unit keeps the spare actualised by copying to it its current state at each save-point, over a data channel, sometimes called "update bus". The memory of the spare unit serves as a repository for the state of the working unit. Therefore, the spare is already loaded at recovery time and it is immediately available. In addition, the spare can monitor the activity of the working unit to track possible interactions of the working unit with the plant since the last save-point. This is symbolized by the dotted lines in Figure 5-1:

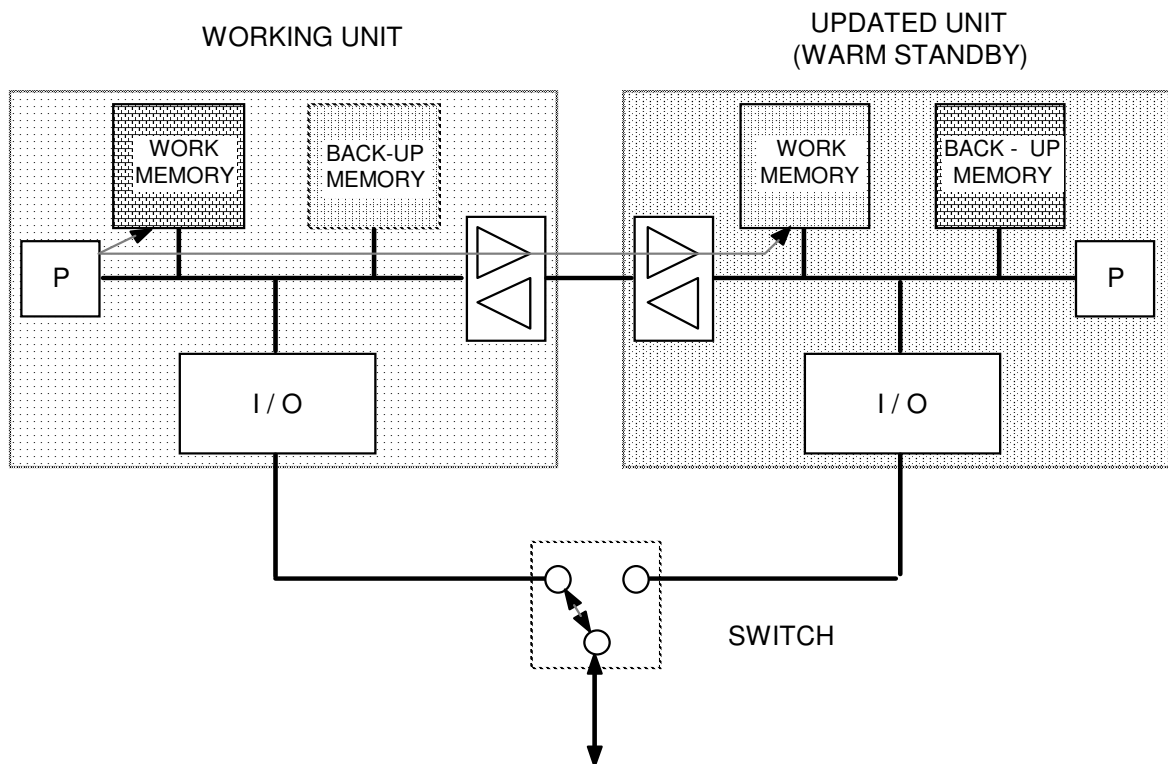


Figure 5-1: Warm Stand-by.

In Figure 5-1, the stand-by unit has conceptually two storages: one for its current tasks, and one as a back-up for the working storage of the working unit. The same is true for the storages of the working unit, only that the back-up storage of the working unit is normally unused. It is only there in case the roles of working and stand-by unit are inverted.

Warm stand-by has the advantage that the second unit is free to perform other tasks, except for that small portion of its computing power that it uses for keeping itself actualised. This actualisation function can even be done by a dedicated device like a DMA-controller at little expense. This is symbolized in Figure 5-1 by the link unit. Thus, warm stand-by increases the overall computing power, provided that graceful degradation is possible.

Example : the TANDEM-16 computer uses warm stand-by. The architecture consists of several processors interconnected by a dual, high-speed bus called Dynabus (Figure 5-2).

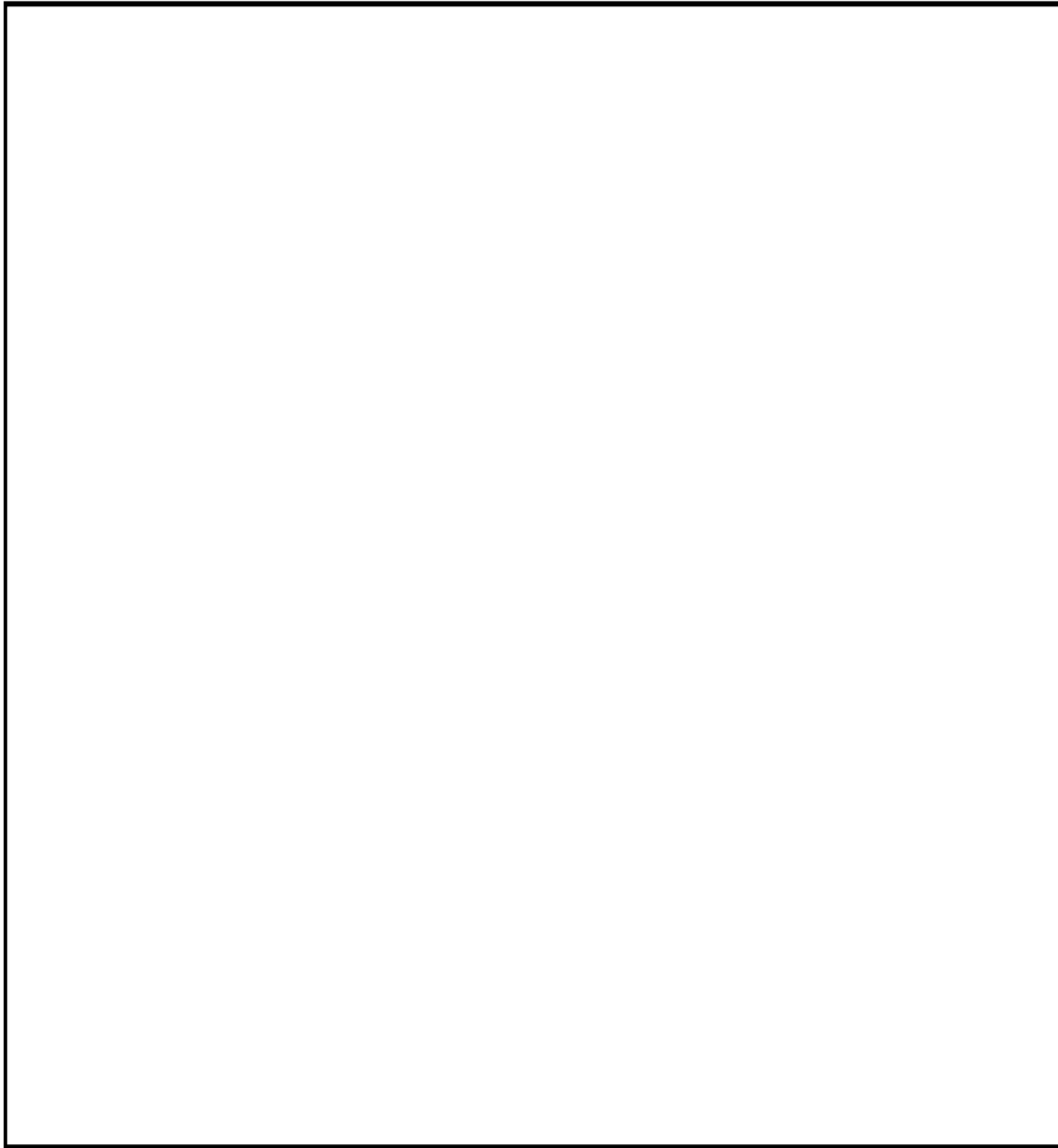


Figure 5-2: Tandem-16 Computer

For warm stand-by, it is sufficient to consider a pair of nodes. A task performed in one node possesses a back-up copy of its state in another node. The changes to the working task's state are communicated to the stand-by over the Dynabus. The copies are triggered by special "checkpoint" instructions, which are inserted by the programmer in each task. At each save-point, the processor copies the current state of the running task, including stack, heap and processor registers (processor context) to the back-up task's memory space over the Dynabus.

Upon detection of an error in the working unit, control is passed to the stand-by unit which continues computations. It first runs a special program prepared beforehand, which among other tasks will release locked resources. Then it continues the computations based on the last state saved. For this, the peripherals are dual-ported so they can be accessed from either node. The former working unit is taken off-line, checked and repaired if necessary, and then reinserted as a stand-by.

Although the warm spare is ready to take over at any time, it requires a short, but non-negligible outage time. Further, some state saving methods introduce additional delays to reduce storage requirements or for errors confinement. The result is a **recovery gap**.

The recovery gap can last some 10 s in a typical system. This may be insufficient for time critical applications, but it is in most cases acceptable for benign plants and for human interaction. The real problem is that the spare may be unaware of commands sent or received signals since the last save point. Therefore, it will be necessary for the

spare to also track the interaction of the working unit with its environment. We will come back to this interaction problems in the next Chapter.

5.2.2 Cold Stand-by

In cold stand-by, the state of the working unit is regularly saved to **back-up storage**, which is essentially passive (e.g. a disk or a tape). By contrast, the state in warm stand-by was saved directly into the memory of the spare unit, which was then ready to take over with little delay. A cold spare has no relevant internal state, i.e., the spare is not actualised before insertion. One reason for this could be that the spare is taken from the repair shop, or from a pool of non-dedicated spares. A more frequent case, common in computing centres, is that there is no actual spare: the faulty computer must be taken off-line, repaired and reinstalled. Since these operations void the computer's memory, it must first be reloaded from the back-up storage before continuing.

Although the hardware is not duplicated, the minimum requirement is that the storage must be replicated as shown in Figure 5-3. As we shall see in the next Chapter, approximately a threefold replication of storage is required.

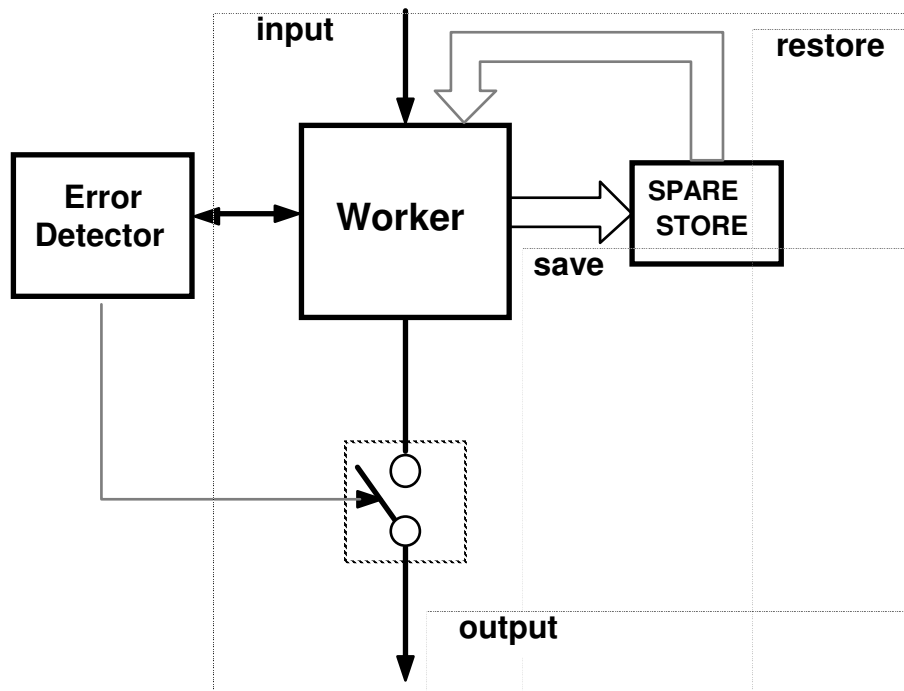


Figure 5-3: Cold Stand-by

Upon detection of an error in the working unit, the cold spare or the repaired working unit is inserted and then reloaded with the last relevant state which was saved in the back-up storage.

Cold stand-by is used at different levels in the hierarchy, from the bottom hardware levels up to complex software levels:

- Some processors implement **instruction retry** for redoing failed computations (IBM 370, UNIVAC 1100). All activity of the CPU is logged and the storage is assumed not to fail. In case of CPU error, the CPU is restored to its previous state with the help of a log of the CPU's activity and the instruction is restarted. The technique is similar to the handling of page faults in microprocessors which support virtual memory: when the memory address is not present in physical memory, these processors interrupt the failed instruction, start a routine to read the missing page from memory and redo the interrupted instruction. Such techniques are implemented in all major 32-bit processors. Instruction retry uses the same mechanism, with the difference that the instruction is only re-executed. An instruction retry takes some ten microseconds.
- If the contents of the semiconductor memory are lost, but its hardware remains intact, for instance in case of transient error or operating system crash, the computer can reboot by itself using state information saved on disk. For this it may need to cancel the task it was currently executing and to correct information already written on disk. This recovery takes less than one minute.
- If the computer is damaged, then the computations could be kept on in a cold spare unit which must first be connected and loaded. This form of recovery takes some minutes, and, if off-line repair must be performed, it can take some hours.

- The worst case is that the whole computer and its disks is lost, for instance in a fire or a severe operating system crash. Little of the computation is lost if the state of the installation has been regularly saved on tape. This operation is typically done once a week, so at most one week of work is lost. Several techniques allow to reduce this interval by continuous back-up. Recovery takes about one hour.

The duration of cold stand-by depends on the extension of the damage. The problem of cold stand-by is not so much the duration of the outage itself but the fact that a cold spare is not capable of monitoring the working unit's interaction with the environment since it is either completely passive or not existant.

Besides the relatively long down time, the main problem of cold stand-by is that interactions done by the working unit between its last save point and the moment it failed escape to logging. These aspects will be investigated in the next Chapter.

5.2.3 Other Architectures

The state actualisation technique developed for stand-by can also be applied to on-line repair of work-by systems: Once a working unit fails, is outvoted or switched off, the system works with reduced redundancy. If the former working unit can overcome its fault (for instance because the fault was only transient) then it can re-join the system and serve as hot-spare again. To do this, the spare must first be actualised and synchronized to the same instruction as the other unit(s). This operation must also be performed if the on-line repair is done by hand. The techniques used for actualisation are basically the same as for warm stand-by.

5.3 Compensation (Forward Error Recovery)

Compensation tries to bring the system from the faulty state it is in to an error-free state. It assumes that the hardware is still intact and a part of the state is still sound. A special program tries to manage the situation and bring the task's state to a point from where the computation can be resumed. This involves undoing the uncorrect and possibly also the correct, but inconsistent effects of previous computations, redoing computations which have not yet be done, and doing other meaningful computations until an error-free state is reached from where normal computations can be resumed. Errors that cannot be corrected are compensated for. Compensation is related to the techniques of exception handling. In particular, the termination of a faulty task, releasing of all its resources and updating of the system tables is a typical forward recovery technique used today in all multi-user operating systems.

Figure 5-4 shows a typical execution of a forward error recovery:

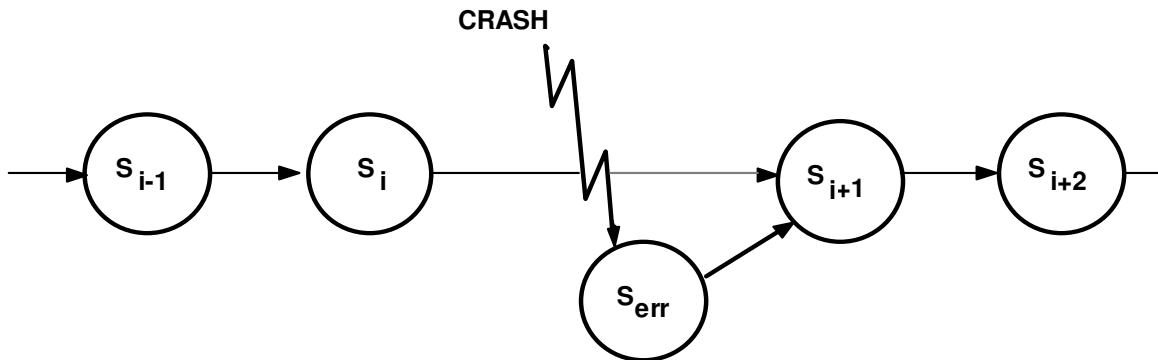


Figure 5-4: Forward Error Recovery.

Although forward error recovery in principle does not need to restore a previous state, it would be unthinkable to restart the tasks from scratch. Therefore, forward error recovery relies on some storage which contains information about the progress of the task(s) which crashed. In general, forward error recovery requires an extensive knowledge of the current state, especially about the possible extension of damage, and intelligence to appreciate which steps must be undertaken to bring the uncompleted action to an end. Therefore, the forward error recovery technique works best when dealing with anticipated errors such as exception conditions in a program. We will devote little attention to forward error recovery since it is very much application dependent and lacks an accepted theory [Mili 85].

5.4 Retry (Backward Error Recovery)

Retry attempts to restore an error-free state which prevailed at some point earlier in time, and to resume the computations from that point on, using the same program as the one which failed. Retry consists of two distinct operations, rollback and roll-ahead.

- **rollback** consists in using previously stored information to restore a known-good state which prevailed at some earlier point in time, called the **recovery state**. This state can be restored by **reloading** lost storage and **undoing** the inconsistent parts of the surviving storage.
- **roll-ahead** consists in restarting the computations from that restored state, while considering possible actions which have occurred since the detection of the error (Note that roll-ahead has nothing to do with forward error recovery)

A trivial recovery state is the reset state of the computer. This is called a **cold-start**. Cold start is generally not acceptable because all computations done since the cold-start are lost.

To achieve a minimum loss in computations, the program is divided in **recovery intervals** separated by save-points, which play the role of firewalls in time. At each save point, the current state of computation is saved in a safe place. When an error occurs, the rollback procedure will use this information to restore the state which prevailed at the save point. The restored state is called a **recovery point**.

One also finds the terms of "recovery point", "retry point" or "checkpoint" in place of save point. The difference is that a retry point is only established at recovery time while a save-point is an action taken repeatedly at run-time. It may further be necessary to drop some save points and restart from a retry point which corresponds to an earlier save point, or even from a state which did not exist before. This is why one must distinguish between retry points and save-points. "Recovery point" is used in both the sense of "retry point" and of "save point" depending on the author.

The distance between save points is a trade off between computation loss and recovery time. The closer the save points, the less computations are lost in case of crash and the faster recovery will be. On the other hand, frequent state saving steals a substantial part of the computing power. Further, we will see that the save points should not be inserted at random: the clever insertion of save points in a program can substantially reduce the amount of time and hardware required for state saving and restoring. The recovery points should divide the program into **repeatable sequences**.

The following Figure 5-5 shows the typical execution of a backward error recovery:

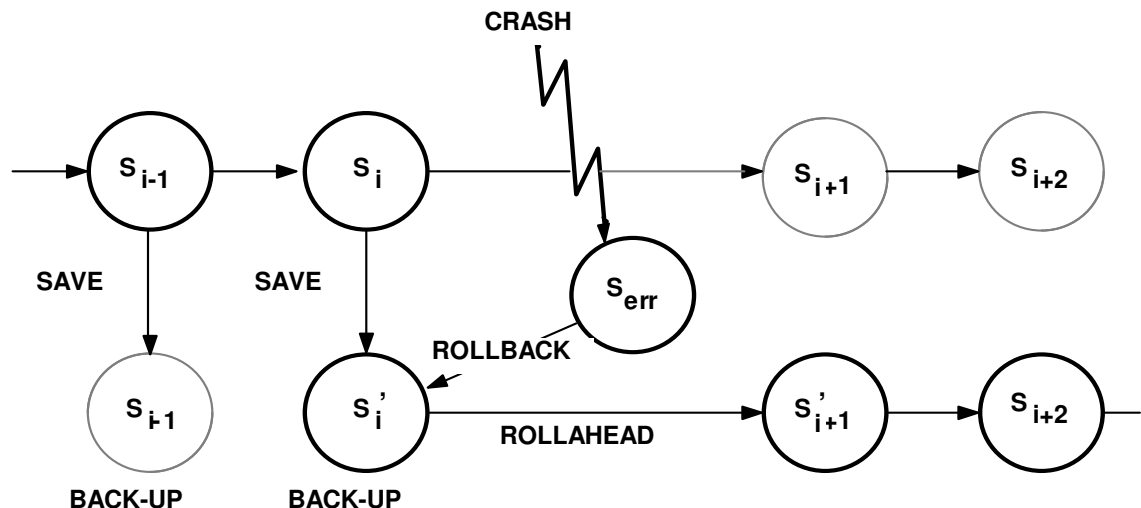


Fig. 5-5: Backward Error Recovery (Retry)

The probability of a second failure during recovery cannot be neglected. In fact, it is highly probable since hardware errors tend to show up in bursts. Therefore, both the operations of rollback and of roll-ahead should be repeatable at will, with the same results, even if they are interrupted by a second failure. One says that such operations are **IDEMPOTENT**.

Most of the work of recovery is to make repeatable, idempotent operations out of operations which are non-repeatable by nature.

5.5 Scenario of retry

5.5.1 Recovery Script

We shall consider a general scenario of retry, which applies to both warm and cold stand-by (Figure 5-6):

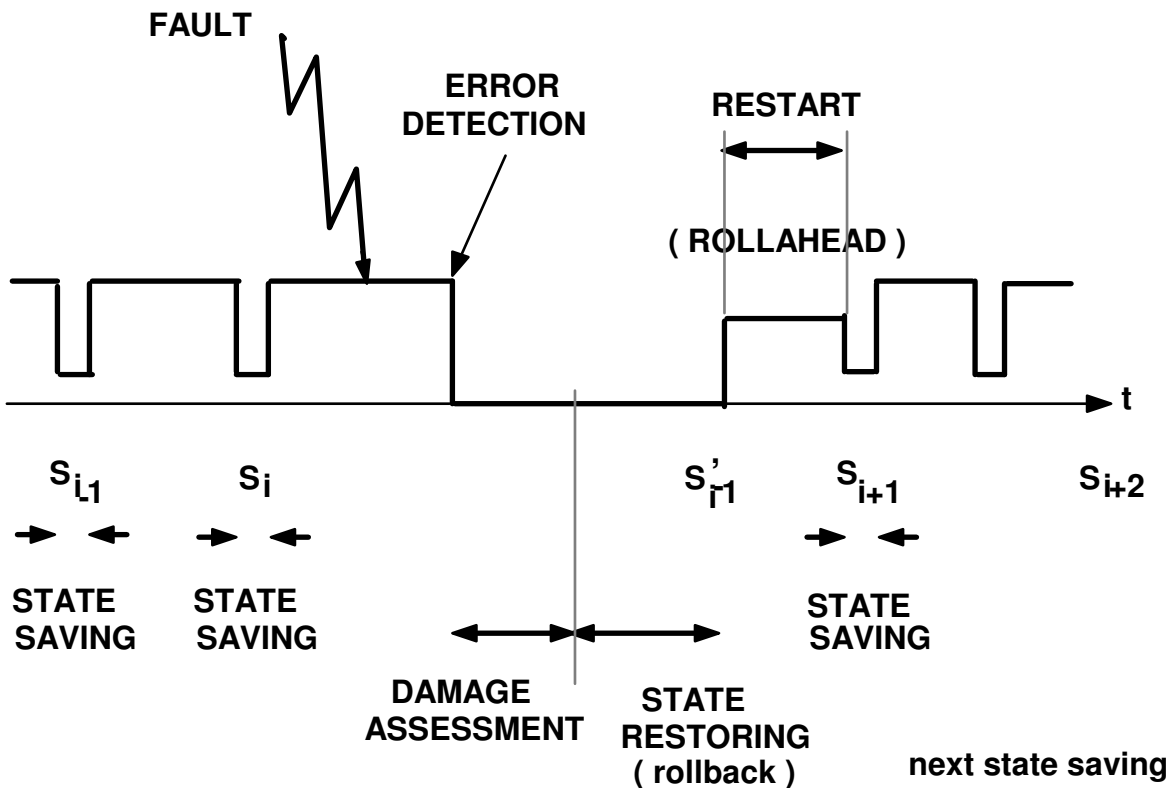


Figure 5-6: Phases During Recovery

The phases of recovery are:

- state saving
- error detection and damage assessment
- restoration of a valid state
- restarting computation

5.5.2 State Saving

During normal operation, the working unit's state must be regularly saved to back-up storage in prevision of an error. The information saved at each save-point depends on which parts of the computer are likely to fail. One distinguishes between:

- **Volatile storage:** its content cannot be trusted after a crash and is considered as lost (e.g. the processor's registers). In prevision of loss of storage, the actual state of the modified variables must be saved at each save point.
- **Stable storage:** its content remain the same after a crash, but could be inconsistent, perhaps because of unfinished operations (e.g. the database). In prevision of storage correction, the value of variables as they were before the modification took place are saved in an UNDO LOG at each save point.
- **Environment:** it is not affected by the crash but may interact with the computer during recovery in an undesired way, for instance by sending data which are not received or receiving uncontrolled commands. In prevision of interaction with the environment, all activities are recorded at each **log point** in a **journal**. A log point should be taken for each interaction between two save points.

How this is done and how the state is restored is described in detail in the next Chapter.

It is important that the storage in which the state and the logs are recorded is stable, i.e. survives a crash. For this, it is sufficient that the working storage and back-up storage be **not decay-related**, i.e. one assumes that there is no failure mode that could let them fail both at the same time.

- In the case of warm stand-by, the back-up storage is the memory of the spare, which is physically independent.
- In the case of cold stand-by, the back-up storage is some kind of external storage, for instance a disk or a non-volatile RAM memory;

In addition to state saving, warm-stand-by requires the stand-by unit to monitor the execution of the working unit, as we shall see. With the rest of the computing power, the spare can perform other tasks, if any.

The information saved at each save point depends in the first place on which storage of the machine is considered as volatile, which is considered as stable and which belongs to the environment

5.5.3 Error Detection for Recovery

Error detection is done by one of the methods explained in Chapter 3. We consider three possible situations, depending on the error latency:

the error has been detected before it left the confinement zone. This assumption can be enforced by fail-stop units.

the error has left the confinement zone and corrupted storage, but the corresponding operations have been recorded in a log and can be still undone. An error can still be corrected if it leaks only to stable storage, but not if it managed to leak to the environment, as long as one can keep track of it.

the error has corrupted the outer world beyond the possibilities of repair. This is an unrecoverable case which we consider equivalent to a mission failure: it requires human intervention for correction. In some cases, it can be handled automatically by forward error recovery, which is of course application-dependent.

5.5.4 Rollback

After detection of an error, the rollback procedure is invoked. Rollback uses the information which has been saved during normal operation at each save point to restore the recovery point. Restoration is implemented in different ways, depending on which state must be reconstructed:

- The volatile state, which is lost in a crash, is reconstructed by reloading the hardware with the back-up copy taken at the last recovery point. This copy has the same size as the physical storage.

In warm stand-by, the volatile state is already loaded in the stand-by unit and the computations can be continued on that unit. Because of delays in transmission and buffering, it may be necessary to update the spare with the latest data still waiting in a queue (redo operation).

In cold stand-by, the back-up state is loaded into a spare (or into the repaired working unit). Here also, some redoing may be required in order to save storage.

- The stable part of the storage, (that which survived the crash), may still need correction. This is done by undoing the last, inconsistent operations. Undoing assumes that the current state is erroneous, but that the errors can be corrected. For undoing, one must save the value of the variables before modifying them (save-old-value). The UNDO phase selectively restores the modified variables to their previous value until the recovery point is reached.
- The external world is difficult to restore to a previous state. Recovery depends very much on the particular application. In some cases, given commands that have not yet been executed can be cancelled. In others cases, executed commands can be compensated by other commands, but it is quite difficult to command a machine to roll back to a previous state.

The rollback may be **perfect**, in the sense that the unit can be returned exactly to the same state as prevailed at the last save point, or it may be **imperfect**. Of course, the law of physics prohibits such a thing as perfect rollback: all physical processes are irreversible and time cannot be rolled back. So, a weakened definition calls "perfect rollback" a restoring of all information to their previous value, such that repeating the execution from that point would yield exactly the same result as the first execution. Perfect rollback is relatively simple to achieve as long as there have been no interaction of the computer with the environment. For instance, this is a common assumption

made in the case of instruction retry, where the operations take place in the closed world between processor and storage. But the very presence of input/output devices make perfect rollback questionable.

Rollback restores the system: volatile storage, stable storage and environment to a state they had before the fault. Perfect rollback can never be achieved, but only approximated.

5.5.5 Roll-ahead

After a starting state has been established either by reloading or undoing, computations can be resumed from that state on. These computations are however not necessarily identical to the computation which has been interrupted by the fault. Roll-ahead must take into account what the rollback could not undo. We distinguish two cases:

- If rollback was perfect, roll-ahead can consist simply in continuing computations from the recovery point with the same program, without special precautions. The execution can take the same path, but with different results and outputs, or even take a completely different path. In the latter case, the difference with forward recovery (compensation) is not great. Indeed, when a task is cancelled, for instance because of a software error or deadlock break, it is not desired that this task be executed again, so the second execution can take a different path or even not take place at all.
- If rollback was imperfect, then roll-ahead must consider actions taken by the working unit between the recovery point and the present time, since rollback could not cancel them. The script for roll-ahead is the same program which has been interrupted by the fault, except for all the interactions with parts which could not be rolled back. Roll-ahead must try especially not to duplicate outputs already done by the crashed task, and not to ask again for inputs that were already sent. Roll-ahead consists of a careful repetition of the former computation. It is complete when the program reaches the next recovery point that the failed execution was not able to reach.

Even if rollback is not perfect, roll-ahead would be no problem if every action could be repeated at will with the same effect, i.e. if every action could be IDEMPOTENT.

Performing a write operation to memory or a read operation from memory is basically idempotent.

Most operations, even within a computer, are not idempotent. Examples are:

- A read-modify-write (for instance incrementing an index)
- Printing of a cheque or a bill.
- A command to a stepper motor control or another non-reversible device.
- A read from file or from a network buffer

Unfortunately, most actions with the environment are not idempotent. Roll-ahead can deal with this by not repeating output or input actions already done and by doing every action the failed program did not manage to do. This requires a **perfect tracking** of the interaction between the computer and its environment. A perfect tracking must be done by an independent unit, such as the stand-by unit, but cannot be provided by cold stand-by.

Thus, idempotency depends both on the environment and on the kind of operation. Especially, it depends on the placement of the recovery points, as we shall see in the next Chapter.

Roll-ahead is a careful re-execution of the failed program which considers the imperfections of rollback, and tries to avoid loss or duplication of interactions.

We will consider these aspects in more details in the next Chapter, and especially distinguish the techniques for state saving and restoring.

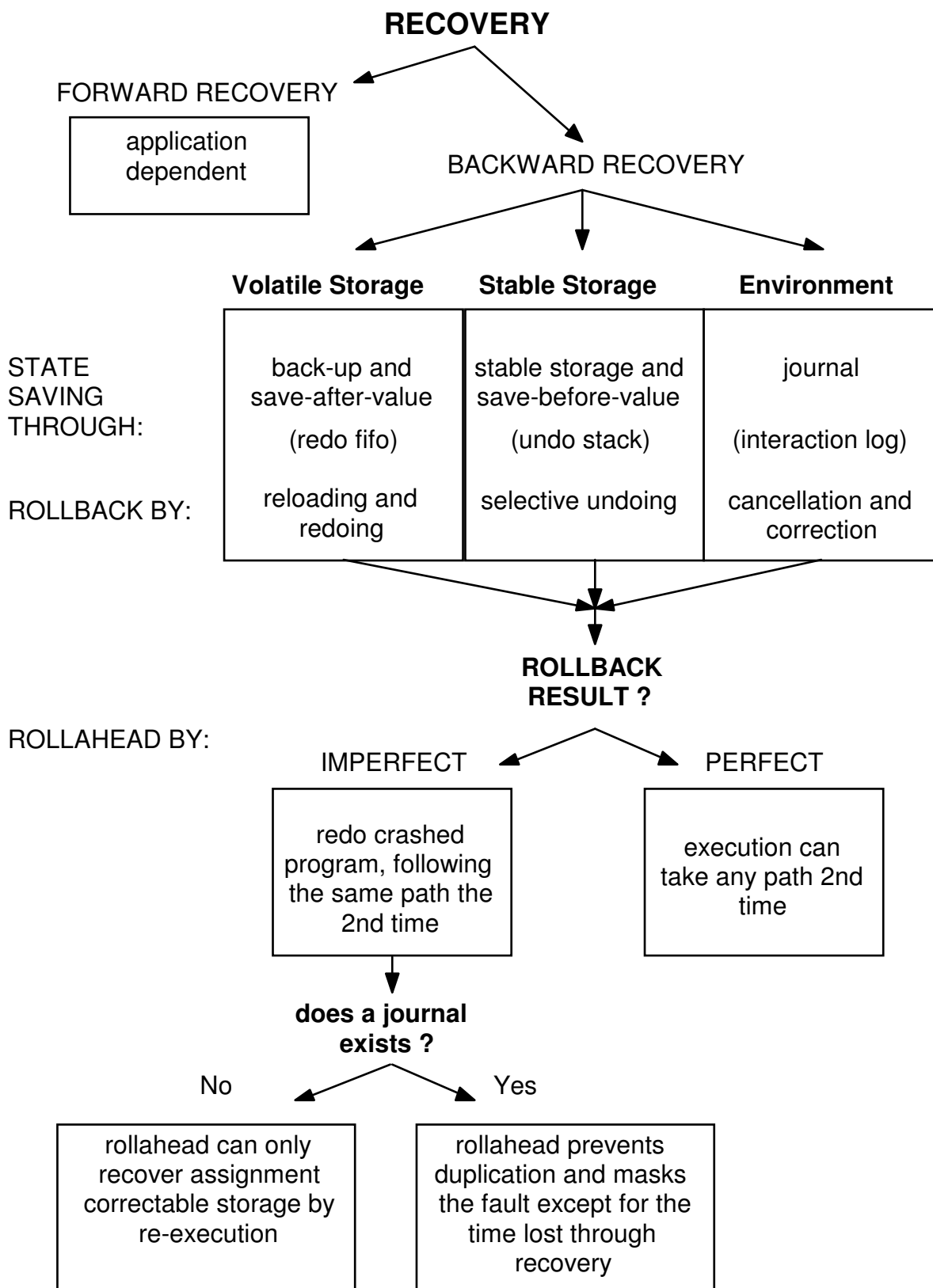
5.6 Summary of the recovery techniques

Backward error recovery is only a conceptual view: backward error recovery is a form of forward error recovery in which the extension of damage extends to the whole state. The recovery strategy consists of using the original program to repeat the operation. Whether we consider it to be forward or backward recovery depends on whether we look at the recovery process as done by an external entity or by the computing system itself.

On the other hand, forward recovery by efficient exception handlers requires that the system be rolled back to a defined point in execution to take appropriate actions, and backward error recovery must take into account the irreversible real world.

So, concepts like rollback and roll-ahead provide a general framework, to which it will be necessary to append exceptions. We will in the following Chapters discuss backward error recovery and leave forward error recovery as a technique to deal with anticipated software faults.

We summarize the recovery techniques in the following diagram, which will be developed further in Chapters 6 and 7.



5.7 References

- [Mili 85] A. Mili
 "Towards a Theory of Forward Error Recovery",
 IEEE Transactions on Software Engineering, Vol SE-11, No. 8, pp. 735-748, August 1985

6 State saving and restoring

In this chapter, we will review in detail the techniques used to save the state of a computation and to restore it for backward error recovery. These techniques apply to all stand-by (backed-up) systems.

The problem we address here is how to take save points, how to perform rollback, i.e. restore a computer to a previous state, and what should be done when continuing computations at roll-ahead.

First we define a model of computation, and what a computation state is. This will allow us to estimate the information quantity which should be saved at each save point.

Then, the hierarchy of storage is defined by dividing the state of a computation into a volatile state, a stable state and the environment. The corresponding three basic techniques for rollback and roll-ahead are discussed:

- FULL BACK-UP and SAVE-AFTER-VALUE (SA) for recovery of the volatile state
- STABLE STORAGE and SAVE-BEFORE-VALUE (SB) for correction of the stable state
- INTERACTION LOG for dealing with interactions between the computer and its environment.

6.1 A model of computation

Retry or Backward Error Recovery relies on rollback, i.e. on means to restore a previous, known good state on the same or on a different hardware after an error occurs. To achieve this, it is necessary to save the current state at regular intervals at each save point. Before discussing how save points are taken, it is necessary to understand what the state of a task is.

6.1.1 Task State

We consider the simplified model of a computer consisting of processing elements (PU) which do not contain memory, and of storage which contains the state of the task(s) in execution.

To this purpose, the PU's registers are considered as separate from the computing function of a PU. The code of the program executed by the PU is not part of the task's state. In fact the program could be cast into a PROM or a ROM, or be part of the instruction set. We will assume that the program code is not modifiable and therefore disregard it.

The inputs and outputs must be modelled separately. The outputs can be considered as being that part of the state that is made visible to the outside, the inputs are that part of the state of another machine that has an influence on the machine considered.

The **task state** (or state variables) is the set of data that defines completely the progress of a task at a given moment [Horning 73].

The execution of a task can be seen as a sequence of **actions**, which bring the task state S_i to another state S_{i+1} (Figure 6-1):

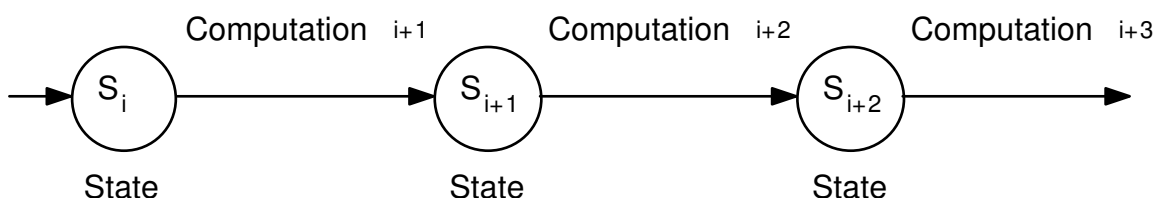


Fig. 6-1: Computation as a Sequence of States.

Even the execution of a No-Operation instruction is an action: it increments the program counter. The operation performed on the state is specified by an instruction, and may additionally depend on the value of the inputs.

The abstract notion of "task state" in the theory of computation becomes a concrete meaning in fault-tolerant computers: if we could interrupt a computation at any given time and completely void the memory and reset the processor, the task state would be defined as the set of information we need to reload into memory and processor

in order to continue the interrupted computation at a later time exactly as if there had been no interruption (except for some time lag).

Recovery mechanisms require that the state of a task be saved at defined points in the execution, in order to resume it later. Therefore, it is important to determine the amount of information that belongs to the state and must be saved.

Obviously, the state is fully defined by the contents of the disks, semiconductor memory and by the registers of the processor and of its I/O devices. But only a subset of this storage is required to define the current state.

At any time, the state of a task consists of a **relevant** part, which must be saved, and an **irrelevant** part, which needs not be saved to reconstruct the total state. The amount of relevant state information is a function of the level of nesting in which the task finds itself, i.e. it is a function of the progress of execution.

Let us begin at the rock-bottom of the machine: if a processor would be interrupted at any point in the middle of a processor instruction, the amount of storage one would need to save to proceed from that point on would be very high: the relevant state would consist of all internal registers, even those which are only accessible to the micro-program. Even the temporary and dynamic storages should be saved; Even of the exact state of the electrical pulses along the signal lines would belong to the state. It is obviously not possible to save such a large quantity of information.

One can get a feeling of this by considering the interrupt mechanism in current processors: an interrupt causes the processor to be interrupted after it finished an instruction. The state of the processor that is saved at interrupt time consists at the minimum of two words: the PC (program counter) and the PSW (processor status word).

Processors that support virtual memories allow interruption in the middle of an instruction in case of page fault. These processors, like the MC 68010, have to save a larger amount of internal information than they do for normal interrupt handling, since interrupts are recognized at the end of an instruction only and do not affect the micro-program registers. The 68010 saves in case of page fault no less than 29 words, which include internal registers normally not visible to the programmer.

At a higher abstraction level, a procedure can be seen as a complex instruction. During execution of a procedure, its local variables clearly belong to the state. Once the procedure is exited, these local variables are not defined anymore and can be forgotten as any other off-the-stack information.

As instructions become more complex, the amount of relevant state within an instruction increases, and conversely, less state is relevant between two instructions. This is true of all interpreters: if the task is doing interpretation, for instance interpreting a program in Basic, its state would be minimum between two statements, since most internal variables of the interpreter are irrelevant between two statements.

At the highest level of granularity, a task can be considered to be a very complex instruction of an interpreter that brings the file system from one state to another; the memory belongs to the irrelevant state. For instance, a text editor is mapping the previous state of a text file to a new state. Then, the relevant state consists only of the state of the files after or before an execution of the task. In case of crash, the state of the main memory is considered irrelevant; tasks that were in progress at the moment of crash will have to be restarted. This view is taken by most transaction systems: the relevant state is on disk, the content of the processor and memory can be lost and are irrelevant.

Finally, a computing system can be viewed as a machine that generates its outputs based on its inputs and its internal state. The internal state is irrelevant as long as the machine does not communicate with the outer world.

The amount of state to be saved depends therefore on the dynamic structure of the execution, and more exactly on the level of abstraction considered: Therefore, the amount of information that belongs to a task's state at a given moment (and which should be saved to conserve that state) is a function of the dynamic nesting of the program. It is therefore also a function of time. Figure 6-2 shows the relevant state size as a function of time for procedure calls. The variations of context size during execution of instructions is shown as a kind of noise.

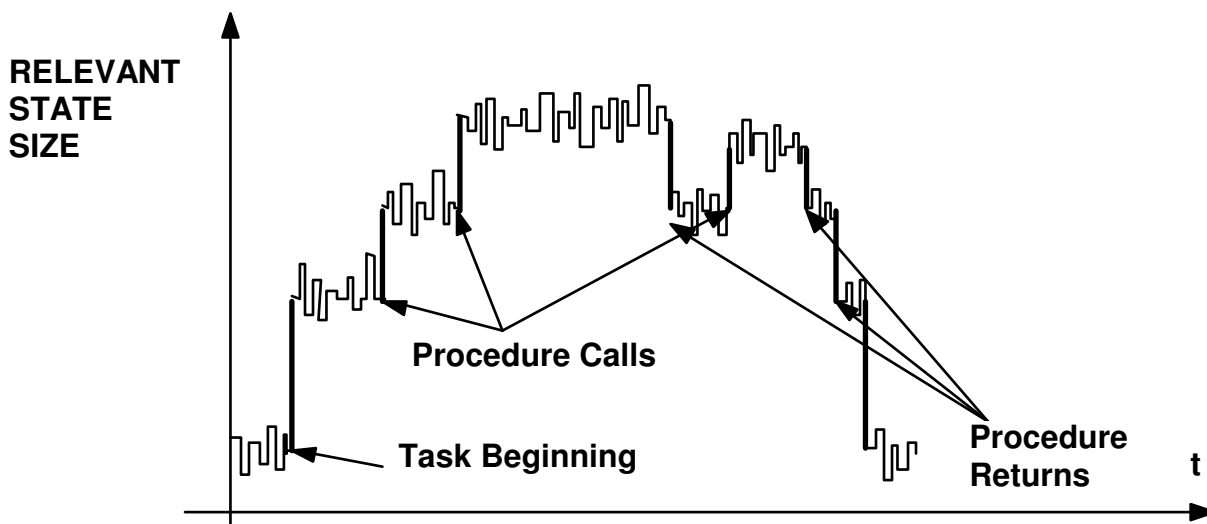


Fig. 6-2: Relevant State Information in Function of Time.

6.1.2 Taking Save Points

To minimize the amount of information to be saved, save points must be taken at carefully chosen places in the program.

- Taking a save point at any moment, for instance in the middle of an instruction, would require the saving of a exceedingly huge quantity of information, including the content of the micro-program registers.
- Taking a save point after each instruction would require saving all registers at each instruction. This is only feasible when one expects the instruction to fail with a high probability, for instance in anticipation of page faults.
- Taking save points when the execution returns to the main module, or when a task is finished will minimize the amount of state to be saved, since most of the variables are then irrelevant. As one increases the level of granularity, the more variables become irrelevant, and the less state must be saved.

If the save points are implanted at points which are optimal for minimum state information, the interval between the recovery points may become too large for them to be of any use. The recovery time could exceed the allocated grace time.

Further, one must also consider the interaction between processor, stable storage and environment in placing the save points. This aspect will be discussed along with the recovery techniques. We will see that it may be necessary to insert save points at places which are not optimum from the point of view of information transfer to maintain the state consistent.

A last aspect which must be considered in the insertion of save points is the application dependency: it is most of the time less costly to implant save points at regular intervals (for instance upon triggering by a clock) than to afford the additional programming burden associated with the optimum placing of save points by the programmer.

Examples: In the TANDEM 16 computer, the insertion of save points, called "checkpoints" is left to the application programmer, which must know when it is wise to take them. The COPRA computer also uses save points (called "retry points"), but these are automatically inserted by the compiler, and remain invisible to the application programmer

6.2 A model of storage

Before one can decide on the technique to use for state saving and restoration one must make assumptions about the damages that results from the crash.

The state of a task is kept in different storage parts in the computing system: registers of the processing unit, cache, RAM, disk and archival storages. Further, the state of the computer is not the only relevant state: in an embedded system, the state of the external world must be considered as being part of the task's state. We will classify the storage depending not on their physical location or technology, but on their dependability, as shown on Figure 6-3:

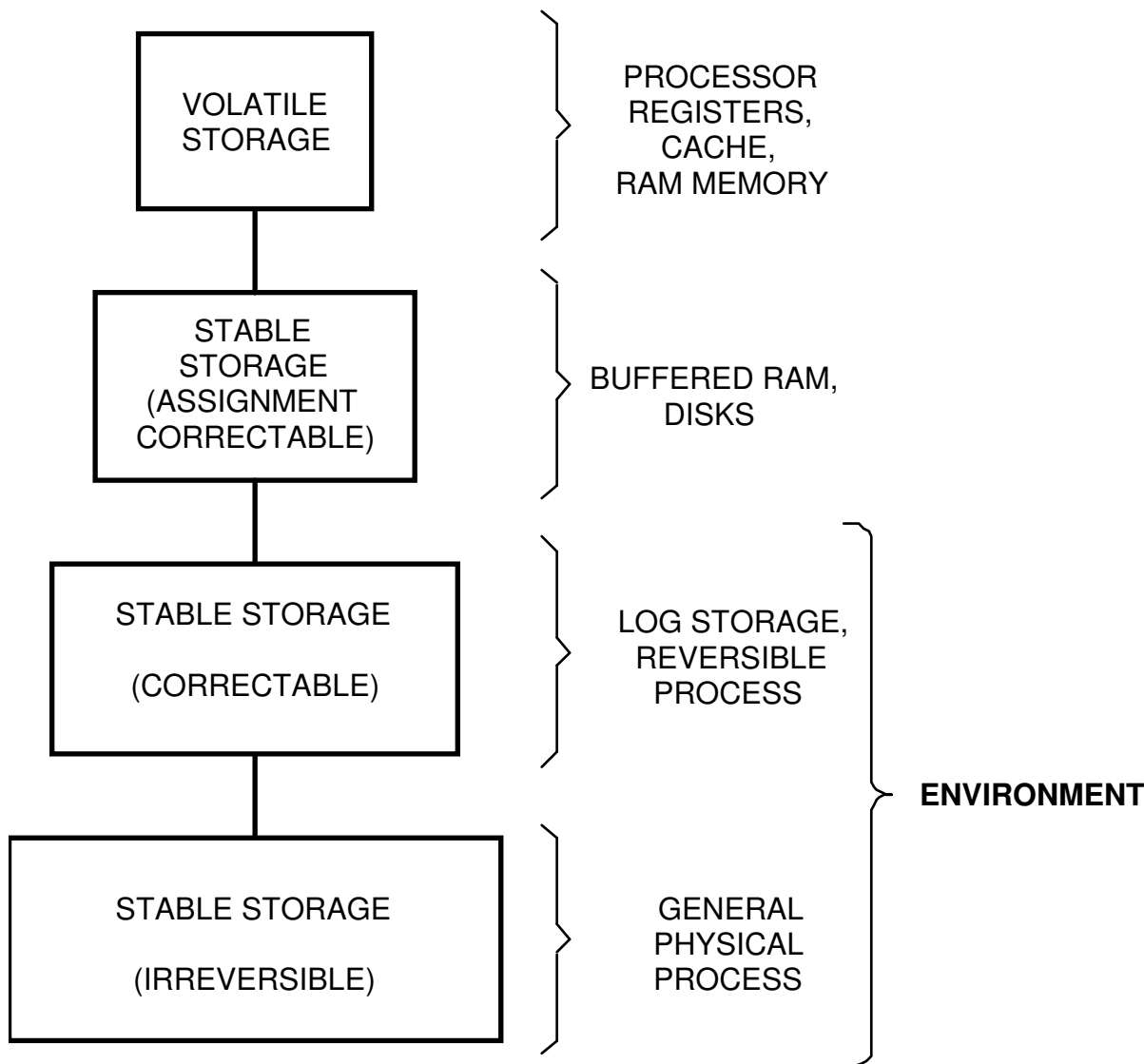


Fig. 6-3: Hierarchy of Storage with Respect to Recoverability.

- **volatile storage**: this is a part of the task's state which is lost or cannot be trusted anymore after a crash. A common assumption is to treat every unbuffered semiconductor storage (CPU's registers, cache, RAM) as volatile. Files which have been opened for writing them are often located in RAM should be treated as volatile. The CPU designers consider the CPU registers as belonging to volatile storage, and expect to be able to restore them if they are lost. We shall consider as volatile storage every part of the storage which is actually used for computations and which is not replicated.

To restore volatile storage, there must exist a **full copy** of its relevant parts in an independent storage.

- **stable storage**: stable storage is not lost in case of crash. Stable storage can be stable by nature (it is not really stable, but the probability of failure is low) or by construction. In this latter case, stable storage can be implemented by a combination of volatile storages that should not fail at the same time. One calls them **fail-independent** or **not decay-related**. Stable storage is realized in database systems by replicated ("mirrored" or "shadowed") disks, in semiconductor storage by duplicated memories. ECC-protected memory is not considered stable. ECC is merely a mean to reduce their failure rate.

Example: a stable storage can be implemented by writing the data into two (volatile) memory units, which are supposed not to fail both at the same time. This assumption is justified when both storages belong to two separate nodes or are powered separately. These memories are individually checked by an error detecting code (correction is not necessary). Data is read from one memory only, and checked. If the data read is correct, the computation continues. If the data is incorrect, the data is read from the other memory and checked. If the second data are correct, it is written into the other memory to correct it in prevision of transient errors. If the second data is incorrect, too, continued operation cannot be ensured.

This method has been applied to semiconductor storage and disks. Disks cannot be written into simultaneously. Therefore, a pair of disks is used. One disk holds the old valid state, while the other is being actualised to the new state. Then the other disk is actualised. This way, a crash during the write operation will leave one disk intact. If one wants to be sure that the write did took place, one can read the disk after having written the information into it and repeat the write till success. This method is known as **shadow** disks or **mirrored** disks in commercial computing. Note that writing to stable storage requires more than double the time to write to a normal disk (Figure 6-4):

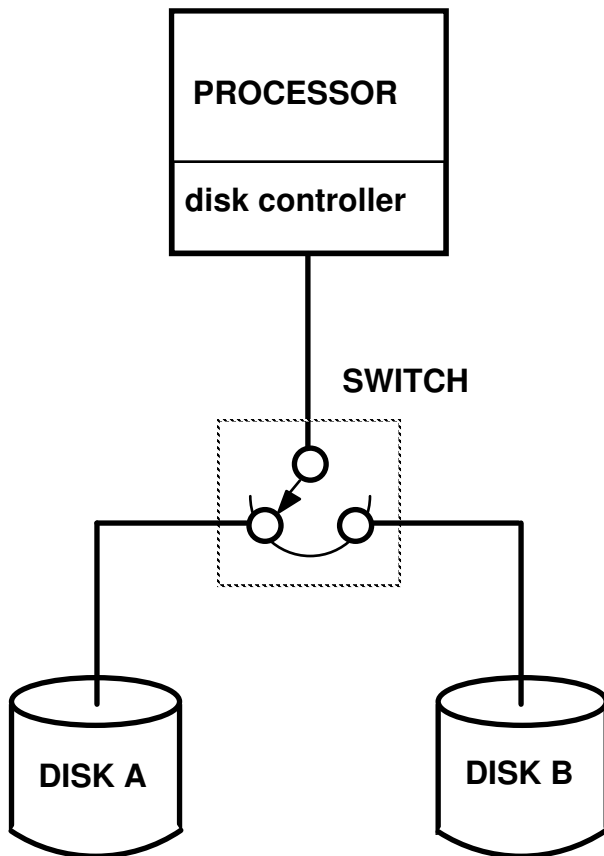


Fig. 6-4: Shadow Disks.

Although stable storage is not affected by the crash, it can be corrupted by erroneous information if the processor is not fail-stop. Another and more frequent reason is that the processor did not output false data, but did not complete a sequence of operations that it began. The volatile state is then inconsistent. Stable storage can be corrected by overwriting selectively the erroneous or inconsistent data with their previous values. To restore a stable storage, there must exist in an independent storage a **log** to **undo** all the changes that have been done to the stable storage since a defined starting point.

- **environment:** The environment can also be considered as a kind of stable storage. We set here an arbitrary limit and consider the following kinds of storages as being typical of the environment of a computer. We will include in the environment all parts of the computer and its external world that cannot be recovered just by overwriting the erroneous data with correct data.

The environment can be assignment correctable, reversible or irreversible.

An **assignment correctable** environment behaves like a stable storage: it can be corrected by writing a new value into it.

Examples:

- a Digital-to-Analog converter that outputs a set-point for a controller.
- a memory location also behaves that way.

A **correctable environment** is a storage that can be corrected by an action different from overwriting. Erroneous information can be compensated for, for instance by inverse actions or cancel messages.

In simple cases, there exists a undoing action for each do-action: closing a switch can be corrected by opening it again (if no damage has yet resulted). A text editor may have an **undo** key that undoes the effect of the last

operation by an inverse operation (the operation "delete word" is undone by the operation "undelete word" which restores the previous word).

All output instructions that cause incremental changes to the plant must be corrected by a positive undo-action.

Example:

an erroneous command that was sent to a stepper motor, "advance by 23 steps" requires a corrective action "step back 23 steps" (if it is not yet too late).

The undo operation can become quite complicated and involve humans.

Example:

when a wrong instruction is given to open a cement silo, there is no "un-open" instruction. The rollback involves humans shuffling the substance back to the silo and modifying the production.

An **irreversible environment** is a part of the stable storage that cannot be corrected once erroneously written into and which will make error recovery fail. The irreversible storage lies outside of the confinement region of the processor that writes into it. For instance, if the designer's assumption is that an error cannot leave a node, the storage on any other node belongs to the irreversible storage. Most of the outside physical process belongs to this category. We will therefore consider that any erroneous information that leaks to the plant leads to a total outage of the computer.

Example:

the erroneous closing of a high power switch to ground has to be counted to irreversible storage.

A grey zone exists between irreversible and reversible systems when time is concerned.

Example:

when a false data item may be sent to a mailbox, it can still be corrected (it is assignment correctable) as long as nobody has read the mailbox - i.e. as long as nobody did consume the data. After that, communication has taken place with the reading entity, which must then be involved in the recovery process. If the other entity is not reversible, then a failure occurs.

The following Figure 6-5 shows the hierarchy of storages according to their behaviour in case of failure:

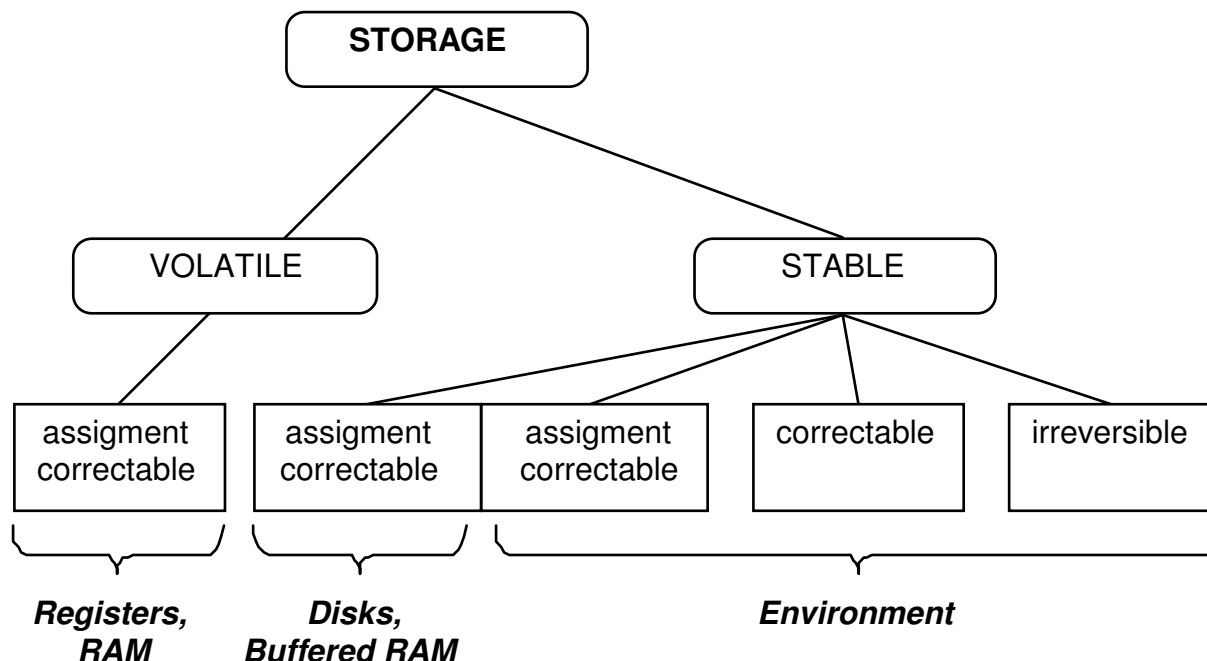


Fig. 6-5: Hierarchy of Recoverable Storages.

The above distinction into volatile storage, stable storage and environment is not a static assignment, but depends on which faults are anticipated. It may well be that the same storage belongs to two different categories depending on the kind of fault expected.

Example: A computing system consists of a processor, an ECC-code protected RAM memory and a database consisting of fail-stop disks.

For the first case, we assume that a fault in the processor has erased its registers, but that memory and disk have survived. The previous state of the processor can be reconstructed from RAM memory, considered as stable. Inconsistent modifications to the memory and disk state must be corrected. This is called a **hot restart**.

As a second case, we consider that the memory failed, but that disks survived. The memory and the processor can be reloaded from a copy maintained on disk. Modifications to the disk state since the taking of the save point must be corrected. This is called a **warm restart**.

As the worst case, the database on disk is considered as lost. Its state can be reconstructed from an archival copy (back-up) on tape, for instance. This is called a **cold restart**.

And if we cannot rely on any stable storage to reconstruct the database state, well, this means trouble!

6.3 State saving and restoring of the volatile state

6.3.1 Principles

We assume here the processor proceeds through execution by modifying a **work storage**, which contains the current state of the task. The working storage is volatile, it is not trusted and considered as lost in case of failure. Therefore, a full copy of the state, called a **back-up** state must exist in a safe place to resume operation. This repository is the **save storage**, also called **back-up storage**.

The back-up copy must preferably be held in a non-volatile storage, or at least in a storage of which one assumes that it will not fail at the same time as the work storage. The minimum requirement is that the save storage be fail-independent from the work storage.

Note: the save storage (copy of the volatile storage) is logically independent from the stable task state (part of the state which survives). Both can however be recorded on the same physical medium.

The full copy must contain a valid state of the task, i.e. all information necessary to resume computations from that state on, including the state of the processor, its coprocessors and MMU registers and of the I/O registers. The save storage has therefore at least the same size as the work storage, plus the processor registers. A 100% storage redundancy is therefore required. A simple arrangement is shown on Figure 6-6:

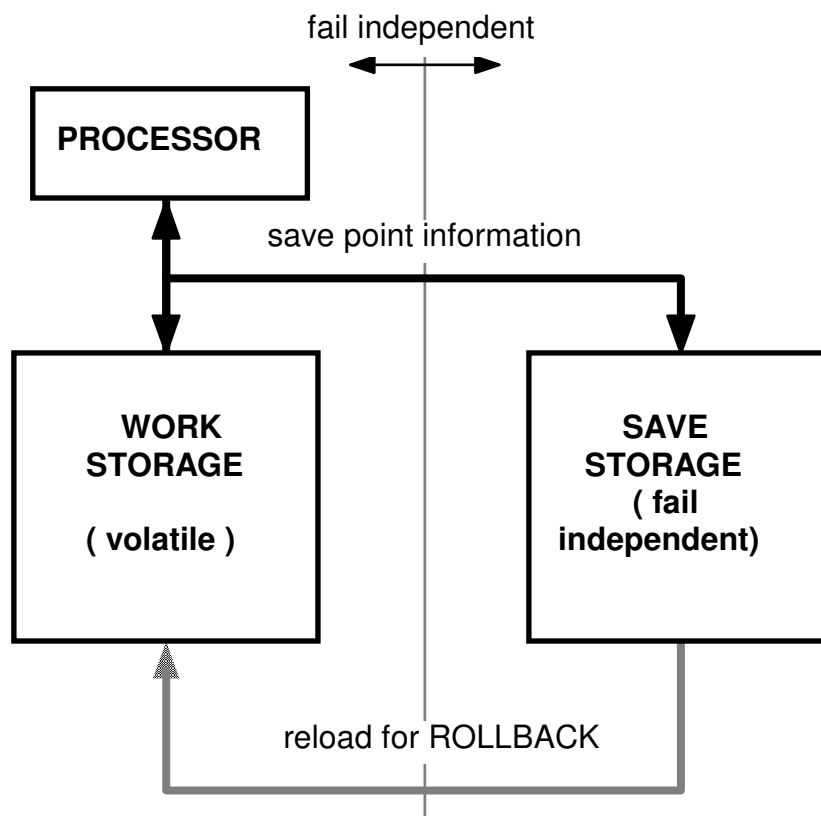


Fig. 6-6: Work and Save Storage.

The bar in the way between the storages symbolizes that they are fail-independent, i.e. they belong to two different confinement regions.

The ideal would be to keep the back-up copy continuously actualised. One could for instance perform every write in the work storage and in the back-up storage at the same time or one after the other. This is however not sufficient. The reason is that the back-up state consists not only of the variable's state in memory, but must also contains the state of the processor's registers. Therefore, one would need to save the processor's registers at every write operation. This would take most of the computing power of the processor for state saving.

Therefore, the back-up state is only actualised at regular intervals, which correspond to the save points of the program. At these places, the current state of memory and the processor's registers are copied to safe storage. But we will see that a continuous copy is nevertheless possible.

6.3.2 Full Back-up

The simplest technique consists in making a **full copy**, or **full back-up** of the volatile storage to the save storage at every save point (Figure 6-7):

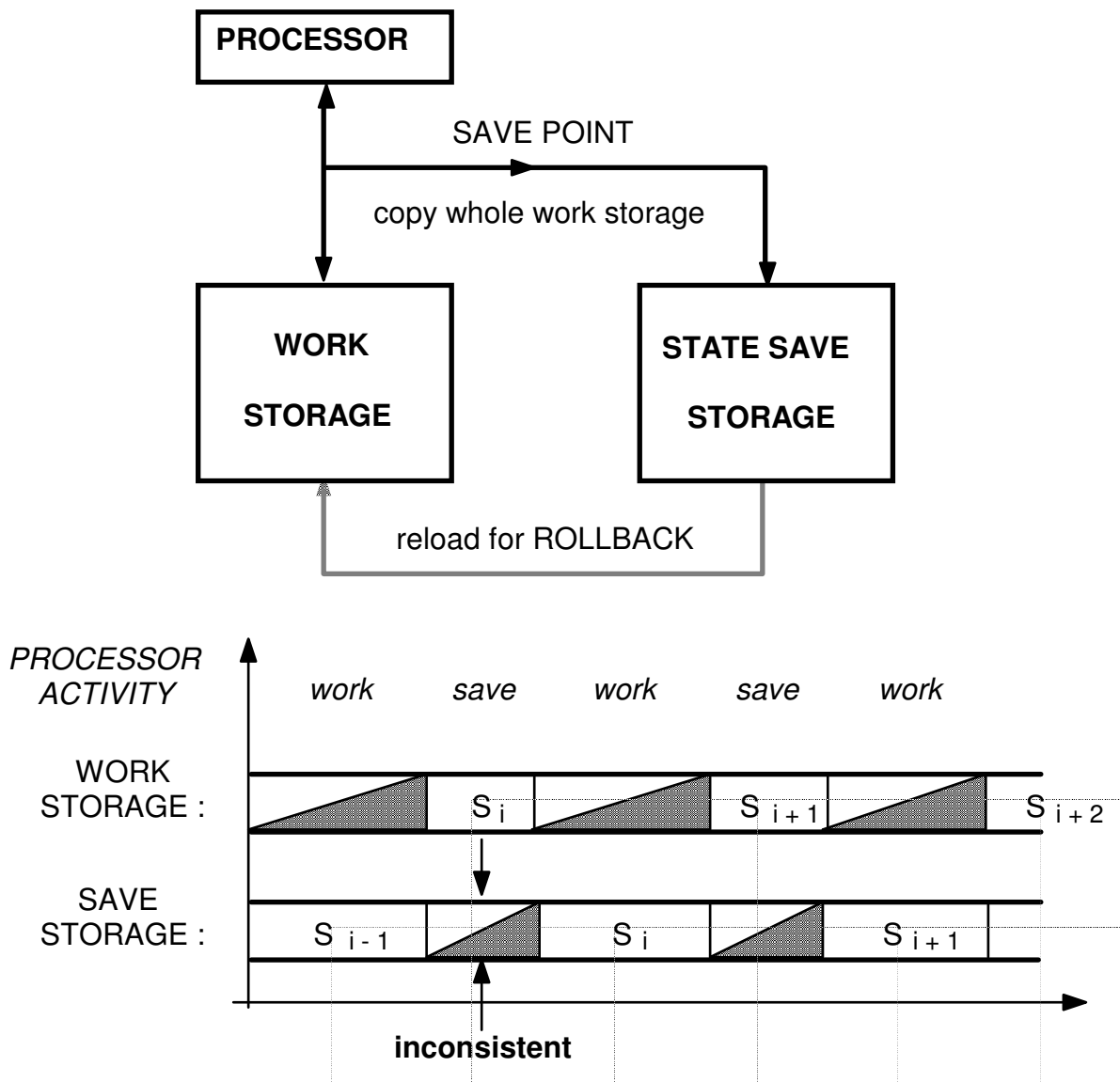


Fig. 6-7: Full Back-Up.

The timing diagram below the blocks represents the state of the work and the save storage in function of time. The shaded triangular area expresses the fact that the corresponding storage is changing in an unpredictable way.

The full-copy technique is applied in most commercial computing centres. To protect against a disk crash, the whole database is copied from the disks to tapes and stored in a safe place. A full back-up takes quite a long time, up to several hours, since all disks are supposed to belong to the volatile storage (for the kind of crash envisioned) and

the computer is halted during that time to prevent inconsistent copies. Therefore, such full-back-up only can take place once a week or once a month. When a major crash occurs, the operating system is restarted, the tape copy is loaded into the disks, and work is resumed.

6.3.3 Protection against Crash during Copy

The state in the back-up storage is inconsistent during copy. The solution is to use two back-up storages, which we will call now state save units (SSU). The SSUs are used alternatively, one serving as a depository for the former state while the other is being actualised (Figure 6-8):

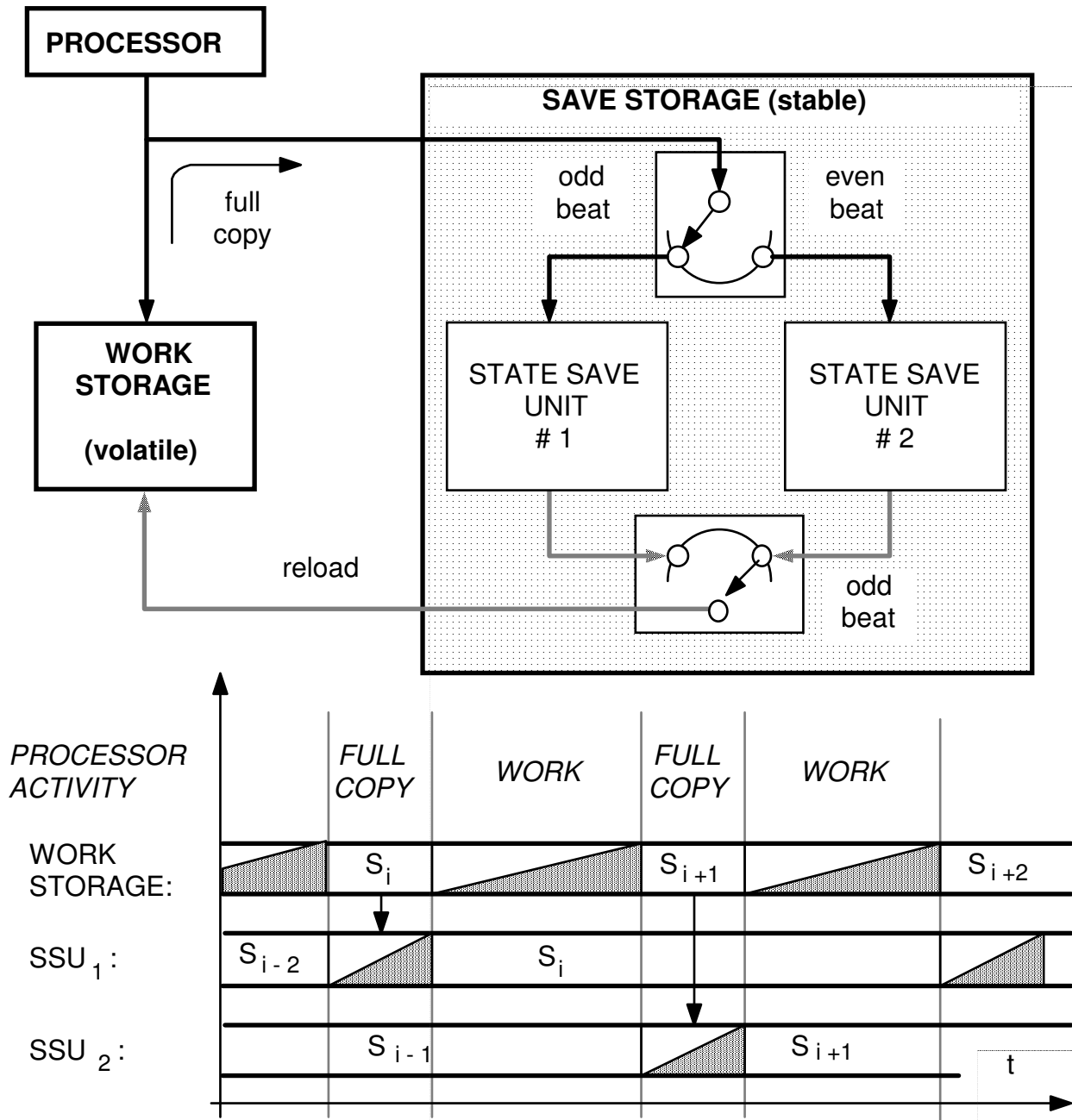


Fig. 6-8: Using Two Separate State Save Units.

Example:

This only reflects the experience that it is unwise to use the same tape to hold the back-ups of two successive save points. If the processor or the volatile storage crashes during the update, the tape will be in an inconsistent state and cannot be used any more. Rather, a minimum of two tapes must be used. In a computing centre, one could use one for the even days and one for the odd days. Anyway, a computing centre would not rely on just two tape copies for all back-ups, since the cost of tapes is negligible and one can afford to keep the back-ups of several weeks. This guards further against error latency problems, that is against errors that are only detected a long time after they occurred. For instance it may be necessary to reconstruct files that were erroneously erased some weeks ago, with several save points in

between. We do not however consider this problem here, since we consider firstly hardware errors, and secondly assume that every error is immediately detected (fail-stop behaviour).

The principle of the use of multiple copies for tapes cannot be extended directly to warm stand-by, or to other methods which use a semiconductor memory as a save storage, since such memories are costly. But we can deduce from it that the back-up storage of the stand-by must be at least twice as large as the working storage of the working unit, i.e. contain one valid copy and one copy in the process of copying.

When considering cold stand-by implemented in hardware, for instance by non-volatile RAMs, exactly the same considerations apply. One needs a save storage twice as large as the working storage. We will see means to reduce the back-up storage's size at the expense of a longer recovery time.

6.3.4 Partial Back-Up

A full back-up takes quite a long time. We have seen that a computing centre can afford such a back-up only about once a week. But even at the hardware level, the time for saving can be considerable.

If the state consists of 64KB of RAM, copying it with a processor in DMA mode which can do a read or a write operation every $1\ \mu\text{s}$ costs $2 \cdot 65536\ \mu\text{s} = 130\ \text{ms}$, without counting the time required to save the registers and set up the DMA transfer. If the distance between save points is 1 s, then about 13 % of the computing time is lost to state saving. If a maximum recovery time of 10 ms is required, this solution is clearly impracticable.

To bring the save points closer, **partial back-ups** (sometimes incorrectly named "incremental checkpoints") are used. For this, a full copy of the state must exist in the save storage. This copy has been taken at the last FULL SAVE POINT, which is a special save point at which the totality of the state has been saved.

The storage itself is divided into regions, which are generically called **domains**. A domain is a part of the storage that is modified as a whole, atomically. A domain can be a file in a database system, or a memory location in a process control computer. Every time a domain is modified, its new value is recorded in a **redo-log** (also called a **redo-FIFO** when it is implemented in hardware), along with the value of the processor registers and pointers.

In cold stand-by, all changes to the objects which have occurred since the last taking of the full copy (last full back-up) are recorded in the log, which is organized as a FIFO (Figure 6-9):

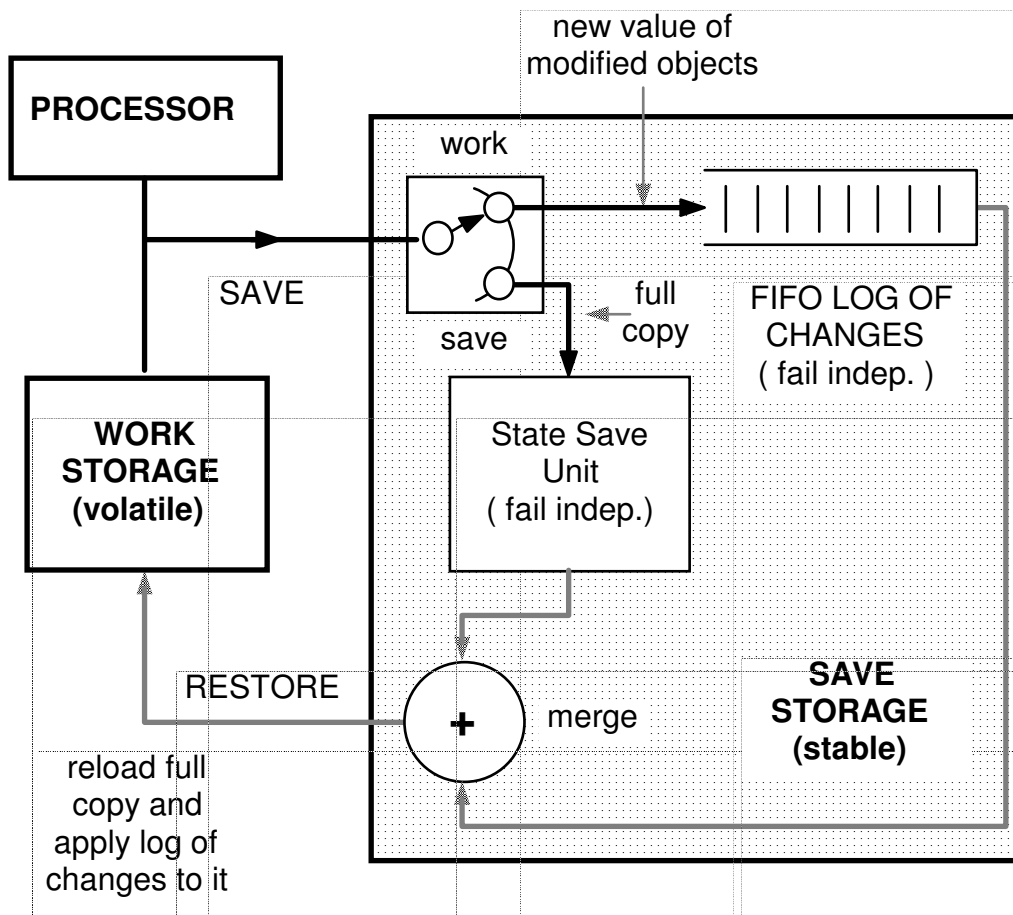


Fig. 6-9: Partial Back-Up and Redo-Fifo.

This method is common to back-up databases: a full save point is taken for instance every week (back-up) on tape. Each time an object is modified, a save point is taken and its new value is recorded on disk or on a streamer tape. That way, a **redo-log** of changes to all objects modified since the last full save point is kept in save storage. After a crash, the full back-up is loaded into the spare (or the repaired processor) and the log of changes since the last full back-up is applied to it, in the same order as the changes were made (hence the name "fifo"). The redo-fifo can be considered as a shortcut to computation: it prevents the repetition of computations which have taken place since the last full back-up was made by merely substituting the result in the computation. The playing of the log can be quite costly in time, but it should occur very seldom. The playing of the log is done during a phase called **redoing**. Redoing is not identical to roll-ahead. Rather, redoing is a shortcut to computation which does not use the interrupted program as a script, but redoes the assignments maintained in the log.

The following Figure 6-10 shows how recovery is done:

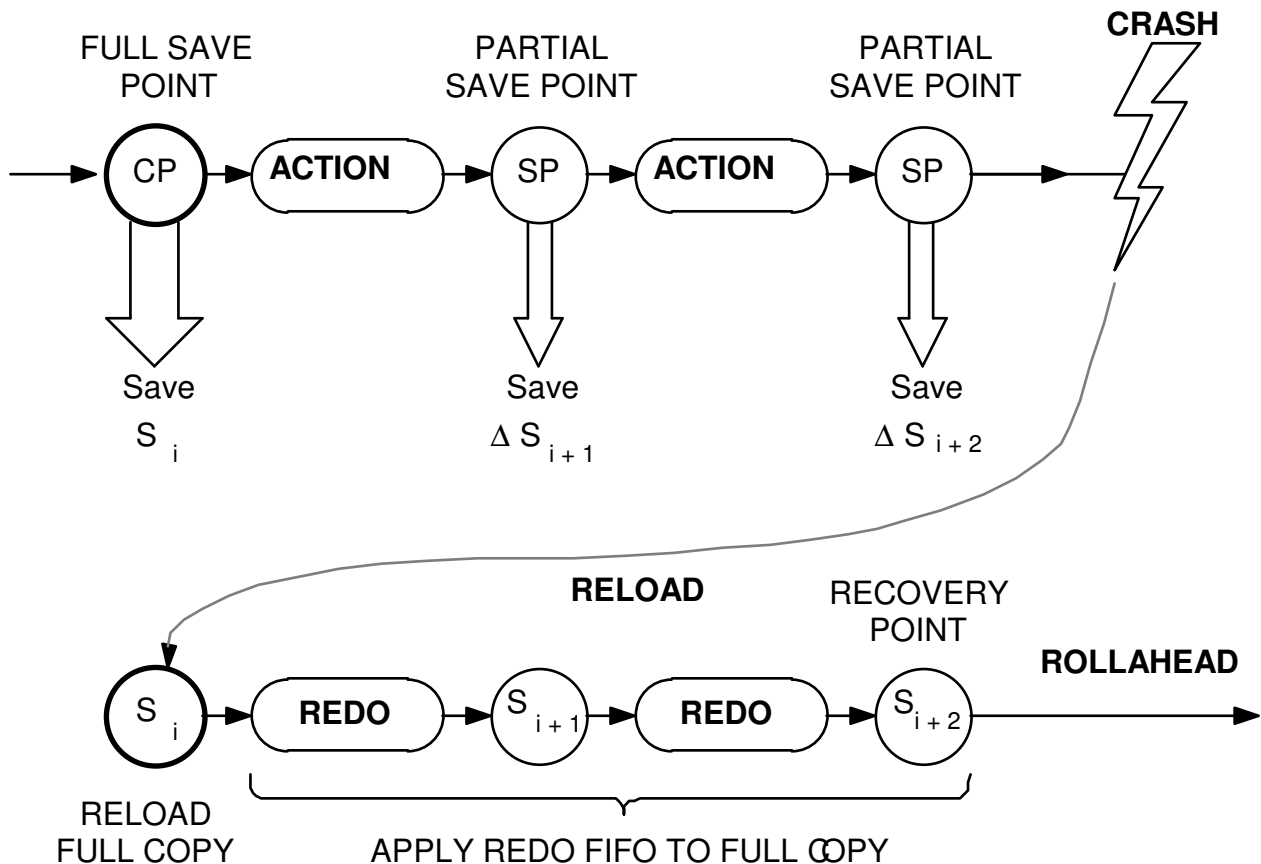


Fig. 6-10: Save Points, Recovery Point and Roll-ahead.

6.3.5 On-Line Back-Up Actualisation

To reduce the recovery time, the full copy can be actualised in parallel with the computations: each time an object is modified, it is automatically added to the redo-log, and this modification is added to the full back-up. The redo at crash time will be minimized since only the last modification will need to be added. The time for keeping the full copy actualised is taken from the normal computing time, but it is in any case less than the time required for a full copy. It is nevertheless wise to keep the full back-up and the redo-log separately, in case of problem in the actualisation program.

In computers with cold stand-by, a background task could continuously actualise the full copy of the state by applying the redo log to it. To this purpose, two copies of the state must be maintained, and actualised one after the other, to cope with a possible crash during copy.

In warm stand-by, the contents of the Redo-Fifo are directly used to actualise the full copy, since there are only one or two state save units. At every save point, the content of the fifo is copied into the state save units, first actualising

one unit, then the second to avoid the problem of a crash during the copy operation. Therefore, there is no necessity to redo the computations at recovery time.

An example of partial back-up is used in the TANDEM 16 computer: The state is saved on a task-by-task basis, i.e. each object is a task state. At each "checkpoint", the contents of stack and heap of the task are saved, along with the registers of the processor and some administration information. This information is used to actualise the state copy which resides in the stand-by node.

6.3.6 Continuous Copy

To reduce further the recovery time, one could perform each modification on the work storage and the back-up storage simultaneously. This actualisation costs practically no time if the processor's bus can be operated in broadcast mode, i.e. a write accesses the work storage and the back-up storage simultaneously (Figure 6-11):

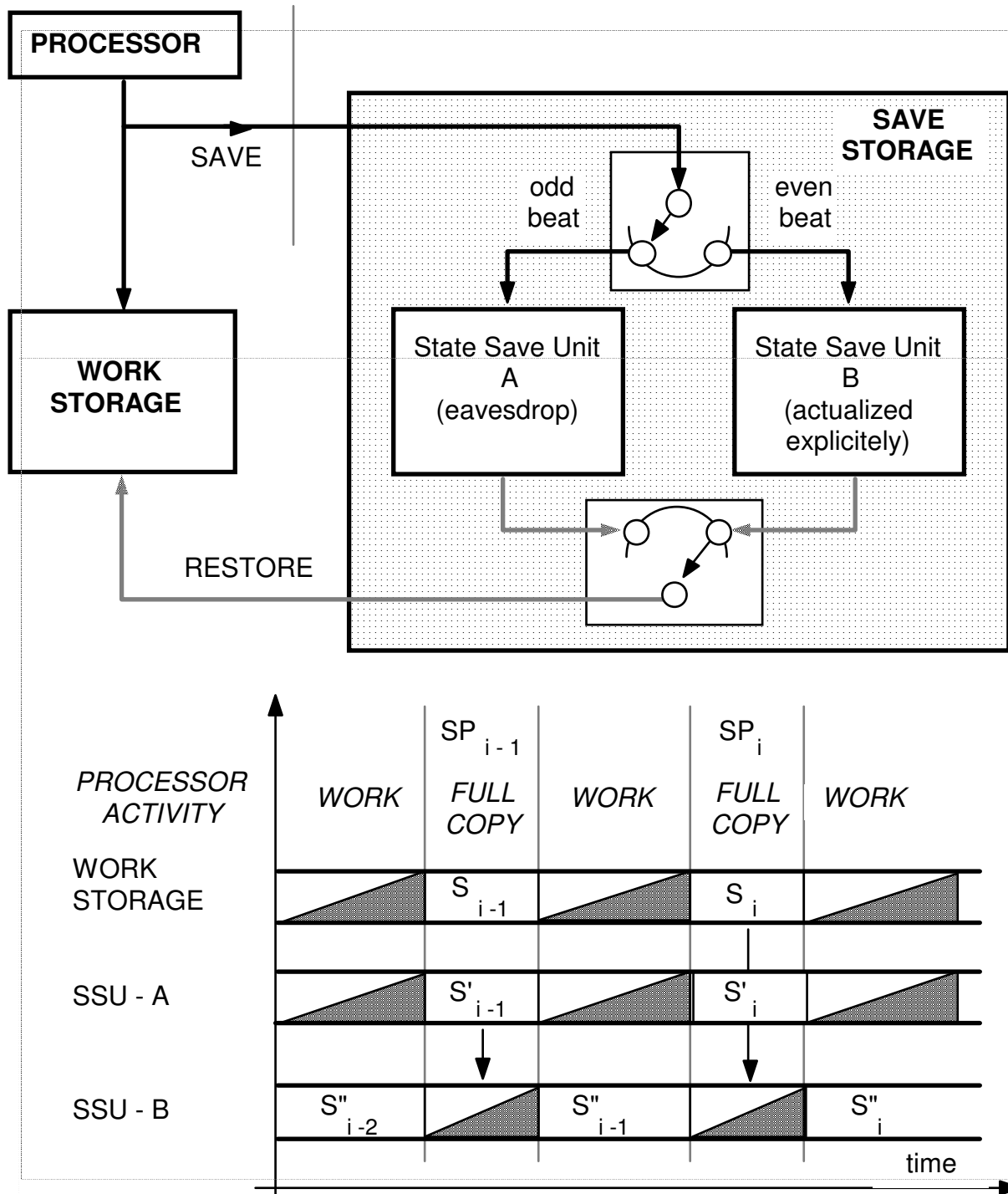


Fig. 6-11: Continuous Copy.

The function is quite simple: we can suppose that both state save units (SSUs) have the same state S_{i-1} at save point $i-1$ (see Figure 6-11) a time t_{i-1} . While SSU-B keeps the state S_{i-1} which prevailed at the save point $i-1$, SSU-A is actualised by the processor from state S_{i-1} to S_i . At the end of the recovery interval, when reaching save point i , the processor's registers are copied into state save unit A. The copy in SSU-A is now consistent and its state is S_i . Before continuing operation, the processor must first actualise SSU-B from state S_{i-1} to state S_i . After that, the processor continues execution, actualising SSU-A from S_i to S_{i+1} .

The actualisation of SSU-B could be done by copying the content of SSU-A into it before continuing. This solution would require a DMA controller to copy SSU-A to SSU-B, but little would be gained in time with respect to the solution shown in Figure 6-11.

One could gain more by copying only what has been changed from state S_{i-1} to S_i and not dumping the whole of SSU-A. So, one possibility is to use a redo-fifo, which records all changes performed between S_{i-1} and S_i , and which is dumped into the SSU after a save point has been reached and before the processor is allowed to keep on (Figure 6-12):

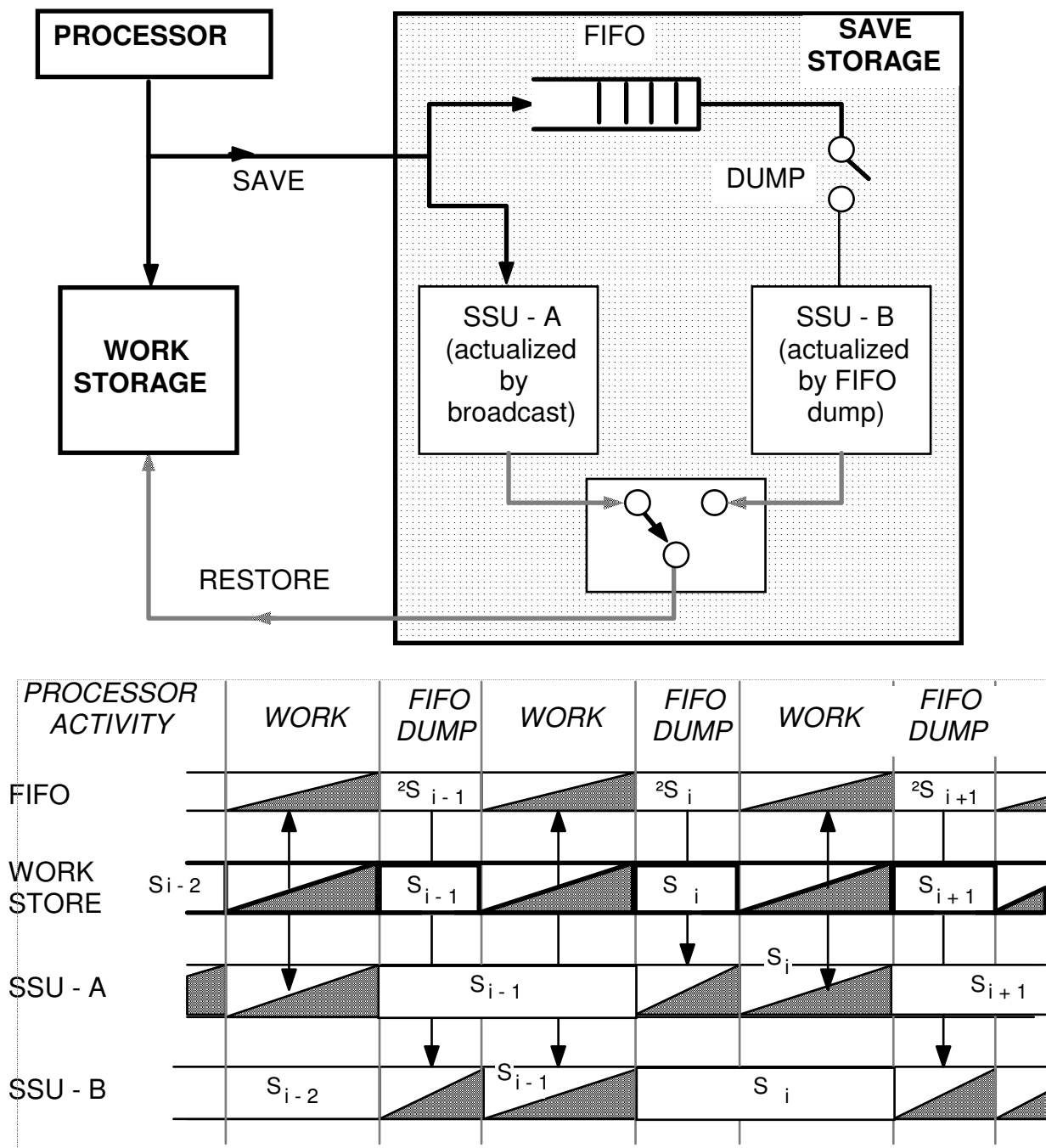


Fig. 6-12: Actualising one Copy by Broadcast, the other by Redo-Fifo.

6.3.7 Merge Back-Up

The Redo-Fifo solution is attractive, but a more elegant solution exists. The reason why the FIFO, or an additional complete copy is necessary is that the SSU which holds state S_{i-1} must be brought from state S_{i-1} to state S_i in order to serve as a back-up until save point S_{i+1} . Now, one could bring the SSU to state S_i by copying the other SSU into it, and from state S_i to S_{i+1} by following the processor's activity (Figure 6-13):

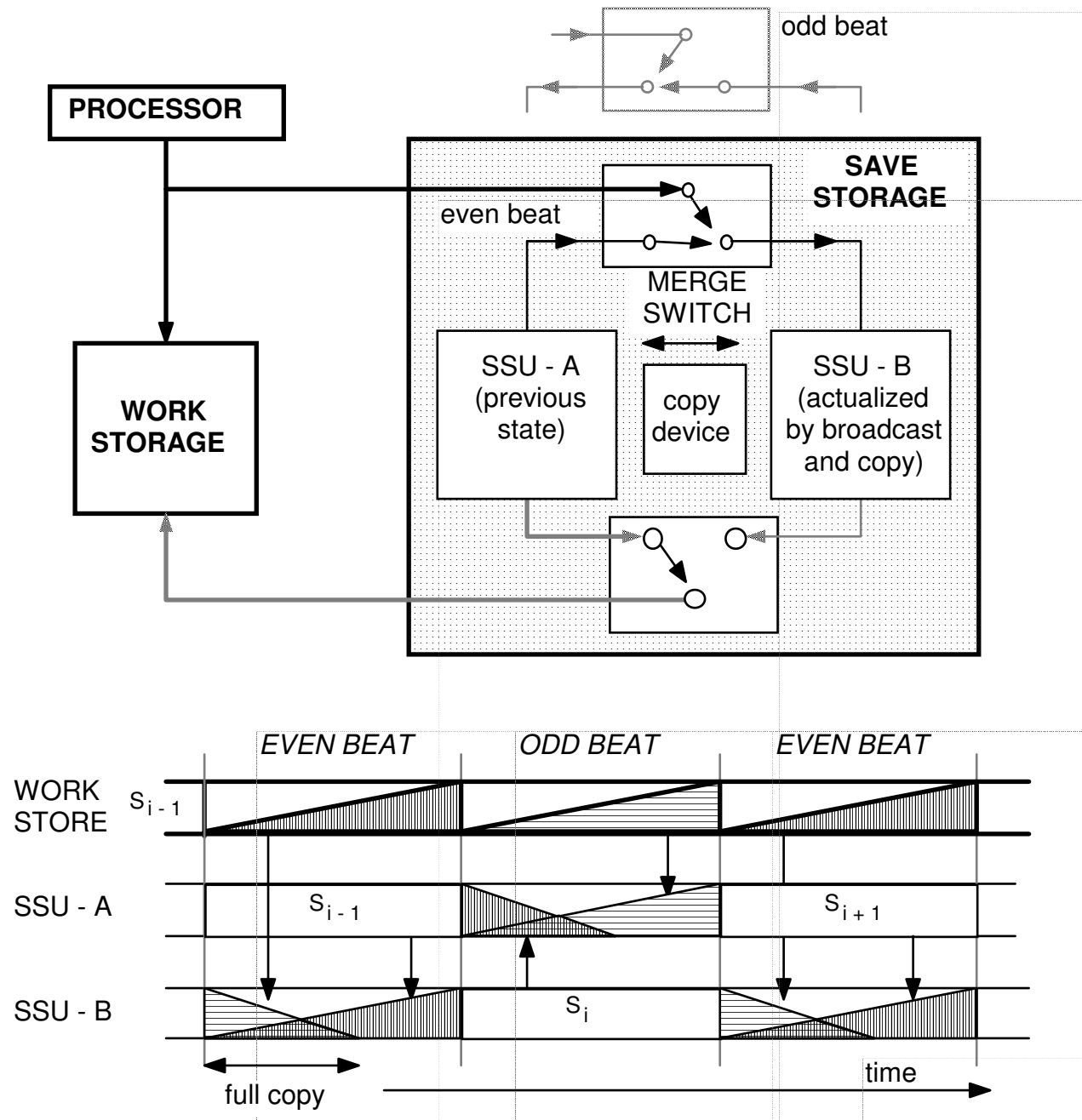


Fig. 6-13: Merge Back-Up.

The idea of merge back-up is to perform both operations simultaneously. As Figure 6-13 shows, as one SSU retains the state S_{i-1} , the other is simultaneously updated by copying the contents of the first SSU and by following the processor's activity through broadcast. This only works if the updates coming from both sources, the other SSU and the processor, are coordinated. The rules that must be followed are:

An object that has been modified by the processor may not be changed any more by an update coming from the other SSU.

The processor is not allowed to proceed after the next save point as long as the copy of the first SSU has not been completed.

To realize rule 1, each entry in the SSU needs one additional bit (a tag) that marks it as modified by the processor during that interval. At the beginning of the recovery interval, all "modified" tags must be reset.

The merge back-up costs also a non-negligible overhead time for the processor, which consists of the time required by rule 2 and the time required to reset the "modified" tags. Unfortunately, unless one uses custom memories with a reset pin, the resetting of all tabs takes about as long as dumping the memory, so this solution is not as interesting as it looked like.

6.3.8 Merge and Swap

A last improvement can be made by not actually copying the values, but only by moving pointers to them. There are still two copies of the state in the SSU, but the consistent set consists of memory locations belonging to a double-size SSU. That means that to one memory location of the working store correspond two memory locations in the (now unique) SSU. To each memory location, a pointer tells whether the location belongs to the (consistent) previous state or to the state that is in the process of building. The arrangement is similar to Figure 6–13. The principle is show in Figure 6-14.

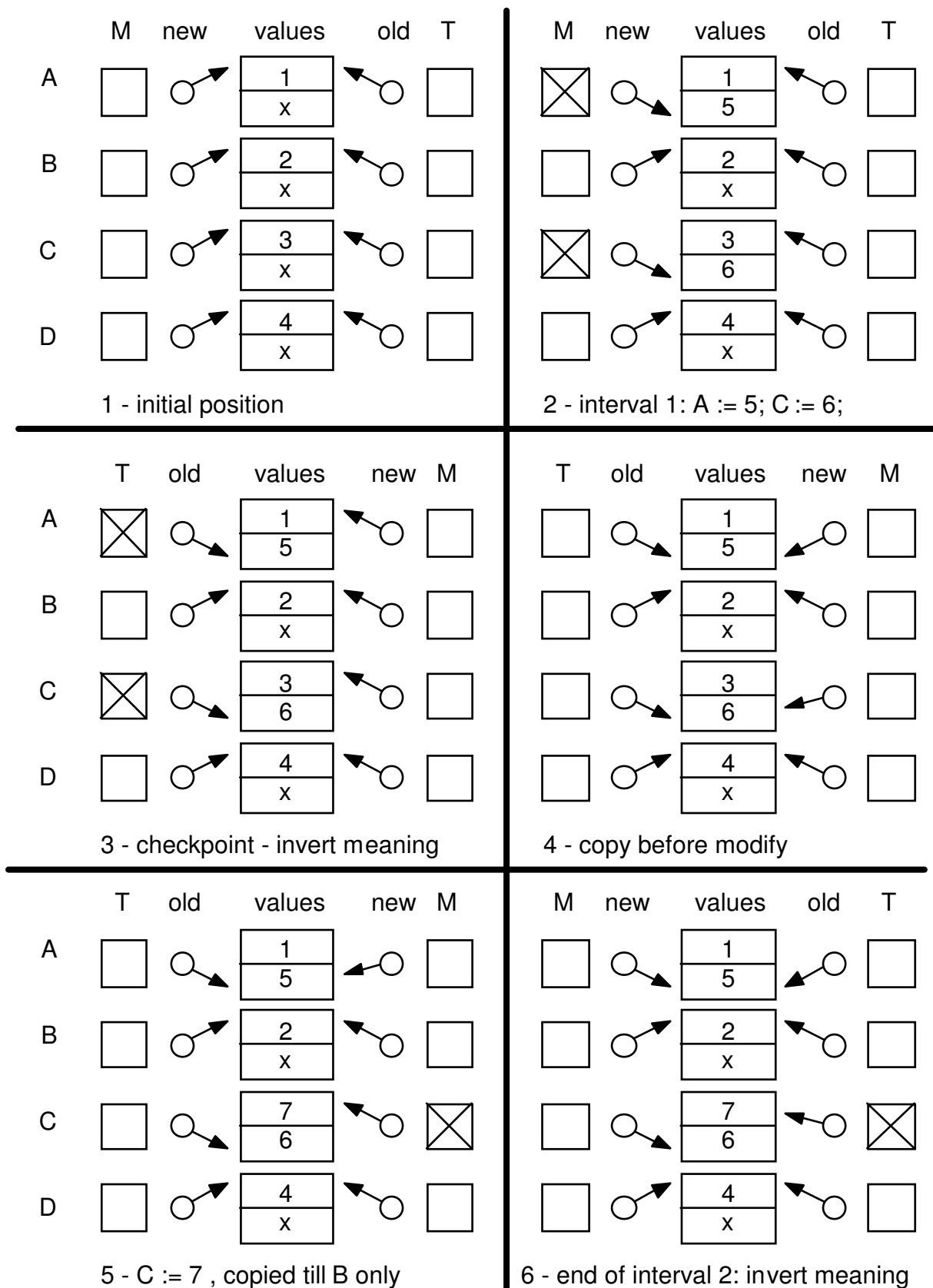


Fig. 6-14: Merge and Swap Pointers.

The principle of operation is that it is easier to modify a pointer to a variable than to copy values from one SSU to the other. In this scheme, each variable is stored in two memory locations, corresponding to the two states required: one that is valid and the other that is been actualised. For instance, variable A in Figure 3-14 is stored in two locations with values 1 and x. The two values are stored in the same physical storage: there is no need for two physically separate SSU anyhow; the only requirement is that the SSU be fail-independent from the work storage.

To each variable, four tag bits are associated:

- O- an "old" bit which points to the one of the two values that represent the last valid state. The vector of the "old" pointers therefore always represents the previous, valid state.
- N- a "new" bit that points to the one of the two values which is modifiable in the current cycle. The vector of the "new" pointers does not represent a consistent state until copy is finished and the next checkpoint is reached.
- M- a "modified" (M) bit which tells that the (new) value has been modified during the current cycle and consequently should be copied during the next cycle.
- T- a "transfer" (T) bit which means that this value has been modified during the previous cycle and should now be transferred.

The operation can be followed from Figure 3-14:

1. Initially, the SSU is loaded with the values of variables A, B, C and D. Admitting that these values have not been modified since several cycles, the "old" as well as the "new" pointers point to their current values (1,2,3,4). None is tagged as modified. Let's call the state pointed to by the "old" set as S_{i-1} state.
2. The variables A and C are modified. Since a value pointed to by an "old" pointer cannot be modified (it belongs to the consistent state S_{i-1}), the "new" pointer is flipped and the value is written. At the same time, the "modified" bit is set.
3. At the end of cycle i , a checkpoint is taken. The meaning of "old" and "new", respective "M" and "T" is reversed. At that point, both "new" and "old" sets point to a valid state. The "new" set points now to state S_{i-1} while the "old" set points to state S_i .
4. The previous valid set S_{i-1} is discarded and the values actualised to form state S_{i+1} . To this effect, memory is scrubbed from one end to the other by a piece of logic within the stable storage. The variables marked "T" (transfer) have been modified during the cycle i and their "new" pointer must therefore be flipped to point to the valid value. There is however an exception, as now follows:
5. The scrubbing of memory takes place in parallel with write operations to the variables. If a variable has been modified during the current cycle $i+1$, such as here $C := 7$, it must not be copied. In this case, the "T" bit is reset and the variable is tagged as "modified". When scrubbing comes to this position, it will overlook it since the variable is already actualised.
6. When the end of cycle $i+1$ is reached, the "new" set points to state S_{i+1} and the old set to S_i . The meaning of the pointers is again reversed and the game keeps on.

The whole process works only as long as the scrubbing of memory takes less time than the recovery interval between two consecutive checkpoints. This is normally the case. This method allows a very efficient state saving, since only four bits of information must be modified during scrubbing. While the information itself can be stored into a memory which is just as fast as the work storage, the tag bits can be stored in a high-speed memory to fasten scrubbing.

A full description can be found in [Krings 85].

6.3.9 FIFOs Back-Up

All the above solutions require the back-up storage to consist of two independent state save units. This is not due to the fact that a state save unit could fail, since the back-up storage is considered to be fail-independent from the work storage. A failure from the back-up would not cause a failure of the processor. The reason for doubling the SSU is that the processor may fail during the copy operation and leave there an inconsistent copy, from which recovery is not possible.

Now, we could come back to the idea of the FIFO depicted in Figure 6-12. If we are ready to pay some additional time at rollback time, a back-up store which uses only one SSU and two fifos may be used, as shown in Figure 6-15:

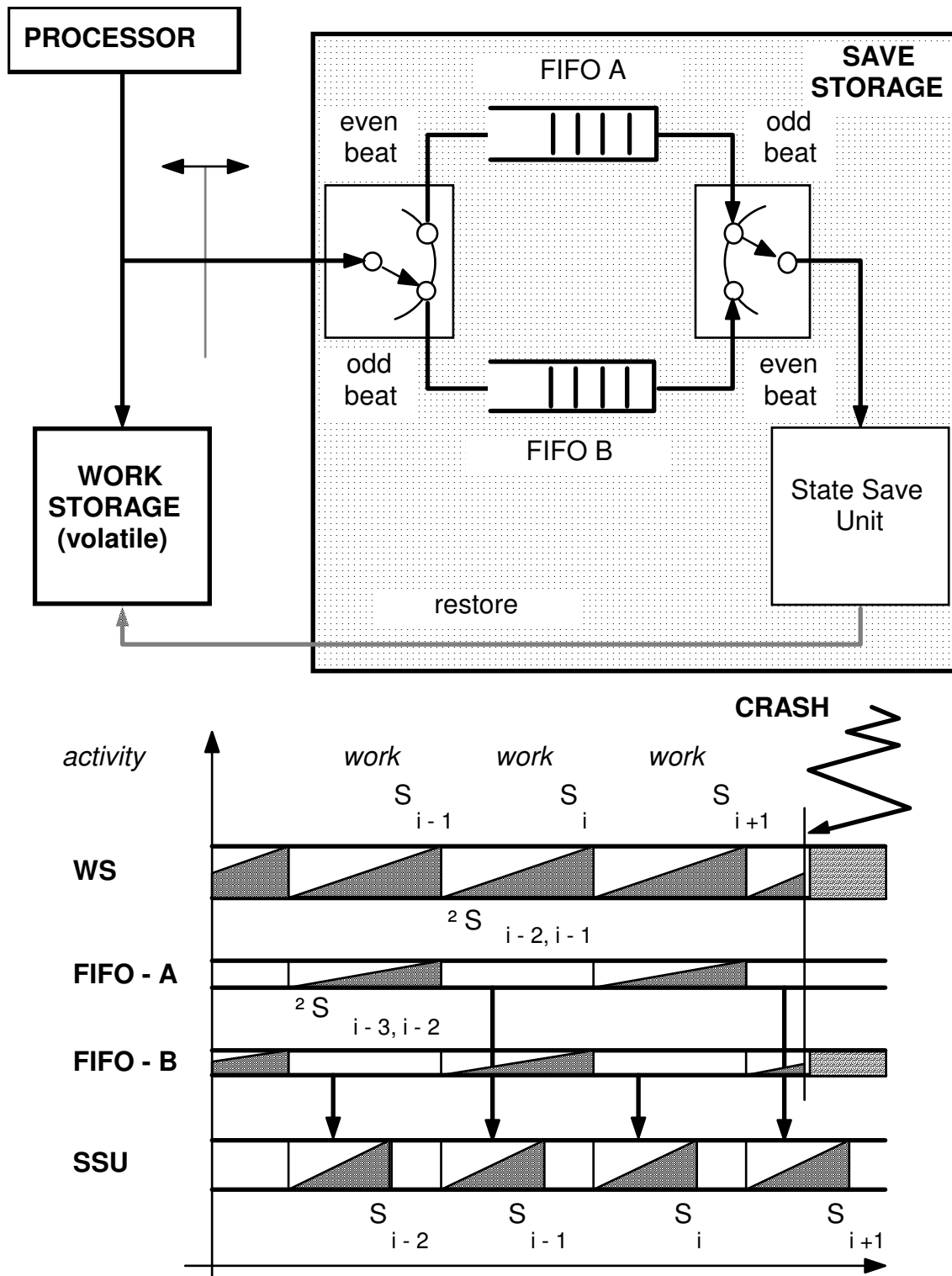


Figure 6-15: State Saving with one SSU and two Fifos

Suppose that the work storage is at state S_i at the beginning of a recovery interval. While the processor is transforming state S_i into state S_{i+1} , all changes from S_i to S_{i+1} are recorded in FIFO-A. In the next interval, the fifos are exchanged, and FIFO-B will record the updates from S_{i+1} to S_{i+2} . Thus, at RP i , FIFO-B contains the updates performed between $R_{P_{i-1}}$ and R_{P_i} and will remain unchanged until $R_{P_{i+1}}$; FIFO-A is void, and will be used to record the values from R_{P_i} to $R_{P_{i+1}}$.

Now, while the processor is proceeding from state S_i to S_{i+1} , the state save unit will be brought from state S_{i-1} to S_i by writing the contents of FIFO-B into it. It may be objected that if a crash occurs now, the state save unit will not be in a consistent state. This is true, but we assumed that the back-up storage and the work storage are fail-

independent. Therefore, if the processor or the work storage crash, the operation of copying the fifo into the state save unit can continue, and bring the SSU to state Si.

6.4 Restoring the stable storage

6.4.1 Dealing with Stable Storage

Stable storage contains that portion of the state which is assumed not to fail, but which may contain incorrect entries. One may wonder why the entries may be incorrect even if the storage is not affected by the crash. There are at least three reasons for this:

the processor was faulty at the time the entries were made (wrong address or wrong data) - this is the case when a full error coverage cannot be provided and the processor is not fail-stop. The situation is further typical of software errors, for which complete coverage is difficult to achieve.

the processor is not faulty, but a part of its state is volatile (internal registers). It is therefore necessary to undo the volatile storage back to the last point where the volatile state of the processor was saved, i.e. to the last save point. This is not yet a sufficient reason to rollback the stable state, since a second execution would generate the same results as the first. However, rollback is necessary if the processor relies on the former value of a variable to generate the new one (e.g. increment an index). A careful positioning of the save points can deal with this problem, at the expense of a frequent and unpredictable taking of save points.

the state in stable storage corresponds to a calculation which must be cancelled, for instance to break a deadlock or cancel a transaction. In this case, perfect rollback must be achieved so all effects of the former execution are erased.

6.4.2 Instruction Retry Mechanism

The second point of this list shows that an undo operation is not necessary if one follows some rules in the insertion of save points. A typical application of this idea can be found in retry mechanisms for processors. The retry mechanism is a device that allows the reloading of the processor registers after a crash. For this, a copy of the registers as they were before the execution is maintained in a hardware save storage. If an error occurs, the processor is reloaded and the computations continued. Since most errors are transient, this method significantly increases the dependability of the processor. Such a mechanism is implemented in the IBM 370 and UNIVAC 1000 computers.

As an example, we will now explain how recovery is done in the COPRA computer [Meraud 76]. The COPRA has an instruction retry mechanism that relies on two save register sets working alternatively. One register set holds the last save point while the second is being actualised. The task state is in stable storage (Figure 6-16):

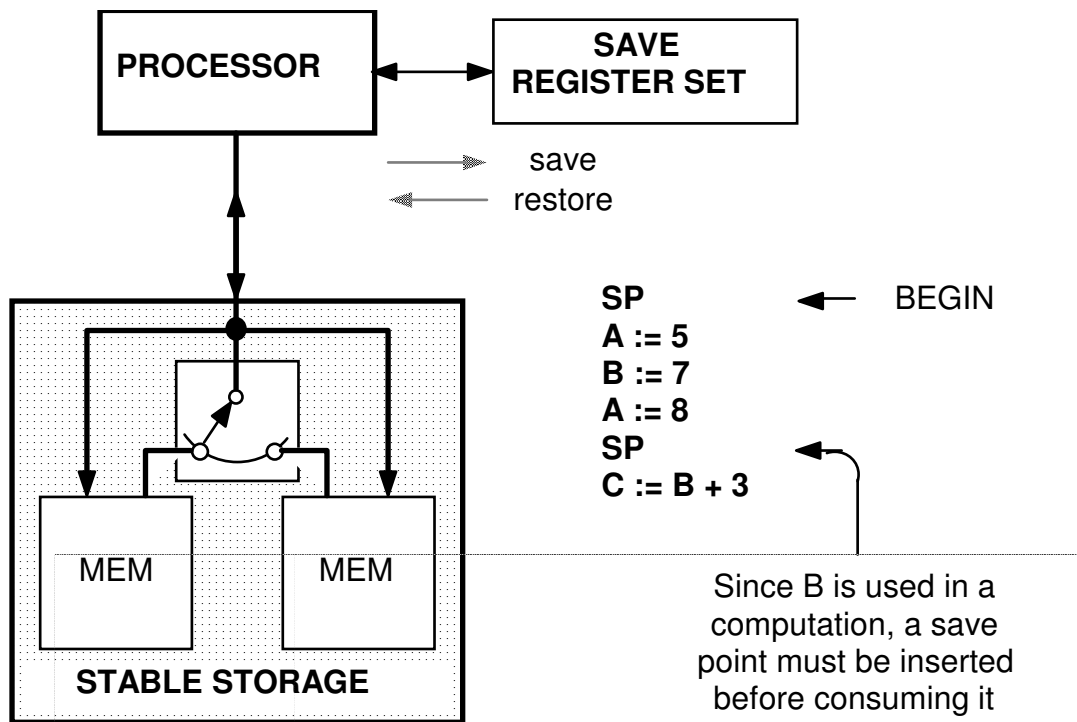


Fig. 6-16: Retry Mechanism of COPRA

The stability of storage is achieved by writing simultaneously into two independent memories (each protected by parity). Read is done only from one. If a read error occurs, the processor retries the read from the other memory unit, and if the read is successful, it is copied to the failed memory before continuing. This mechanism deals effectively with volatile faults and justifies the assumption of a stable storage.

At each save, the volatile part (essentially the processor's registers) is saved in the retry hardware, in a way similar to an interrupt handling.

The peculiarity of the COPRA mechanism is that undoing is made superfluous by inserting saves in such a way that the program is divided in repeatable sequences. The assumptions are:

- the processor never writes incorrect data to stable storage, i.e. the processor is fail-stop. This is realized by a self-checking pair of processors.
- the processor's execution does not depend on data which have been generated since the last save. This is enforced by the correct insertion of save points.
- the second execution of the processor will overwrite every variable which the first execution did modify (same execution track). This assumes that the machine is deterministic.

Under these assumptions, it is sufficient to put the processor in the state it had at the last save point and let it execute the program again from that point on. The objects modified since the last save point will be overwritten by the same values at the second execution.

This method implies that the saves are exactly chosen, following the following set of rules, according to:

Rule 1: a save point has to be implanted between the use and the production of an internal variable.

Rule 2: the production of external variables is made in two steps: first the external variables are produced, prepared in a register and then validated at the same time as the save point is finished.

Rule 3: a save point is created between the time an external variable is read and the time it is consumed by the program.

Rule 1 ensures that the processor's execution path does not depend on a variable that it produces. Rule 3 ensures that the execution path does not depend on external variables which may change.

Let's consider an example:

```

SavePoint;          (* section 1 *)
A      :=           6;
C      :=           A      +           3;
B := A + 1; (* B is produced, A is consumed *)
C := B + A; (* A and B are consumed *)
SavePoint;          (* section 2 *)
B := C;              (* a save point has been inserted since B *)
                      (* is produced after having been consumed *)
A := B + C (* no need to insert an additional RP since *)
           (* exists already *)

```

Note how assumption 3 plays: When section 1 is repeated, the value of C must be overwritten by the second execution (C:= A + 3). It is that way returned to its previous value by an implicit undoing.

Indivisible increments are not allowed.

For instance, A := A + 1; must be coded by:

```

Intermediate      :=      A;
SavePoint
A:= Intermediate + 1;

```

Rule 3 ensures that the execution is always repeated with the same inputs, even if the input changes in between:

```

SavePoint;
Read(Port2,              Position);
SavePoint;
A := Position * 360;

```

If the execution crashes before the 2nd SavePoint, it will read a new value of the input, but not consume it again. That ensures that once the second save point is reached, there will be no repetition of the input.

Let us look at the third rule: it ensures that an output is repeated in case of crash:

```

SavePoint;
Output(Port12, SetPoint);

```

We could modify Rule 3 in the sense that the output is not repeated in case of crash:

```

Output      (      Port12,      SetPoint      );
SavePoint;

```

Remains the slight probability that the crash takes place exactly between the output and the save point - hence the requirement that the output be made atomically with the save point: the same pulse which causes the save point to be taken (loading of the register's state) also loads the output register. The uncertainty window that remains does not exceed 20 ns. Although this does not give a 100% probability of success, one can evaluate the probability of failure and in most cases neglect it when considering other sources of unreliability.

The insertion of save points in the COPRA computer is done automatically by the compiler based on the above rules. On the average, there is a save point every 10 to 20 machine instructions. The overhead due to the save points does not exceed 8 %.

6.4.3 Save-Before-Modify

The above method of state restoration in the stable storage deals effectively with hardware errors. It is however not capable of dealing with software errors, since it is not possible to provide a 100% coverage against software errors, and some false data could reach the stable storage.

A restoration of the previous state in presence of false data is possible if all interactions between the processor and the stable storage are monitored, and the value of each object that is to be modified is first saved in a stable storage before it is updated. An object can be modified several times between two save points, but only the value saved at the first modification is important. The storage medium is therefore conceptually an **undo stack**, in which pairs of address and data are recorded (Figure 6-17) before the objects are updated. All writes accesses to the stable storage are therefore intercepted by the rollback stack, which performs a READ operation on the object before it writes into it.

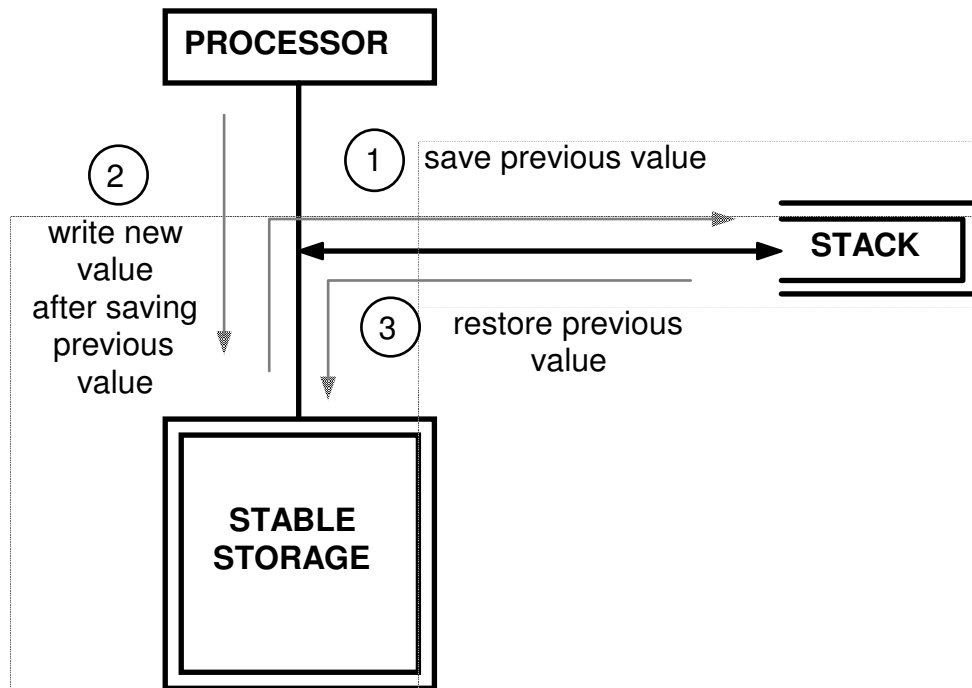


Fig. 6-17: Undo Stack for Stable Storage.

Restoring consists of applying the stack's entries to the stable storage in the reverse order in which the changes have been made, hence the concept of the save stack. An example is given in Figure 6-18:

former state:	computations:	crash:	restored recovery:	state:
A: 3	A:= 5; B:= 3; A:= 7;		A:= 5; B:= 4 A:= 3	A: 3
B: 4		B:		4
C: 5		C:		5
A:3; B: 4; A: 5				
(saved in stack)				

Fig. 6-18: Example of Save-Before-Value.

This method is used by several existing systems. The stack can be either a physical element [Kubriak 82] or just a conceptual stack maintained by the program itself. For instance, IBM's database system IMS uses a log of changes to the objects known as **undo-log**, which is recorded on a disk (a tape is difficult to read backwards). The basic rule is that the value of the object must be recorded **before** it is modified. A crash between modifying and recording would otherwise leave the state unrecoverable.

We must distinguish the entries to the stack from the taking of the save point. At the save point, a consistent view of all volatile parts of the system has been recorded. At every save point, only the previous value of a variable that is to be modified is recorded. At rollback time, the **undo-stack** is applied to the current state until the last save point is reached, as Figure 6-19 shows:

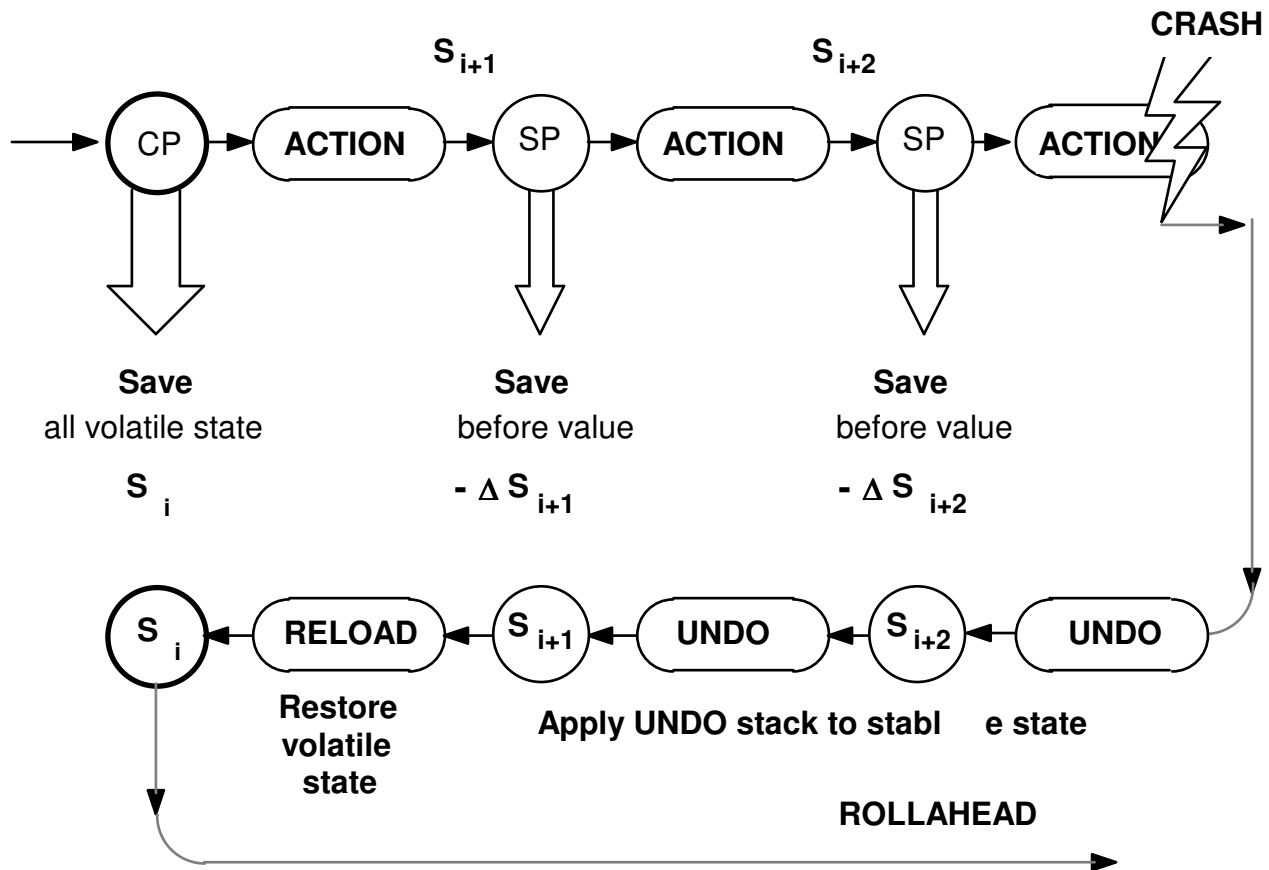


Fig. 6-19: Rollback, Save Points and Recovery Points.

We see now why it was necessary to make a distinction between save point and recovery point:

A **SAVE POINT** is a point in execution at which a part or whole of the state is saved. This may be in form of a before-image, an after-image, a complete dump or a set of undoing operations.

A **RECOVERY POINT** is a point in the execution at which one can conceptually return and resume the execution after an error occurred. Not every save point can become a recovery point. Further, a recovery point is not necessarily tied to a saving of a task state; it can consist of the recording of a few markers.

We will also speak of "restoring a recovery point" when restoring a previous state at recovery time.

6.4.4 Technical Improvements: Recovery Cache

A variable may be modified several times, but only the value prior to the first modification is relevant. By unrolling the stack, as shown in Figure 6-18, unnecessary assignments to variables are made (consider variable A for instance).

The recovery time could be shortened if only the first assignment to an object within a recovery block would give rise to an entry. Therefore, a mechanism could determine whether the object has already been saved or not. Such a mechanism is an associative memory, which has become popular under the name of "recovery cache". Although one can argue whether the term of recovery cache is well chosen, we will retain it.

A recovery cache is an associative memory connected between a processor and a (stable) memory, which intercepts the writes and performs a save-before-value in case the object has not yet been modified during the recovery interval. Such a recovery cache has been implemented at the University of Newcastle-Upon-Tyne [Lee 80], Figure 6-20:

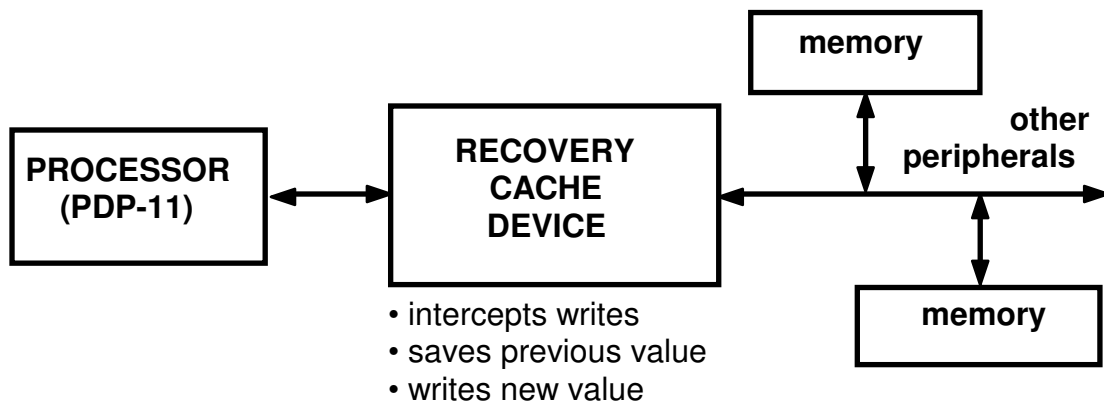


Fig. 6-20: Recovery Cache.

The recovery cache has however some problems of its own: first, it requires a "cache invalidate" signal that clears the cache at every recovery point. Since clearing a memory takes as long as writing to every location, the taking of a recovery point takes quite a long time.

Second, the cache must be able to give room to all entries. No object can ever be discarded. Therefore, the cache must have exactly one entry for every object that can be modified and so will be quite large. Its operation is therefore fundamentally different from the operation of a cache, which only holds a subset of the objects that could be modified. To limit the cache size, a mechanism decides that the cache is full and triggers a recovery point.

Third, the recovery task requires that all entries in the cache be passed to stable storage. Since one does not know when the cache is void (are they still modified entries in the cache), the cache dump takes a long time.

Since the recovery cache has been designed to deal with software errors and exception handling, the occurrence of rollback can be quite frequent. It is therefore still doubtful that the recovery cache has any advantage over a simple stack.

6.4.5 Comparison between save-before-value and save-after-value

It is interesting to compare the two mainstreams in state saving, save-before-value for stable storage and save-after-value for volatile storage. Their properties are summarized in the following synoptic table:

	save-after-value	save-before-value
Used for:	stand-by	retry
Trusted state:	stable back-up storage + redo-fifo contents (if there is no redo-fifo, then the back-up storage is doubled)	stable work storage + undo-stack contents (if there is no undo-stack, then the implantation of recovery points must follow precise rules)
Lost state:	task's state	only processor's state
State saving by:	saving a complete copy of the state and recording in the REDO-FIFO the new value of every object after it has been modified.	recording in the UNDO stack the previous value of every object before it is modified and recording the volatile part at the recovery point.
Recovery state reached by:	loading the spare with the archival state, and redoing the operations by applying contents of FIFO if necessary.	undoing the stable storage by restoring the changed objects to their former value, stored in the stack in addition, restoring of the volatile part by independent means.
Rollback time:	warm stand-by: short (switchover and actualisation): cold stand-by: hot restart: short (instruction retry) warm restart: medium (memory reload from disk) cold restart (loss of disk storage)	short - depending on the distance between recovery points.
Technical	replace the FIFO by a cache that holds the last modification only. (BBC's trace cache)	replace the STACK by a cache that holds the earliest value only (Randell's recovery cache)
Implementation:	IBM's IMS REDO-LOG Feridun's state-save units	IBM's IMS UNDO-LOG Randell's recovery blocks
Multiple rollbacks	yes, if the FIFO is kept and the archival storage points to the earliest recovery point (otherwise one state copy is needed for every recovery point)	yes, if the former values are kept in the stack for several recovery points along with the value of the volatile part at that places
Suited mainly for:	hardware errors	software errors
Hardware cost:	+ 100% storage capacity of backed volatile storage + 100% to back stable storage during copies. If a FIFO is used, the FIFO must be doubled to allow copies.	+ 100% needed to implement a stable storage with a pair of volatile storages The STACK must be in stable storage, too

6.5 Recovery and the environment

Until now, we considered recovery within a closed computing system, as an interaction between processor, volatile memory, database and logs. Within that world, the conceptual view of perfect rollback was achievable, since a computer is basically an assignment reversible machine.

When the environment is considered, the complexity of recovery increases. Rollback and Roll-ahead become quite complicated issues.

Rollback of the environment is complicated or impossible.

Roll-ahead must prevent omission and duplication of interactions and requires in most cases a perfect tracking of interactions.

6.5.1 Avoiding Rollback and Roll-ahead

Since rollback and roll-ahead are problematic, the best would be to avoid them altogether by taking recovery points atomically with every output or input action.

So, the data received should be immediately consumed and a recovery point taken in the same operation. If the probability of a crash between the reception of the data and the taking of the recovery point is acceptably small, the operation can be considered atomic.

The classical solution is to divide the output operation in two phases, similar to the order sequence in the military for firing: ARM - FIRE. The "Arm" order has no effect and can be repeated at will, and becomes an effect only when followed by the "Fire" command. The "Fire" command can be then repeated at will with no effect.

The same method is applied in safety equipments: the values are output to the plant, but they are just buffered. This operation can be repeated any number of times if necessary. Then, a VALIDATION ORDER is sent, which makes the orders effective. The validation pulse will have no effect if sent a second time, since it is ignored until a new value is sent to the device.

Therefore, the validation pulse must take place "instantaneously" with the taking of a rollback point. It will not be repeated by roll-ahead, since there will be no rollback for it.

This kind of "instantaneous" actions is intuitively suspect, since there is a small probability that the computer fails during the validation pulse, that it reaches some devices and not some others, etc... However, one must think how small this probability is: if the pulse lasts some 100 ns to be effective, and the rollback interval is 10 ms, a gain in reliability of 10000 has been achieved. The COPRA designers count that the period of uncertainty is not more than 20 ns. This may be sufficient to fulfil the specifications, especially when the plant itself is less reliable.

Atomicity can be assumed when the mechanisms for state saving are implemented in hardware. But as soon as one must deal with saving operations on disk or in memory, the atomicity of the operation is not admissible any more.

6.5.2 Rollback of the Environment

The environment cannot be rolled back. If a ground station missed the radio signals of a landing space probe, they will never come again. When time plays a role, when a computer is embedded in a physical environment, such as an industrial plant, the assumption of reversibility is at best an ideal, which holds as long as the computing system did not do any unrecoverable actions while it was faulty, such as opening the valves of a dam by itself. There are some limited ways by which one can rollback an external process:

- If the environment is assignment reversible, one can treat it as a part of stable storage, and correct the value by overwriting it with the previous value.

Assignment reversion of peripherals only works in rare cases, such as signal lamp pattern, setting of set-point in a control loop or positioning of a disk head over a specific track.

When the external world consists of a computing system, then the correction can consist in letting this computing system rollback itself. This can lead to a propagation of rollbacks, as we shall see. We will detail these aspects in the next chapter.

- If the environment is correction reversible, one can write in the **undo-log** an **undo** instruction which reverses the action. In fact, the undo-log is in reality a stack of undoing actions. All outputs that correspond to unidirectional state transitions in the plant require a corrective action distinct from the original action.

There could be a message on the display of the user inviting her to repeat the question.

A bottle filling chain could be brought to an initial state by a rollback procedure, removing half-filled bottles, resetting the bottle counters, and relying on some human help to remove the stuck pieces.

- If the environment is irreversible, then it is best to prevent actions on it, and leave repetition to **roll-ahead**. Unfortunately, most actions on the real world are irreversible, either physically (such as firing a rocket) or because the correction involves too much work (such as deleting a file). Erroneous actions to irreversible environments must be prevented, and cannot be neither repeated nor left undone. Therefore, irreversible operations require:

a fail-stop behaviour of the computer in order not to do any false action

a perfect tracking of the interactions in order to prevent loss or duplication of outputs and inputs at roll-ahead. Alternatively, the action must be performed atomically with the taking of the recovery point, like in the COPRA computer, as we shall see below.

6.5.3 Roll-ahead of the Environment

As a simplification for roll-ahead, we will assume that every part of the environment which received false data has been rolled back. The simplest way to achieve this is not to let out any false data (fail-stop).

We consider now the repetition of interactions at roll-ahead. The problem we face is that one cannot be sure how far the failed process progressed before failing. Therefore, loss or duplication of output or input can result.

If one repeats an output and the failed processor already did it, then a duplicate appears which can cause a failure. If one does not repeat at roll-ahead, and the failed processor did not reach the output instruction, then the output is missing, which also may cause a failure.

Finally, there remains the problem of data received while the computer is in the process of recovery. Here, one must count with data losses or else rely on a mechanism that can perfectly track interactions with the environment. This can be achieved by warm-stand-by, but not by cold-stand-by.

The rules to follow during roll-ahead are quite simple:

Input: if a task received inputs from the outside world before it failed:

I1- the source must be readable several times with same effect
 I2- or the read information must have been tracked
 I3- or the sender must also be rolled back so as to repeat the sending.

Output: if the task made outputs to the outside world before it failed,

O1- the receiver must not care about repetition
 O2- or the actions of the failed processor must have been tracked
 O3- or the receiving task must be rolled back.

Note that the input and output rules with the same numbers are symmetrical. Next we will look in detail at these rules:

6.5.4 Idempotent Input and output

Rule 1:

If the input can be made by idempotent reads, and the output made by idempotent writes, then roll-ahead just consists in repeating the original computation

Input Rule 1:

"Inputs shall be idempotent"

This is the case when the computer reads from ADC converters, status registers, or files, and every other time the information read does not change when input is repeated. This is obviously not the case when reading the registers of serial ports.

The data value read the second time may be different from the value read the first time, if this does not affect the algorithms or lead to data losses. For instance, if one reads a temperature for a regulation task, it makes no big difference if one reads the temperature one second later, even if it changed in between.

The condition is however that the value of the data read may not influence the execution path. For instance, there must not exist any program branch that depends on the value read. If this were the case, the second execution could depart from the failed execution track and the stable storage could not be corrected any more.

Therefore, it is necessary to insert a recovery point between reading a value and using its results in the program. An example has been seen with the input rules of the COPRA computer [Meraud 76]. Clearly, inputs are not repeatable when **events** should be registered. Such is the case for instance when a photocell signals to the controller that a bottle just passed by a detector.

Output Rule 1:

"Outputs shall be idempotent"

Doing more than once has the same effect as doing it once. This is often the case for the output to digital-analog-converters or to set-points of analog controllers. When the receiver is assignment-reversible, it can even tolerate faulty outputs.

Example: Outputting a set-point of -3500 K to a furnace will do no harm if the correct set-point of 600 K is issued within a short time.

In some cases, the receiver is equipped to handle duplication, for instance duplication of messages: each message carries an individual sequence number. If the message is repeated, it gets the same sequence number as the first one, and the receiver will discard it. Thus, the idempotency is implemented by the construction of the receiver.

6.5.5 Traceable Input and Output

Rule 2:

If idempotent reads and writes are not possible, it is necessary to prevent duplication or loss. This can be achieved by keeping track of every interaction the failed process had with its environment since the last recovery point.

The objective of this second rule is to make input and output operations **durable**. Durability means that once an interaction has been completed, it will not be repeated. Rather, its results will be used instead of the actual interaction. This is an extension of the notion of durable operations, which cannot be repeated once they have been successfully completed. Rather, an attempt to repeat them will cause the operation to be replaced by its result: calling it just updates the state with the corresponding data. Durability assumes that all interactions can be perfectly tracked.

Input Rule 2:

"Non-idempotent reads must be durable"

All data received by the failed computer since the last recovery point is buffered in a safe place, called an interaction log or **journal**. At roll-ahead, whenever an input interaction is encountered which has already been done, roll-ahead reads the results of the read from the interaction log instead of performing an actual read from the environment. Once the journal is void, the program then keeps on with the data received from the environment. This mechanism assumes that the order of reads is the same in the first and second execution.

The interaction log may be maintained in stable storage by the processor which is subject to failure, as is done for instance in word processors (see below). If a failure between the reading and the storing is feared, then an independent receiver that tracks the input stream of the working unit must record the reads. This tracking has the additional advantage that it allows unsynchronized communication, i.e. it can record events which otherwise would have gone lost during recovery, but perfect tracking requires **warm stand-by** hardware.

Example: A fault-tolerant word processor could intend to rollback to the latest instruction. Then, the whole state of the editor is part of the relevant state, including the position of cursors, the page being displayed, the search parameters, etc... Some 10K of information should be saved at every rollback point to perform recovery during the work of the editor. The approach taken traditionally in text editors is to consider the state during the execution of the editor as irrelevant. Only the state that prevailed before the editing session is reconstructed, and the editor must be restarted manually. Since this would involve losing completely the work done since the beginning of the session, the restoration work is done by a

***journal** that records (e.g. in a file) all entries done by the user since the starting of the session. At recovery time, the editor is restarted, the initial file is loaded and journal is played to the editor as though it would be a user input. Then, when the journal has been played, the input is switched back to the user's console. This technique is not identical to the redo-log method, though, since the original program itself does the recovery. The journal is just an interaction log of the input: it speeds up roll-ahead by making superfluous the rollback of the user.*

Output Rule 2:

"Non-idempotent writes must be durable"

When the output is not idempotent (for instance it consists of incremental orders to a stepping motor, or service messages in a network), then a monitoring of outputs must be performed by an OUTPUT LOG. Perfect tracking of the output requires an independent unit since the same unit is amnesic of all what occurred since the last rollback point.

At roll-ahead, already executed writes will be just skipped (there is no state update like there is for a read). This is done by consulting the output log. The monitoring memory needs not compare the output messages with the messages which already were sent: since one assumes that the second execution will follow exactly the footsteps of the first, a simple counter is sufficient. The counter is loaded initially with the number of output operations performed since the recovery point. The counter inhibits the outputs until as many values have been output as the first execution did, then the outputs are enabled.

The mechanism used to implement durability of input and output requires that roll-ahead follows exactly the same execution track as the failed execution, and this can only be guaranteed if the input stream is the same for roll-ahead as for the failed execution and if execution is purely deterministic.

If the roll-ahead could follow a different track, then the monitoring unit must have a great deal of intelligence: it must recognize duplicates which may come in a different order, and which may have different contents. The problem still remains if values are output which the failed execution was not supposed to send, or values are not repeated which the failed execution did already send. This is a case which can only be handled by forward recovery.

In short:

Durable interactions require that the execution path at roll-ahead be identical to the execution path followed during the failed operation

Or stricter:

The input values received during the roll-ahead operation must be identical to the input values the task has received in the failed execution.

6.5.6 An Example of Interaction Tracking: Auragen System 4000

Auragen's System 4000 [Glazer 84] has a warm stand-by architecture similar to the TANDEM 16's (Figure 6-21):

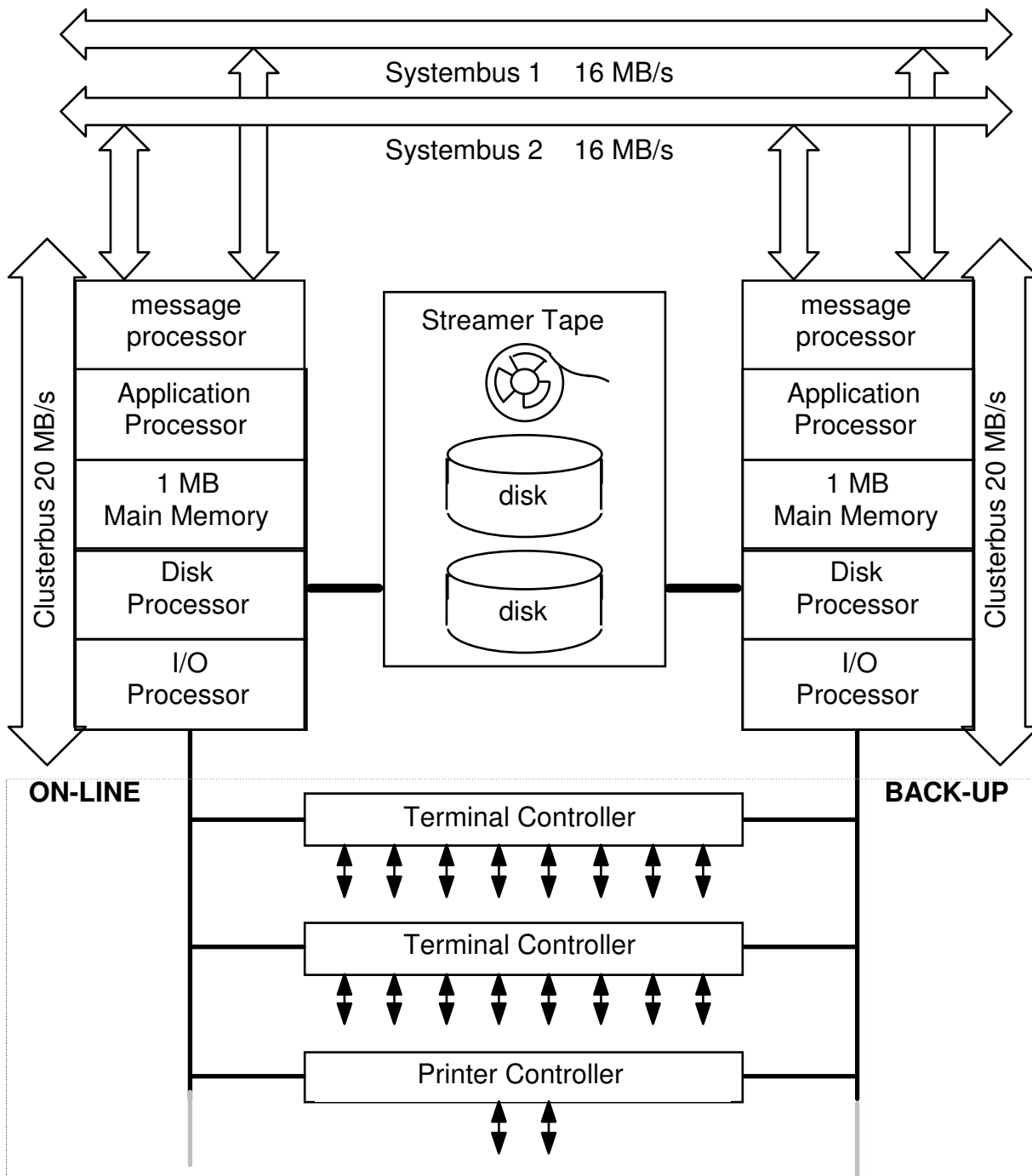


Fig. 6-21: Auragen's System 4000.

Auragen's 4000 consists of a number of nodes (clusters) which communicate over a high-speed (16 Mbyte/s) SystemBus. This bus is duplicated for availability, but conceptually, one has to deal with only one transmission path.

Each on-line node owns a back-up node, which lies in a physically distinct place. Up to 16 on-line/back-up pairs can be connected to the SystemBus. The back-up node may be performing other tasks than the on-line node for which it stands by. The distribution of tasks among the processors must ensure that an on-line task and its back-up task are not located on the same unit, since each node is assumed to fail as a whole.

The AUROS operating system implements recovery: a state saving (called "checkpointing") is done automatically at regular intervals (called "synchronization points"), at which the state of the working task is communicated to the back-up task. The taking of save points is usually triggered by a timer (but can be explicitly by the programmer). The programmer must not care about the interaction with the outer world since:

- The processor is fail-stop (memory and buses are protected by error detecting codes, the processors (Motorola's 68000), are built as self-checking pairs, so no false results will leak to the process.

- All interactions with the environment take place through messages. The protocol ensures that no loss or duplication of messages occur. When a message is sent by an on-line task, it is received by three destinations: the destination task and the back-up of both the sending and the destination task. The interaction is monitored in the communication processor of the back-up unit: the stand-by maintains a fifo (queue) of all messages received by the on-line unit, and a counter for all messages which the on-line unit did send. This interaction fifo is called a "queue-and-count" system by its designers.

The queue-and count mechanism is shown in Figure 6-22:

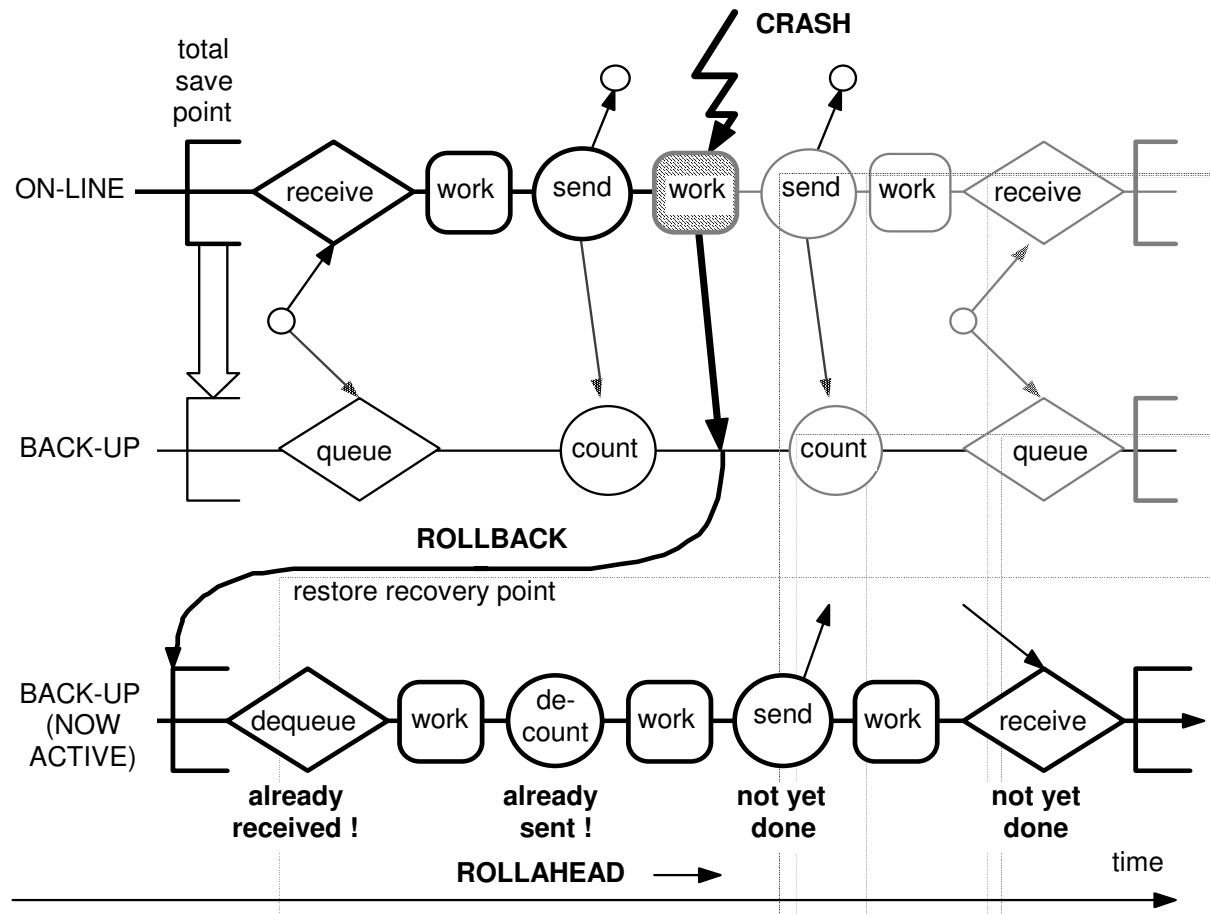


Fig. 6-22: Queue and Count in AUROS.

Upon detection of an error, the back-up starts with the state as it received it at the last checkpoint ("synchronization point"). There is no redo phase. The back-up restarts computations at that point and the interaction with the outer world is taken care of during roll-ahead.

When the program encounters a "receive" instruction during roll-ahead, it reads first the interaction fifo to see if the message has already been received. In this case, instead of reading the messages from the environment, they are read from the interaction fifo.

When the program encounters a "send" instruction during roll-ahead, it reads first the counter to see if a message has already been sent. If this is the case, no sending is done and the counter is decremented. When the counter reaches zero, the message is actually sent.

The receive/dequeue function, resp. the send/decount function is implemented by the same driver. For the programmer, the back-up task is just a copy of the on-line task. The difference is made by a status register in each node, which tells the operating system whether the task is executing as on-line or as back-up during roll-ahead. To off-load the application processor, all interaction tracking is done by the communication processor.

This of course assumes that the execution takes the same path the second time as it did the first time, which should be the case since the system is fail-stop. This mechanism also supports concurrency without domino effects (see next section). It is efficiently handled by a communication co-processor located in each node.

6.5.7 Rollback of the Partner Task

Rule 3:

If input and output are neither idempotent, nor durable, the corresponding sender or receiver must be rolled back.

This situation applies especially to the problem of cold stand-by, since warm stand-by can monitor the activity of the working unit and implement durable interactions, as we have seen with the Auros system. It applies also to warm-stand-by units which do not monitor the activity, like in Tandem systems: data which have been received during the recovery are lost. In case the partner is a parallel task running on another computer (or on the same in case of multitasking), roll-ahead can only continue when it arrives at an input/output instruction by first rolling back the sender task (Figure 6-23).

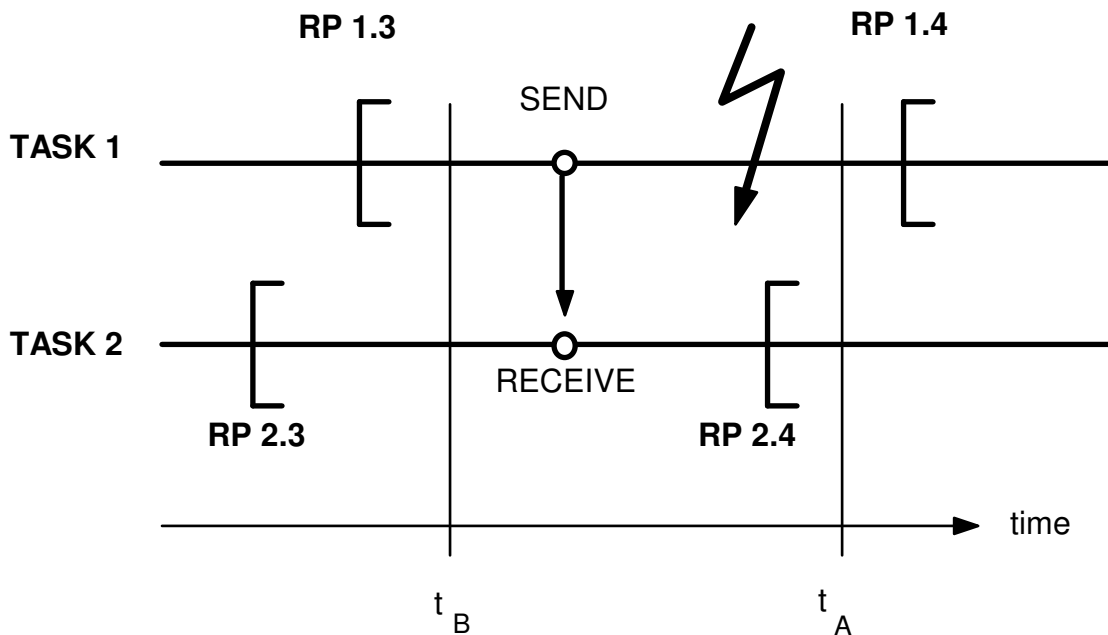


Fig. 6-23: Propagating Rollback.

In principle, one could have rolled back both tasks at the same time when recovery was initialised, and not first roll back one task and then the other during roll-ahead. But at the last recovery point, there was no mean to tell which interactions the task would do until the next recovery point, since this would require a prediction of the task's execution path. Therefore, it is necessary to propagate rollback at roll-ahead.

Consider a crash of task 1 which occurs at time t_A . Since task 1 communicated with task 2 since its last RP, task 2 will be forced to roll back when task 1 rolls ahead. This is easier said than done. Firstly, although it is fairly obvious from the drawing that task 1 crashed AFTER it communicated with task 2, task 2 cannot know it. Task 2 must be informed of the crash of task 1 before it reaches RP 2.4. After it passes RP 2.4, task 2 will not be able to return to RP 2.3 (except if several rollback states are kept, but this only shifts the problem until exhaustion of the redo-log). Therefore, synchronization is required: both participants in a communication should not be allowed to reach their next rollback point before both of them are ready to do so. This restriction severely limits the parallelism.

As an improvement, task 2 could ignore the rollback order if the crash occurred at time t_B , because the communication did not yet take place. Task 1 cannot distinguish whether the communication effectively took place or not, since it is amnesic of what happened since the last RP.

The probability of having to propagate rollback is minimized if tasks in cold stand-by systems take a recovery point or save the input log just after an input or an output is done. There remains a small probability of a crash between the reading of the data and the taking of the save point.

Both cases can be dealt with by letting the receiver acknowledge each data item after having saved it. Figure 6-24 shows a typical conversation between two tasks. Note that the fault is most of the time a communication failure. Then rollback is normally not automatic, but programmed by the user. Programmed rollback is more efficient than using the mechanism implemented for the case of a crash. The other task is ordered to rollback either explicitly (by a Negative Acknowledge signal or NAK) or implicitly (by a time-out).

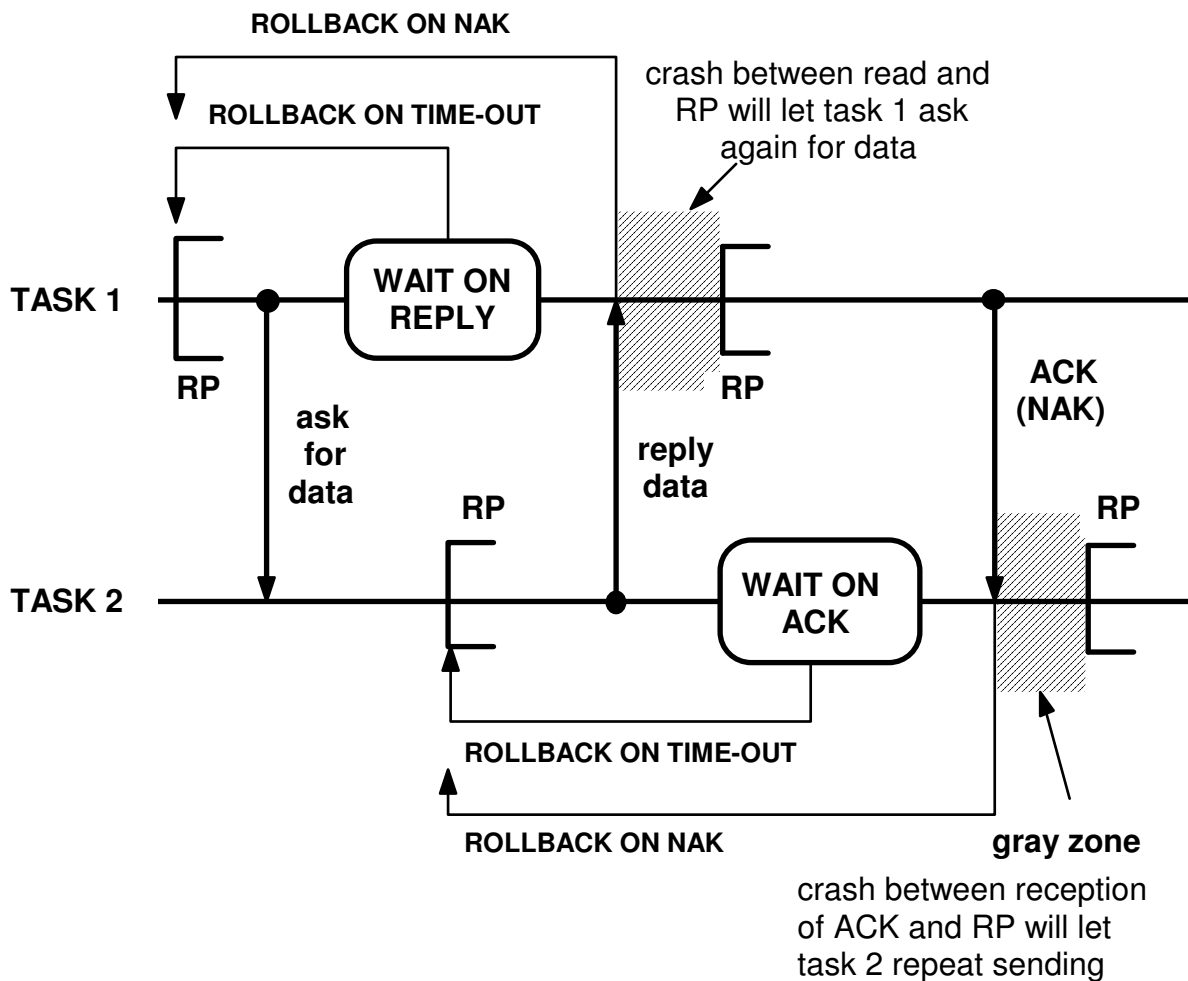


Fig. 6-24: Conversation between two Tasks.

Unfortunately, the time between reception of the data and sending of the acknowledgement may be too large in real-time systems, so that one normally acknowledges only the reception, not the taking of the recovery point. Then, a crash may leap in between and lead to an unrecoverable failure.

We will now generalize this case to a system of communicating tasks.

6.5.8 Rollback of a System of Communicating Tasks

In the above, we only considered the state saving and the restoration of a single, assignment reversible machine. We consider as a generalization a network of (reversible) computers that communicate with one another. The same problem may occur in a multitasking single node, when task recovery occurs on an individual task-by-task basis: a multitasking operating system is just simulating a multiprocessor system.

If a task that failed had communicated with another task since the last rollback point, both tasks must be rolled back. If the failed task did a read operation, the sender of the information should be rolled back to repeat the read again. If it did a write, the sender should do the write again. Therefore, rollback can be propagated from one faulty node to a quantity of other nodes with which it communicated. Figure 6-25 shows the execution of a system of communicating tasks as a function of time.

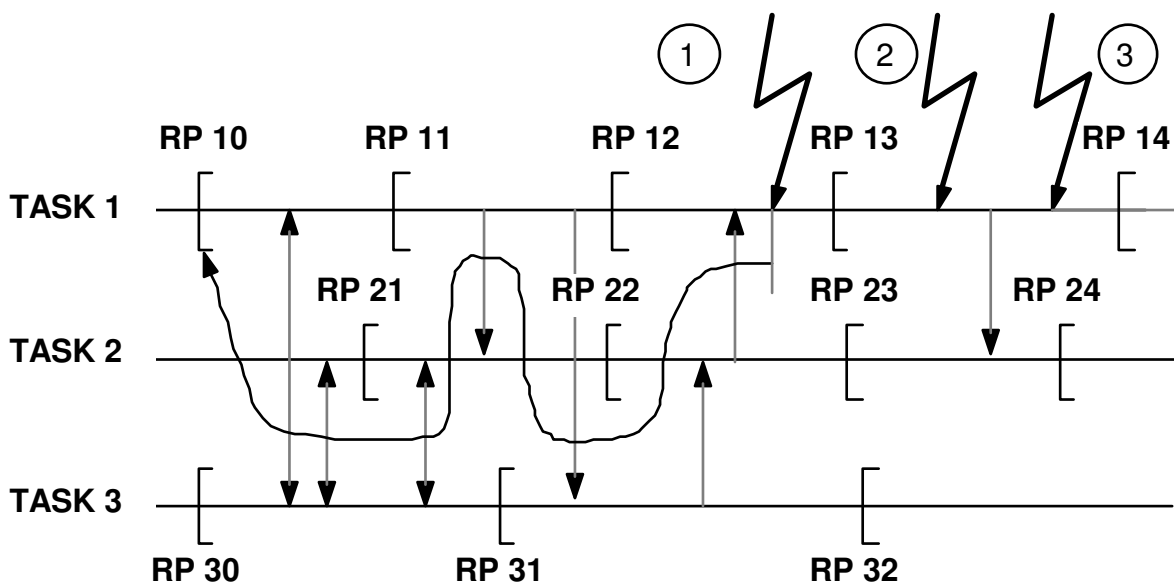


Fig. 6-25: Rollback of a System of Communicating Tasks

Note that the rollback points are asynchronous, i.e. every task takes an RP as it likes. The communication consists of messages, and, to simplify, we consider that all communication involves a buffering (the signal is not lost if there is no receiver waiting) and is acknowledged (the sender waits until the information is picked up).

If the rollback points are poorly chosen, a rollback may cause another to occur.

Consider a crash at place C: Task 1 rolls back to position RP 1.2, but since it communicated with task 2, task 2 retires to RP 2.2. In the meantime, task 2 had communicated with task 3, which must return to RP 3.1. Unfortunately task 3 communicated with task 1, which must return to RP 1.1, and so on, until all task are reset to the origin state RP 1.0, RP 2.0 and RP 3.0. This is called a **domino effect**.

To prevent the domino effect, the recovery points must be carefully chosen. Communicating tasks are only allowed to pass their next rollback point when all tasks with which they have communicated since the last RP are also ready to do so. The system of communicating tasks is known as a **conversation** [Randell 75]. The reaching of the common rollback point involves a synchronization between all members of the conversation. The taking of the common rollback point is a **commitment**. Once a **commit point** has been reached, no more rollback is possible.

The practical importance of this technique is not so high as the theoretical work done on it would lead one to believe. There are easy measures against the domino effect. The simplest consists in taking an RP before each communication, or better, to embed the recovery point in the communication primitive. This is a question of how much one is willing to pay for taking recovery points.

Implementing durable interactions by means of an input/output journal is an easy way to avoid the domino effect, since the partner task must not be rolled back. It even works if the task erroneously asked for information. In this case, though, some means must be found to discard the information that has not been consumed [Russel 80].

6.5.8.1 REFERENCES

- [Glazer 84] S.D. Glazer,
"Fault-tolerant mini needs enhanced operating system",
COMPUTER DESIGN, pp. 189-198, August 1984.
- [HORNING 73] J.J. Horning & B. Randell,
"Process Structuring",
ACM Computing Surveys, Vol 5, No. 1, pp. 5-30, March 1973.
- [KRINGS 85] L. Krings,
"A fast recovery mechanism for fault-tolerant computers",
Brown Boveri Research Report, 1985.
- [KUBRIAK 82] C. Kubriak,
"Penelope: a Recovery Mechanism for Transient Hardware Failures and Software Errors",
FTCS-12, 12th Int. Symp. on Fault Tolerant Computing, Santa Monica, pp. 127-133, June 1982.

- [MERAUD 79] C. Meraud, F. Browaeys,
"A new line of Ultra-Reliable, Reconfigurable Computers for Airborne and Aerospace Applications",
AGARD - Symposium on advances in guidance and control systems,
Ottawa 8-11 MAY 1979.
- [RUSSEL 80] D.L. Russel,
"State Restoration in Systems of Communicating Processes",
IEEE Transactions on Software Engineering, Vol SE-6, No. 2, March 1980.

7 Recovery in databases

7.1 Introduction

The previous chapters considered techniques for masking errors and for recovering from them within a computer. In summary, if the computing elements are fail-stop, then no error should ever leak out. This can be achieved either in the case of masking or in the case of recovery. Recovery just requires some precautions to handle the interaction with the environment, and especially to make interactions durable, i.e. to avoid redoing an operation already done, and to be sure of doing every operation which has not yet been done. This last requirement can only be met partially by cold stand-by, but warm stand-by can provide full durability of interactions, since it is capable of tracking them.

This chapter considers recovery in databases. It is an extension of the previous chapter. Commercial databases manage huge quantities of data, normally residing as files in a disk "farm" consisting of numerous disk units, which can be accessed simultaneously by several users. Typical applications are airline or railroad reservation systems, warehouse management and banks.

In process control, databases are the heart of the SCADA-function (System Control and Data Acquisition). Up to 10'000 process variables must be kept on-line for inspection by the plant's operator. These process variables are continuously updated in the background by the process. Process control databases generally reside in memory for fast access (300 ms in 90% of the cases), while the disk contains mostly history and static data.

Since most of the work done on recovery concerns commercial databases, we will refer to them firstly. The same principles hold for process databases. Database recovery is in principle similar to the recovery of any computer consisting of a volatile storage and a stable storage, so the techniques we have seen in the preceding chapter are applicable here. The new aspects we wish to consider are:

- Databases support transactions, which are a sequence of actions executed as a whole or not at all
- Operations on the database may be voluntarily undone, and the mechanism to undo a sequence of operations must be fast.
- the computer relies on cold stand-by.

The last requirement is mostly a question of hardware costs: warm stand-by requires a second machine and a complex updating mechanism. Cold stand-by relies solely on stable storage and logs, but costs additional complexity during operation and recovery. We will first consider cold stand-by, the extension to warm stand-by being trivial.

Recovery in databases is achieved by combining several techniques for state saving and restoring which account for different kinds of faults. In addition to the problems of recovery, the problem of concurrency occurs since the database is shared among several users.

Whether all activities take place on one computer or on several does not matter as long as the database is centralized. If the database is distributed, i.e. replicated over several nodes (each node holds a copy of the database) or partitioned (each node holds a part of the database), then an underlying protocol cares for the consistent update of the replicated files.

7.2 Concept of actions and transactions

Any computation progresses through a sequence of actions, which maps the state S_i to the state S_{i+1} . The effect of an action is to create, destroy, modify or inspect objects.

Depending on one's view of the computer, one can consider an action as one machine instruction or as a complex operation involving thousands of disk accesses, like reorganizing the database.

An **atomic action** is an action that is either done completely, or not at all. A write to a memory location is considered as an atomic action at the hardware level. When an action takes a certain time, and especially consists of several steps, then special precautions must be taken to realize an atomic action.

Atomic actions are the building blocks for fault-tolerant systems. Atomic actions are operations which are either correctly done or leave the system in the original state. This implies that an atomic action has no effect on the outer world until it is completed.

This also implies that an atomic action that did not finish can be restarted at will. Once it is finished, it should not be repeated any more, since its effects are permanent.

Atomic actions are widely used for **transactions** in databases. A transaction progresses by modifying a database through a sequence of ACTIONS. All actions belonging to a transaction must be executed as a whole, regardless of any intermediate interruptions. A typical example is a booking program that transfers a certain sum from the account of a customer to the account of another customer.

For instance, one action is withdrawing 100 Swiss Francs from Meyer's account and another is crediting them to Martins's account. Both actions must be implemented as a whole. Repeating or omitting any of these operations would lead to a failure.

A transaction knows two possible issues:

1. the transaction is led to a successful conclusion and then **committed**. The committing of a transaction makes its effects permanent. A committed transaction cannot be repeated - it would then be a new transaction.
2. the transaction is **cancelled**. This can be a voluntary operation (for instance when the client of the airline changed his mind after half of his flight has been booked), or involuntary (for instance due to a crash of the machine, a deadlock situation or the impossibility of finishing the transaction, for instance when two passengers compete for the last seat). When the transaction is cancelled by the system or a crash in an unordered way, the transaction is said to be **aborted**.

A **durable action** is an atomic action which cannot be repeated, but which will be replaced by its effects if repeated for recovery purposes.

- for instance, if the computer crashes after the transaction committed, means must exist to redo the transaction without asking the clerks to enter the dialogue again. Instead of repeating the transaction, the result of it must be saved in a safe place and used to restore the state. The individual actions of a transaction must NOT be durable until the action is committed - if the transaction is cancelled, their effects must be undone.

Finally, actions must be **consistent**. Transactions may execute in parallel with other transactions, for instance in a multi-user environment (pseudo-concurrency) or on different computers which access the same database (real concurrency). In both cases, there are certain files that can be shared among the transactions, either for reading or for writing. A locking mechanism must ensure that shared objects be modified in a consistent way. Therefore, an action must be consistent, and the accepted criterion for consistency is that an action be **serializable**, i.e. executes as if it had been done alone on the machine, before or after any other transaction, although in reality it may have executed concurrently with other transactions. Consistency is ensured by an appropriate **locking** mechanism: a transaction holds a lock for each object to which it needs exclusive access. For instance, a two-phase locking strategy can be followed, in which each transaction gathers the access rights of the objects it wants to access, operates on them and then releases the locks in the opposite order.

All practical locking schemes can cause the transaction that asks for a lock to be cancelled:

Pessimistic locking schemes first ask for all locks that the transaction may require during the transaction, perform the transaction and then release the locks. A transaction can then be cancelled because of deadlock (two transactions asking for the same locks in reverse order). Also, pessimistic locking does not allow to add locks which were not foreseen at the beginning.

Optimistic locking schemes allow a transaction to access the objects without holding a lock. In most cases, there is no conflict between transactions, and the transactions complete normally. In some cases, however, a conflict occurs, and all involved transactions must be cancelled and restarted. Optimistic locking is similar to collision arbitration, while pessimistic locking is more related to token arbitration. Optimistic schemes are on the average faster than pessimistic locking schemes

The locking mechanisms are beyond the scope of this tutorial. They can be found in Bernstein's tutorial [Bernstein 81].

7.3 Model of a database

We can consider the database as consisting of three parts: the memory storage, the disk storage and a log of activities which is used to reconstruct the database in case of failure:

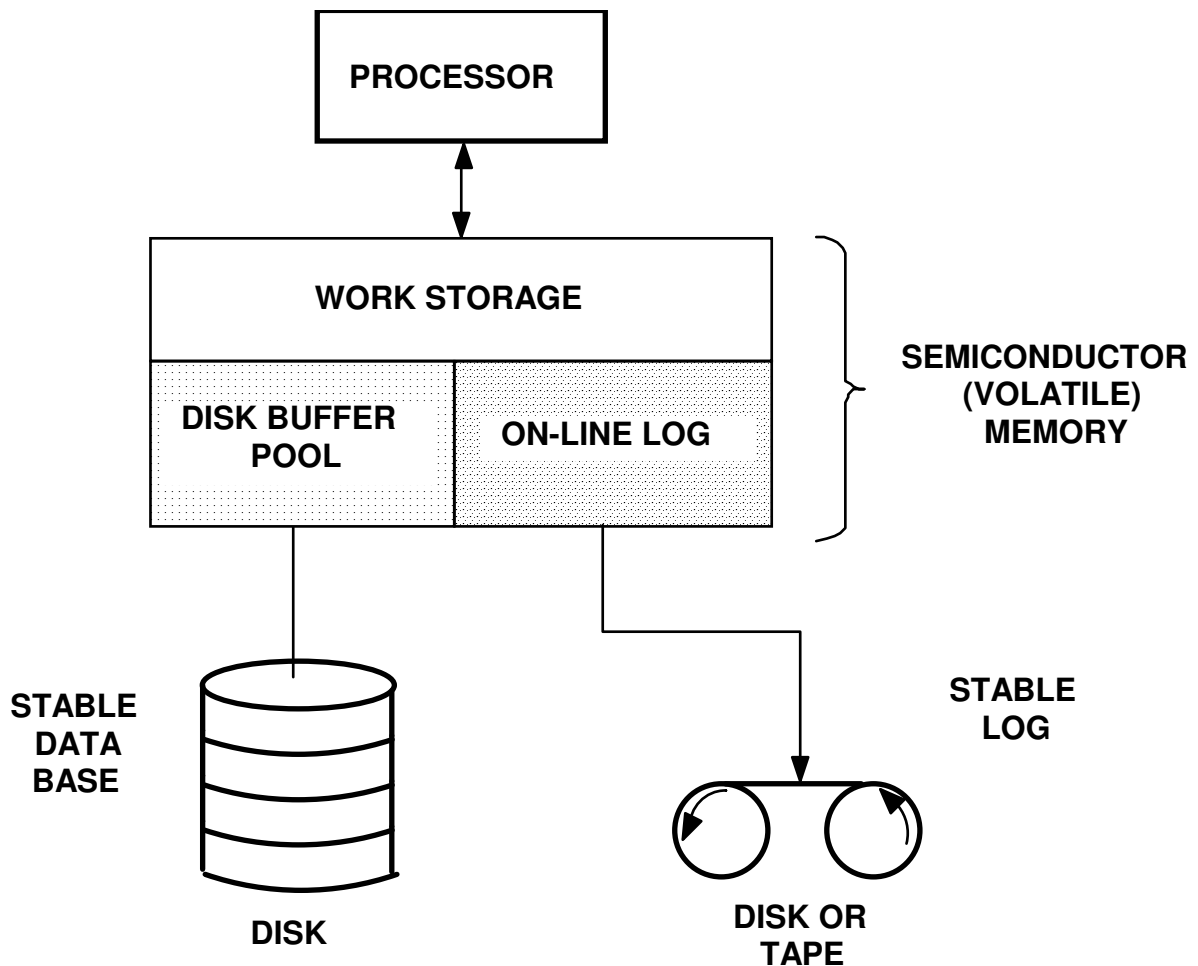


Figure 7-1: Model of a Database Computer.

Each transaction runs in its own memory space within the computer. A multitasking operating system could simply assign one task per transaction, the memory management ensures that the address spaces remain independent. In fact, the operating system lets each transaction believe that it runs on its own processor with its own address space.

Each transaction progresses by modifying objects (records on files) in the database, which is common to all transactions. Some of these objects are shared by several transactions (for instance the directories).

The database consists of files that are normally located on disk. Files that are currently opened for read or write are buffered in an area in memory called the **buffer pool**. It is much faster to access and manipulate files in RAM than on disk, especially for database operations (search and random access). Not all of the file is buffered: a file is divided into fixed-length blocks called **pages**, which are read from the buffer on demand. A changed page is not immediately copied back to the disk, since it could be used further locally. Meanwhile, the file on disk is inconsistent since some of its modified pages have been written back to disk and others still reside in the buffer pool.

Neither is a file consistent when it is closed: other users could need access to that same file and its modified pages still reside in buffer pool. **Propagation** of a modified page to disk is done either by a special instruction "flush pool", or by the operating system itself when it runs out of buffer pool space. In the last case, the least recently used page (LRU) is propagated to disk. The programmer of a transaction has normally no influence over which pages are currently in pool.

We will see that this buffering complicates recovery significantly. If memory is erased in a crash, the currently opened files will remain inconsistent.

7.4 Failure modes in a database

We consider four kinds of failure: set-backs, cancels, crashes and media failure. The first two affect only one transaction and the other two affect all transactions. The terminology and concepts are adapted from [Gray 78] and [Haerder 83].

7.4.1 Failures which affect one transaction

We consider:

- Transaction CANCEL (abort or transaction failure),
- Transaction Set-Back (action failure),
- Crashes (computer failure),
- Media Failure (failure of the mass storage).

7.4.1.1 Transaction Cancel

Cancelling a transaction erases all effects of this transaction (see Figure 7-2).

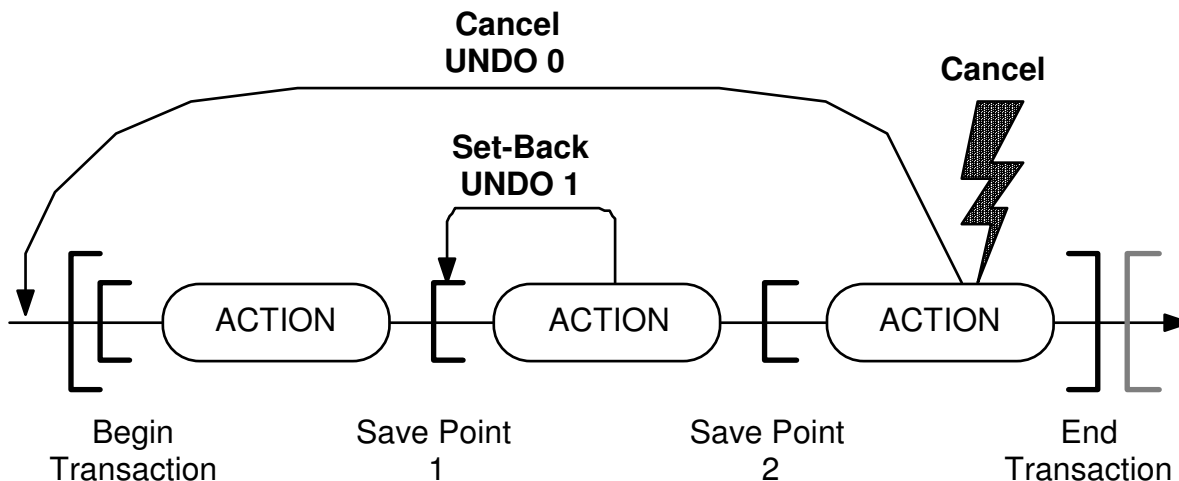


Figure 7-2: Transaction Cancel and Set-Back

Cancelling can be voluntary and originated from the transaction itself because it either cannot finish or human errors have been committed. It can also originate in software errors or in database mechanisms as a measure to break deadlocks. Finally, cancelling can also be due to a crash, but this will be discussed later.

To give an order of magnitude, normally about 3% of the transactions are cancelled for some reason. Cancels occur at the rate of about twice a minute in a large database. Transaction cancels are handled by UNDOING. Recovery only lasts some milliseconds.

UNDOING restores all objects which have been modified by the transaction to their previous values. The transaction state is irrelevant.

7.4.1.2 Transaction Set-Back

Because of small errors in a transaction, it may be necessary to undo the last actions, without undoing the whole transaction. A set-back may be voluntary, as in the case of an entry error, or to leave the way free for an alternate action. It can be called implicitly to deal with operating errors (access violation) or it can even be used as a programming technique.

To execute a set-back, the progress of the transaction is divided by firewalls called SAVE POINTS (see Figure 7-2). Save points are implanted by the application programmer by calling a SAVE procedure. At each save-point, the volatile state of the transaction is saved (typically up to 64 KB of information). The save points are numbered with save point 0 being the beginning of the transaction. The programmer invokes a set-back by calling an "UNDO n" procedure which returns the transaction to the n-th save point.

Set-backs occur about as frequently as cancels, i.e. at the rate of several per minute. Recovery is done by **backtracking**, which takes some milliseconds. Backtracking restores the state of the transaction state as it prevailed at the save point and undoes all objects that have been modified since that save point. One can also undo several save points at once.

7.4.1.3 Crashes

Crashes are due to computer failure, power supply outages, or severe operating system bugs. When a crash occurs the processor and memory state, including the buffer pool, are completely lost. All transactions running on this machine are lost. A crash does not affect the objects stored on disks. But these objects can be inconsistent because some of their pages, which are still buffered in memory, have been lost.

A crash affects all transactions which are currently in progress (Figure 7-3):

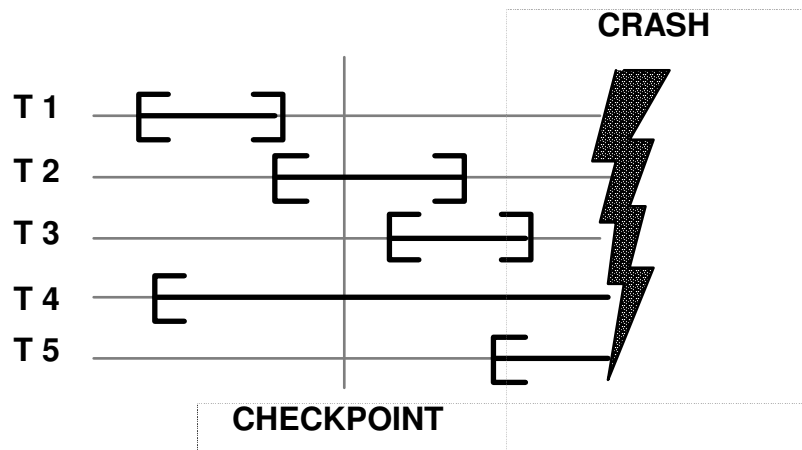


Figure 7-3: Crash

Crashes occur about once a week. Crashes are handled by a RESTART. A restart should last a few seconds.

A restart cancels all the transactions that were active at the moment of crash and redoes the work of all the transactions that committed before the crash. A committed transaction may need to be redone although it has already terminated, as its updates may not yet have been written back into the stable storage.

To avoid redoing old transactions, the execution is divided by firewalls, called **checkpoints**, which are inserted about every 10 minutes to minimize the work to be done at restart. So, in the above Figure 7-3, transactions T2 and T3 must be redone, and T4 and T5 must be cancelled, but T1 need not to be redone, since it committed before the firewall.

Note that a more elaborated technique would allow one to continue the transactions that were in progress at the moment of the crash, by backing them to their latest save-point. This is done in some systems, but costs more state saving.

Note: firewalls are usually called checkpoints, but this last term has different meanings (see vocabulary, 7.8).

7.4.1.4 Media Failure

A media failure is a heavy crash in which the state of the disks is lost, i.e. the disks are considered volatile. This can be caused by a double fault in redundant storage, by operator faults or by a severe software error. A media failure is not expected to occur more frequently than about twice a year.

Recovery is done by **reconstruction**. A reconstruction should be completed in less than one hour.

Reconstruction restores the state of the disk storage and performs a restart, cancelling transactions that were not committed and redoing transactions that were committed.

Reconstruction is made from an archival copy of the database, at a moment when no transactions are active, for instance early in the morning. This copy is a full back-up copy of the database. The back-up copy is guarded in a safe place and heavy reliance is placed on the log, which should be crash proof.

7.5 Recovering from database failures

7.5.1 State Saving

All four above failure modes: set-back, cancel, crash and media failure are handled by gathering the following information during normal operation:

- a FULL BACK-UP COPY of the database is taken at a time when the system is TRANSACTION-CONSISTENT, i.e. there are no transactions in progress (for instance at 3:00 am when nobody works). Thus, the copy needs not include the memory state since it is irrelevant. The back-up copy is kept on tape in a safe place. It is used to recover from MEDIA FAILURES.
- a LOG of changes to the database is continuously kept in an AUDIT TRAIL. This log follows all activities of the database. It must be very reliable, because if it fails, no recovery is possible. For this reason, the log is often duplicated.

The following information is kept in the log:

- 1) Each time an object is modified, its previous value (BEFORE-IMAGE) and its new value (AFTER-IMAGE) are recorded in the log, along with the identity of the transaction which did the modification (Figure 7-4)

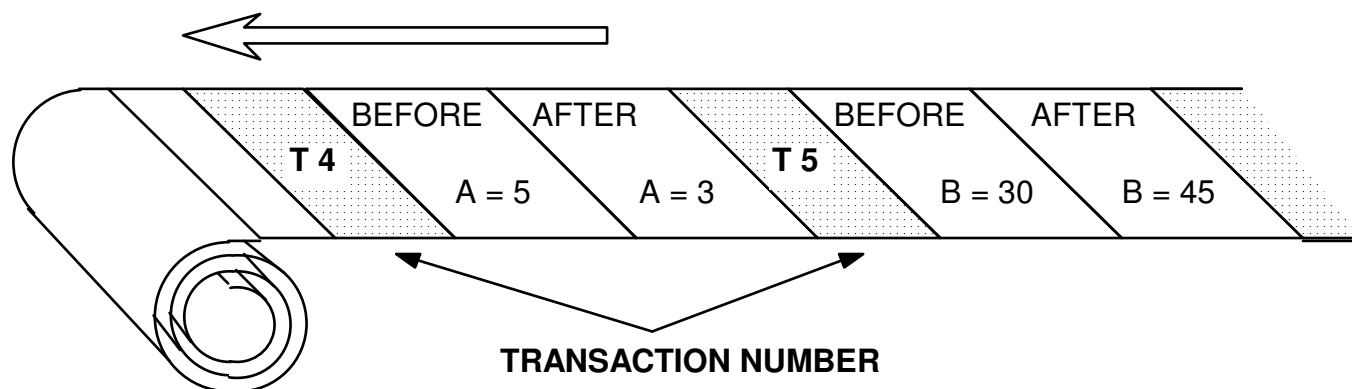


Figure 7-4: UNDO and REDO LOG.

The set of images builds an UNDO-LOG and a REDO-LOG, which correspond to the UNDO-STACK and REDO-FIFO respectively. The before-image is used to recover from transaction cancels, set-back and crashes. The after image is used to recover from crashes and media failure.

Each time a transaction begins or ends, an entry is kept in the log as: BEGIN&TRANSACTION T and COMMIT&TRANSACTION T.

CHECKPOINT information tells when a checkpoint has been taken.

Each time a SAVE-POINT is taken, enough information to reconstruct the volatile state is recorded.

Recovery is based on the stable state and on the log for all recovery modes, except recovery from media failures, which relies on the full-backup and on the log.

7.5.2 Transaction Cancel

On cancelling, the memory space of a transaction becomes irrelevant (since the system must be restored to the state it had before the transaction began). All objects which have been modified by this transaction since it began should be restored to their previous value.

The programmer influences the cancelling by three instructions:

BEGIN&TRANSACTION;

CANCEL;

COMMIT&TRANSACTION;

Note that CANCEL can be called implicitly, for instance by the operating system to break a deadlock.

Undoing trusts that the disk storage is not affected, the current state of the transaction is irrelevant.

To cancel a transaction, the log is read backwards, and each entry which corresponds to the cancelled transaction is extracted. An UNDO-LOG for this transaction is reconstructed. The before-image is used to restore the objects modified by that transaction to their previous value, until the BEGIN&TRANSACTION T mark is reached.

Of course, if the COMMIT&TRANSACTION T mark is found, no recovery is possible.

The undo-log must be read backward for recovery (it has the form of a stack). Therefore, it is natural that the redo log be kept on disk since tapes are difficult to read backwards. In fact, since cancelling occurs so frequently and must be quite fast, (some milliseconds), the logs are kept in memory. This complicates crash recovery.

7.5.3 Transaction Set-back

If smaller errors occur during transaction processing, it may not be required to cancel the whole transaction, but only to set it back to one of the previously set SAVE-POINTS.

In case of set-back, the current transaction state is abandoned. Backtracking must restore the stable objects which have been modified since the save point and reload the transaction's state with the state saved at the save point.

The programmer has two instructions available to handle set-back:

SAVE n; and

UNDO n;

At each save-point, the system saves the transaction state (which can amount to 64KB).

Backtracking trusts the disk storage and the log. The log is read backwards, in the same manner as for cancelling. Each object that has been modified by the transaction is restored to its previous value. When the n-th save point is reached, the state recorded in the log is reloaded into memory and the transaction can continue from this point.

7.5.4 Crash Recovery

The value saved at each save point is not sufficient to restore the system state after a crash. Further, there is no obligation for a programmer to take save points.

Restart trusts the disk state, but the memory space of all transactions is lost.

A restart should undo all transactions that have not yet committed (transactions T4, T5, called LOOSERS) and redo all transactions that have committed (transaction T1, T2, T3, called WINNERS).

Undoing is similar to transaction cancelling, with the difference that it affects several transactions, more precisely all losers. Then the winners are redone.

Note that the redo would be unnecessary if every update of an object was propagated to the stable state BEFORE the action committed. Redo is made necessary because:

some updates are buffered in the memory pool

one must be sure that the log contains the before-image before making the actual updating because there is a probability of crash between writing the log and overwriting the object.

Since all transactions that have been done since the crash would need to be redone, a firewall is introduced about every 5 minutes. A firewall is just an operation that ensures that the memory does not contain any information relevant to the database.

The simplest way to take a firewall is to wait until no transaction is active and then to make an entry in the log just saying "firewall", with no special information saved. Then, the log would not need to be read back from this point.

If there are pages in the buffer pool at that time, then all pages from buffer should be first flushed to the database and then a firewall entry is marked in the log, so now the content of memory is irrelevant.

As a further complication, we depart from the former assumption that the firewall was transaction-consistent, i.e. taken at a time when the system is quiescent (no transaction in progress). This is the normal case, since a firewall should be taken about every 5 minutes and there are some hundreds of transactions active at the same time.

Then a checkpoint must be taken when transactions are in progress. We will therefore have to consider the recovery of transactions of the T2 and T4 type in Figure 7-3.

The method described in [Gray 81] is to take an ACTION-CONSISTENT checkpoint: At the beginning, a log entry is made which states "BEGIN_CHECKPOINT". All transactions are prevented from continuing after they complete their current action. Since each action in a transaction is quite short (some milliseconds), the system reaches rapidly an action-consistent state. The identity of the transactions which are currently active is logged, along with a pointer to their most recent log records. Then, all pages which reside in the pool are propagated to disk, so the files on disk are now consistent. Finally, an END_CHECKPOINT is entered in the log to signal that the firewall is completed.

At recovery, the log of the transactions to be undone is read back until the BEGIN_CHECKPOINT mark is found (there might have been some modifications between BEGIN_CHECKPOINT and END_CHECKPOINT because of unfinished actions).

This is not sufficient to restore the state, since some transactions were in progress at checkpoint time. Therefore, the log of these transactions must be kept so one can redo them. The log entries extends therefore to the time previous to the taking of the checkpoint, but there is no need to consider other transaction than those which were active at the checkpoint.

A more elaborated checkpoint could have prevented the cancelling of the transactions that were in progress at the moment of crash, by using the transaction save-points. The additional state information that must be saved must be traded against the probability of failure.

7.5.5 Media Failure

Recovering from a media failure requires that the database be restored from an archival copy. That is, a GLOBAL REDO from scratch must be made. For this, a before-image of the whole database must be made. This before-image is the full-back-up copy, which is generally kept on tape. The before-image must be transaction-consistent, i.e. no transactions should be in progress at the time the full back-up is copied. This can be done for instance at 3:00 am in the morning when nobody works.

At recovery, the back-up copy is played into the database, reversing it to the state it had at the time the copy was made. Then, the log is read from the moment of crash backwards, to separate transactions which committed before the crash (winners) and transactions which were active at the moment of the crash (losers), until the entry corresponding to the full back-up is reached.

Then the redo-log is read forward, and every entry corresponding to transactions that were committed at the moment of the crash will be restored. Transactions that were active at the moment of the media failure are by default cancelled. At the end of reconstruction, the state should be the same as after a crash. Indeed, the procedure is exactly identical to that followed after a crash, only that the database is first reloaded with the full back-up and that checkpoints since the full back-up are ignored.

7.5.6 Optimisation and Enhancements

There are several techniques that are used to reduce the time required for recovery:

7.5.6.1 Log Compression

The **redo log** is read **forward** to recover. It has the form of a FIFO. If the object has been modified several times, only the LAST value before the crash is interesting (it contains the valid version of that object).

The **undo logs** are read **backward** for recovery, that is, they have the form of a **stack**. If the object has been modified several times since the last recovery point, only the first value after the transaction save point is interesting (it contains the value of the object before the save-point, respectively. before the transaction begun).

Since only the last value, respective the first value are interesting for recovery, a **log-compression** routine can be run in the background which transforms the log by dropping the unnecessary entries, i.e. keeping only the first value of the undo-stack or the latest value of the redo-stack.

7.5.6.2 Checkpointing Consistency

It would be far easier to have just a firewall for recovery against crashes that tells one how far back one must redo a transaction. Then one could drop the logs previous to that point. One can construct a transaction-consistent checkpoint by awaiting the cancel or commit of each transaction that was active at the moment of checkpointing.

The artificial log is then used as a substitute for the actual checkpoint. This is especially of interest for databases which are on-line 24 hours a day, for instance international networks.

7.5.6.3 On-Line Log

Since rollback is such a frequent operation, it is interesting to maintain the most recent part of the log in memory. The problem is that this part of the log is lost in a crash. This by itself is not tragic, but one should avoid a situation in which updates to an object are written to the database BEFORE the corresponding entries of the log have been saved in stable storage. If this occurs, one will be incapable to recover since the previous value of that object will not exist any more. The solution is to force the log to stable storage before the updates are made. This is called the WRITE-AHEAD LOG by [Gray 78].

7.5.6.4 Shadow Files

Shadow files is a technique to speed up recovery from media failures. Each shadowed file exists twice physically, preferably on two different disks: the current version that is being updated and the shadow version that may not be updated. Therefore, one can survive a failure of a file which is not in the log, for instance one file which has been modified before the last checkpoint by a transaction which committed before the checkpoint (T1 type) and which has not been refreshed since. Basically, shadow files are redundant with the log and full back-up technique.

7.6 Summary of transaction recovery

The above techniques for database recovery show how rollback is actually implemented. One difficulty is to distinguish which storage is volatile and which is stable in a crash. Things would be easier if the operations were not delayed by the paging mechanism. The real challenge is to make such a mechanism efficient. Recovery should not cost more than some 5% of the system performances. It is however accepted that the redundancy of storage will be larger than 100%, especially when shadowed disks are used. Most of the problems deal with the fact that the above techniques use **cold-stand-by**. A complete recovery requires therefore that at least some entries be repeated, and at worst transactions be repeated after a crash. Most applications involve human operators which are trained for this.

Database recovery ensures that the committed transactions will not have to be redone and that their effects persist in spite of a crash or media failure. These techniques do not ensure that the transactions will be resumed some time later. For this, cold stand-by is not the proper technique. Indeed, recovery would be simplified by using warm-stand-by techniques. Therefore, the above techniques are not apt to implement highly available systems. At least, integrity can be guaranteed as long as no erroneous data is committed.

7.7 Recovery in distributed systems

Distributed transactions are one level in difficulty above recoverable transactions. A distributed transaction takes place on different physical nodes in a distributed system and is executed by parallel actions. The transaction is atomic when all of these parallel actions commit, or none of them. The problem of concurrency is not different from the concurrency within a database and can be treated with the same locking methods. The difference is that the concurrency is real, and that communication is unreliable.

The basic protocol to realize an atomic distributed transaction is called the two-phase commit in [Lampson 81] and [Gray 78]:

At the beginning, there is one dedicated unit that is the recovery manager. The recovery manager may be for instance the client of the transaction.

The coordinator takes a recovery point and then broadcasts a message: "Ready to Commit ?" to all concurrent actions working on behalf of a transaction, asking them to accept commit.

If the coordinator crashes shortly after, the recovery manager will start it at the last recovery point and let it send "Ready to Commit ?" again.

Each participating action responds by "Ready" or "Reject" or just remain silent for a while, either because it is not finished or because it has crashed.

If the coordinator receives a "Reject", it takes a new recovery point. Then it broadcasts the message "Cancel All" to all participants. If it crashes after that point, it will send "Cancel all" again.

If the coordinator receives a "Ready" from all participants, then it also takes a recovery point. From now on, it cannot repeat the "Ready to Commit" order any more.

The coordinator sends a "commit" message to all participants. Whenever a participant receives a "commit" message, it should commit, since this message can only come if all participants are ready to commit.

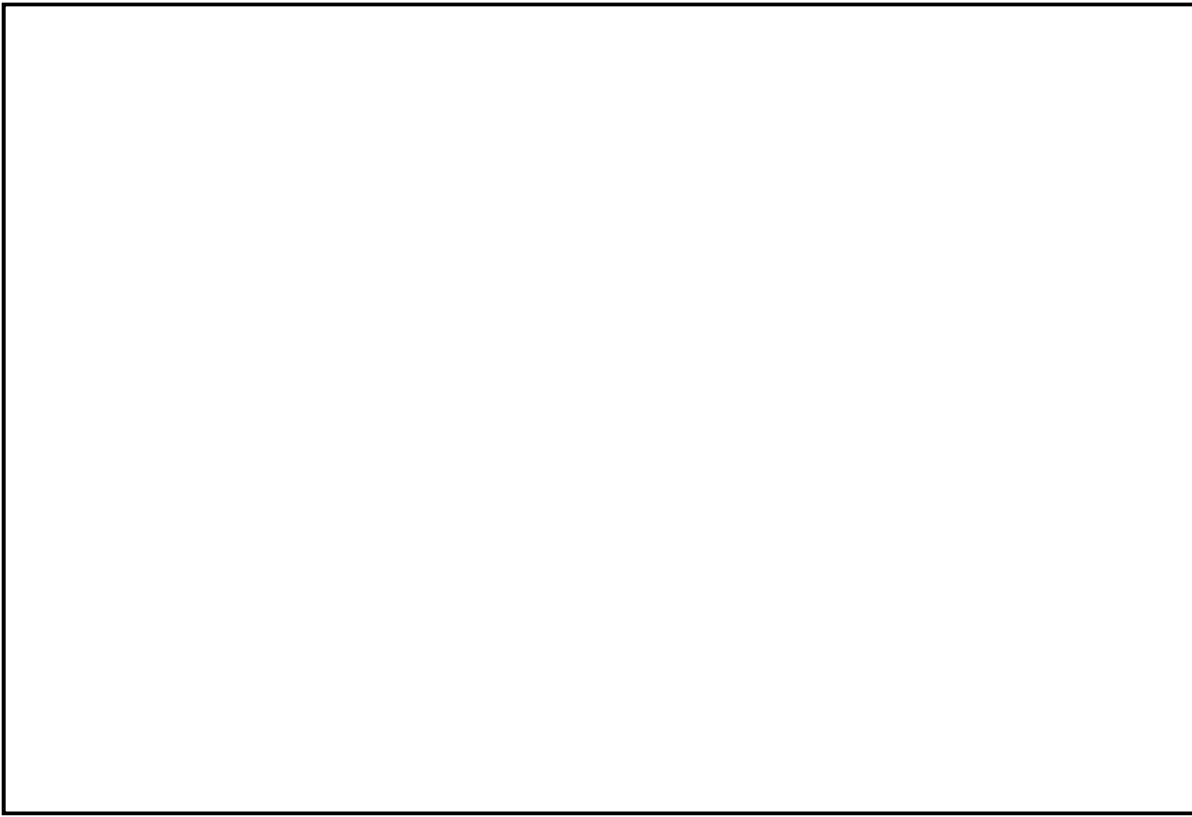


Figure 7-5: Two-Phase Commit.

Note: this method has been used in sailing for thousand of years to make sure that all sailors obey to the same command. Every sailor is supposed to answer "Ready" when the captain yells "Ready to come about ?". Only when all sailors responded "Ready" does the captain give the order "About ship!". Once the captain gave the order "About ship!", he cannot cancel it, but can only repeat it. He may however cancel the "Ready to come about ?" any time before, either because the manoeuvring is not any more opportune or because somebody is not ready. But once it's started, it must keep on.

Note also the similarity with the fire squad order "Arm, Aim, Fire". We have also seen the same method to achieve atomic outputs in safety devices, only that the physical process was the only partner.

The two-phase commit has also some flaws. It assumes for instance that every node will come up again in the transaction after a finite time to complete it. If it crashes after having replied "Ready", it must execute the command it acknowledged at a later time. This is not possible to achieve for a crash in the above recovery scheme which abort current transactions. A finer degree of recovery with the saving of the transaction state at checkpoint time is needed to guarantee that the failed transaction will eventually continue.

The second simplification we made was that all interactions were done by non-faulty units, i.e. all processors are fail-stop. This assumption holds quite well for hardware failures, but is rather difficult to enforce for software failures, since the coverage of software errors is lower.

The two-phase commit has been extended to three-phase commit [Bernstein 83] and n-phase commit, with increasing complexity and decreasing probability of failure. In fact, it can be shown that the problem has no solution which is 100% reliable within a bounded time.

The proof is known as the two army's problem [Gray 78, Yemini 79].

Two allied armies are camped at a certain distance and are separated by enemy lines. The generals communicate by messengers, which must first cross the enemy lines to reach the other army. These messengers risk to be caught by the enemy each time they try to reach the other army.

The armies have only a chance to win if they attack simultaneously, i.e. if they synchronize on a time for attack. Since attack needs a preparation time and an army may well be delayed after it already agreed on a time, the generals must agree on a protocol to start attack. This protocol relies on the unreliable messengers. If a protocol exists which works with N unreliable messengers, then the last messenger is irrelevant, since the protocol must work without him. Then, the protocol must also work with $N-1$ messengers, and the $N-1$ th messenger is not required, either.

7.8 Software issues

Until now, we have considered the mechanisms used for fault-tolerance. Now, we would like to expose their impact on the programming.

7.8.1 Programming Constructs for Recovery

The ambition of fault-tolerant system is to present an interface to the user which is identical to that of a non-redundant computer, so the application programmer and the user has nothing to do with fault-tolerance. This is called transparent fault-tolerance.

For instance, the inserting of checkpoints or save-points in an application program is not desired: this only reflects the fact that the system is not capable of providing an invisible fault-tolerance. The application programmer must think all the time whether it is opportune to insert a recovery point or not. Further, it makes the reliability of the system depend on the programming skills of the programmer, and extends the certification phase to the application program.

\$\$\$ Brown, Boveri's experimental network

The network developed at Brown, Boveri combines the process of communication and of taking of recovery points. Its architecture is similar to that of the Tandem or Auragen 4000.

The basic idea is that the internal state of a node does not matter as long as it does not communicate with the outside world. Therefore, the operations of state saving, interaction monitoring and communication should be combined in one message.

Each time a message is sent, it is received by three destinations, like in Auros. Each message carries piggy-back the update of the working task for its back-up task. The additional overhead is neglectable since the cost of sending a message is practically independent of its length. The message itself is received by the destination task and its stand-by task.

The mechanism of communication is the remote procedure call.

***** * To be extended * *****

[Aschmann 86] H.R. Aschmann Recovery in a Distributed Process Control System

LOCUS,

7.9 Vocabulary

We close by including some definitions often found in relation with databases, and which should help in scanning the literature:

- Audit trail:** a record of all activities that have taken place on a computer. The audit trail can be used for statistics, or for recovery.
- Log:** a record of operations that have been performed by a processor or its environment. Logs are generally written on disk or passed to a tape.
- DO-LOG:** an entry in a log that concerns a modification performed on an **object**. A **do-log** consists of an **undo-log** or a **redo-log** entry or both.
- REDO-LOG:** an intention list of the value that all objects should have if the next recovery point would have been reached successfully. By extension, a list of modifications to be applied to the last full back-up as a shortcut to computation.
- UNDO-LOG:** a log entry which records any information necessary to restore an object to its previous state, taking the present state as a starting point, in particular the previous value of that object.
- AFTER-IMAGE:** IBM's IMS term for an entry in the "**redo-log**" which contains the value of an object after its modification.
- BEFORE-IMAGE:** IBM's IMS term for an entry in the "**undo-log**" which contains the value an object had before its modification.
- Journal:** a log of input or output operations which have been performed by a task. The journal is used to replace input and output which already took place during the ROLL-AHEAD operation. Journalling is only considered in database recovery when one does not wish to cancel a transaction which was not committed at crash time.
- Rollback:** restoring in a spare unit the state which prevailed in the failed unit at a previous point in time (recovery point) In a database, the recovery point can be a save-point (set-back), the beginning of a transaction (cancel), the last checkpoint (crash) or the latest committed transaction end (media failure).
- Roll-ahead:** continuing the computation once the recovery point has been restored until the next recovery point is reached. The roll-ahead differs from the normal computation because it must take into account operations performed on the environment by the failed processor.
- Backward Error:** a recovery method which consists of continuing computations
- Recovery:** on a spare or repaired unit, by first restoring a state which existed previously on the failed unit (rollback) and then restarting computations from that point on (roll-ahead).
- Save point:** a point in execution at which a copy of a part or the whole of volatile storage (part which is expected to fail) is built in a fail-independent storage (called transaction save-point by Gray, checkpoint by Tandem). This term sometimes designates the state saved instead of the saving operation.
- Recovery point:** a point in the execution at which a previous state may be reconstructed, either by reload and redo or by undo, and from which roll-ahead is started. This term sometimes designates the restored state instead of the saving operation. **Fire-Wall:** a point in the execution beyond which no work should be redone (called checkpoint by Gray and Haerder). In theory, no state need be saved at a fire-wall if no operations are in progress.
- Back-up copy:** a redundant copy of the volatile state taken at a checkpoint. Sometimes designates the redundant parts of a machine (back-up hardware) although "stand-by" would be more precise.
- Full Back-up:** a redundant copy of the totality of the machine state. If reloaded into the physical machine, it would restart exactly at the state it had at the moment of saving. Full back-up is a complete copy of a database.

7.10 References

- [BERNSTEIN 83] P.A. Bernstein, N. Goodman & V. Hadzilacos,
"Recovery Algorithms for Database Systems",
IFIP INFORMATION PROCESSING 83, R.E.A. PARIS September 1983.
- [BERNSTEIN 81] P.A. Bernstein & N. Goodman,
"Concurrency Control in Distributed Data Bases",
Computing Surveys, Vol. 13, No. 2, pp. 185-221 June 1981.
- [GRAY 78] J. Gray,
"Notes on Data Base Operating System",
Operating System Lecture Notes on Computer Science Vol. 60 Springer Verlag, 1978.
- [GRAY 81] J. Gray, P. McJones et al.,
"The Recovery Manager of the System R Database Manager",
Computing Surveys, Vol. 13, No. 2, pp. 223-242 June 1981.
- [HAERDER 83] T. Haerder & A. Reuter,
"Principle of Transaction-Oriented Database Recovery",
ACM Computing Surveys, Vol. 15, No. 4, pp. 287-318 December 1983.
- [Lampson 81] B. Lampson & H. Sturgis,
"Atomic Transactions",
Lecture Notes on Computer Science No. Springer, 1981 previously published by Xerox.

8 Standardisation and Conformance Testing

Die europäische und internationale Standardisierung fehlertoleranter Steuerungen hat in den letzten Jahren viel Fortschritt gemacht, sowohl durch allgemeine Richtlinien (z.B. IEC 61508) wie durch gebietsspezifische Normen (z.B. EN 50128/9 für die Eisenbahntechnik). Deutschland war dabei wegweisend, mit Arbeiten des TÜVs, des Technical Committee TC7 der EWICS und der Eisenbahnsignalisierungs-Industrie, insbesondere für das Europäische Signalisierungssystem ETCS.

Im Anhang steht eine Liste von Steuerungen, die durch den TÜV geprüft worden sind. Der Verweis eines Herstellers auf eine erfolgreiche Typenprüfung durch den TÜV oder eine andere Instanz nach einer allgemeinen Richtlinie wie IEC 61508 soll nicht überbewertet werden. Die Typenprüfung untersucht lediglich den Prüfling auf strukturelle Fehler hin, die seine Verwendung in Gebieten verwehren würde, die eine bestimmte Integrität oder Stetigkeit verlangen. Die Typenprüfung sagt aber nichts darüber aus, wie wahrscheinlich ein Integritätsbruch oder ein Stetigkeitsbruch ist.

Allein schon die Tatsache, dass die Anwendungssoftware nicht mitgeprüft wird, welche für mehr als die Hälfte der Ausfälle verantwortlich ist, zeigt, dass eine anwendungsspezifische Prüfung nachträglich unerlässlich ist. Es gibt zwar einige Richtlinien für die Softwaresicherheit; diese sind jedoch so allgemein formuliert, dass darüber keine Konformität geprüft werden kann.

Die wichtigsten Sicherheitsnormen in Europa

IEC/EN 62061	Safety of machinery - functional safety –Electrical, electronic and programmable electronic control systems
IEC/EN 61511	Functional safety of E/E/PES safety related systems – Functional safety: safety instrumented systems for the process industry sector
IEC 61508 (VDE 0801)	Functional safety of E/E/PES safety related systems – International standard (allgemeine Richtlinien)
IEC 60300	Dependability management
ISO/IEC 13849 (EN 954)	Safety of machinery – Safety-related parts of control systems
EN 50159	Requirements for safety-related communication in closed/open transmission systems
EN 50129	Railways applications - Safety-related electronic systems for signalling
EN 50128	Railways applications - Software for railway control and protection systems
EN 50126 (VDE 0115)	Railways applications - Specification and demonstration of reliability, availability, maintainability and safety (RAMS) – allgemeine Richtlinien
(VDE 0116)	Elektrische Ausrüstung von Feuerungsanlagen
DIN V 19250	Grundlegende Sicherheitsbetrachtungen für MSR-Schutzeinrichtungen (wird durch EN 61508 abgelöst)
IEC 880	Software for computers in the safety systems of nuclear power stations

Anforderungsklassen und Sicherheitsstufen

IEC 61508	TÜV Anforderungsklassen	Sporadischer Betrieb Wahrscheinlichkeit einer Funktionsweigerung	Kontinuierlicher Betrieb Wahrscheinlichkeit eines gefährlichen Fehlers	Gefahr/Risiko Stufe
SIL 1	AK 2 & AK3	10^{-1} bis 10^{-2} /h	10^{-6} bis 10^{-5} /h	kleinere Schaden an Anlagen und Eigentum
SIL 2	AK 4	10^{-2} bis 10^{-3} /h	10^{-7} bis 10^{-6} /h	grössere Schaden an Anlagen und Eigentum. Mögliche Personenverletzung.
SIL 3	AK 5 & 6	10^{-3} bis 10^{-4} /h	10^{-8} bis 10^{-7} /h	Personenschutz
SIL 4	AK 7	10^{-4} bis 10^{-5} /h	10^{-9} bis 10^{-8} /h	Mögliche katastrophale Folgen

9 Introduction to Dependability theory and models

This chapter consists of two parts: first the notions and definitions used in the field of dependable computing are presented in a somewhat more formal way than in Chapter 1. Then, calculation methods are explained for the different cases of non-redundant and redundant, non-repairable and repairable systems. The simple cases will be treated by combinatorial probability. The repair cases will be treated with the help of the theory of Markov chains. This theory represents the subset of reliability theory that is required for understanding fault-tolerant computing systems, but the calculations are the same for any other dependable system.

Based on these notions, we will consider in this chapter six categories of systems:

1. non-repairable, non-redundant (reliable)
2. non-repairable, redundant (reliable)
3. repairable, non-redundant (available)
4. repairable, redundant (available)
5. repairable, redundant and reliable
6. gracefully degradable

Finally, a table with the principal results will be displayed and some important references will be indicated.

9.1 Reliability of a single element

9.1.1 Reliability Definition

Let us consider a single element which is likely to fail, but cannot be repaired. From the point of view of dependability, the element has two states: "good" and "failed". This is symbolized in Figure 9-1:

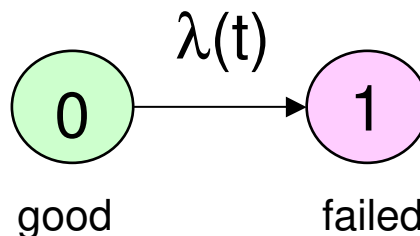


Figure 9-1: Single Element without Repair.

The arrow from the "good" state to the "failed" state expresses that there is a probability that an element that was in the good state passes to the failed state. This probability is expressed by the variable λ , which is the **failure rate**. There is no arrow from "failed" to "good", since the element is not repairable.

Reliability is defined as "a characteristic of an item expressed by the probability that it will perform a required function under stated condition for a stated period of time".

One expresses the reliability $R(t)$ of an item by the probability that this item will be in the working state at time t , provided the item was in that state at an initial instant $t_0 = 0$ at which time its reliability was 1, and has not failed since.

The reliability $R(t)$ is therefore the probability that the element has not failed in the interval $[0, t]$ provided that $R(t_0) = 1$. This is the complement of the probability that the element has failed in the interval $[0, t]$, which is the **unreliability** $F(t) = 1 - R(t)$.

The reliability is always a declining function like Figure 9-2 shows. After an infinite time, the reliability will always reach the value of 0. The definition of the instant $t=0$ is a matter of convention: one can define it as the time the element was fabricated, tested, first put into operation or inspected for the last time.

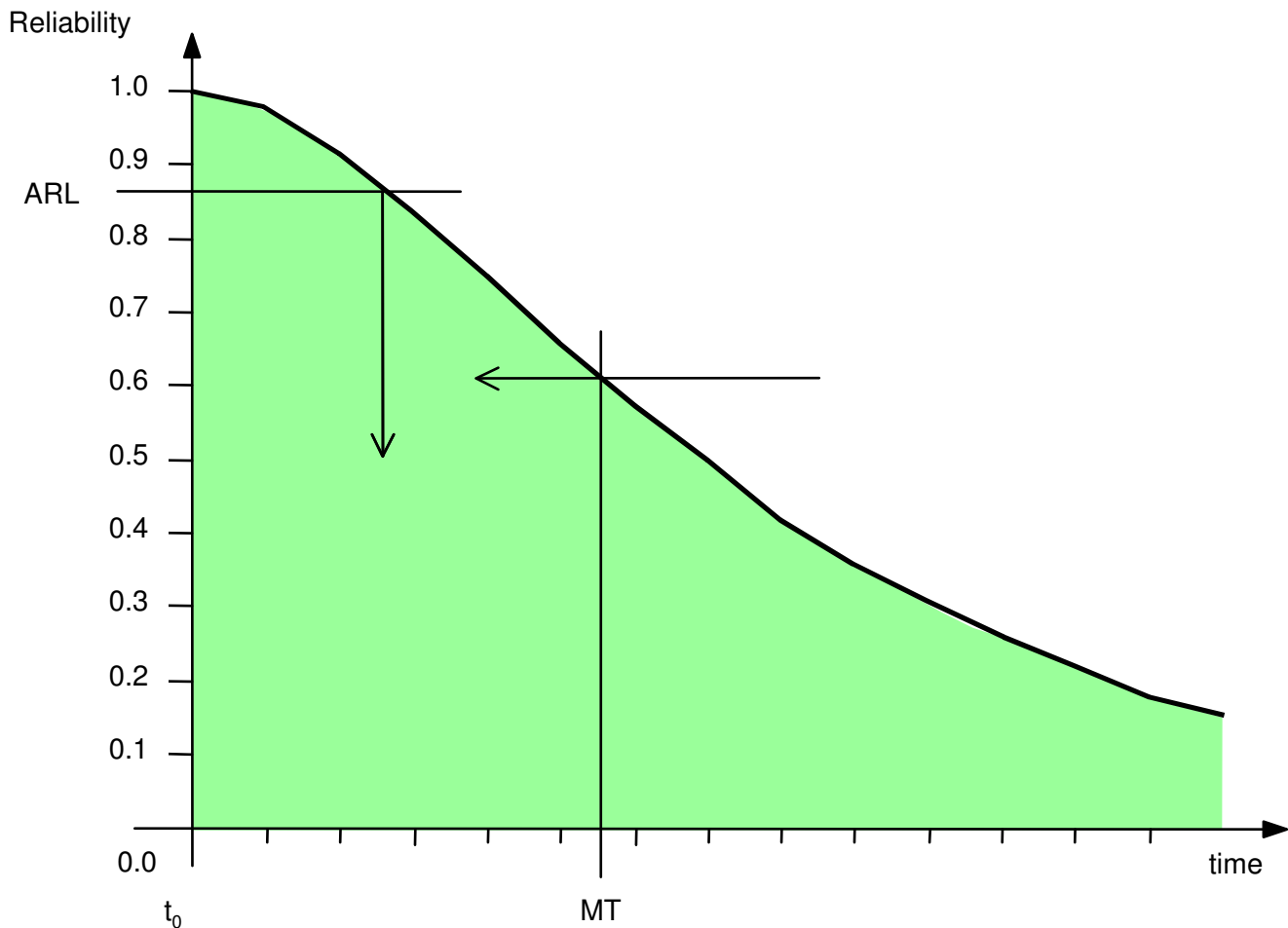


Figure 9-2: Acceptable Reliability Level and Mission Time.

Elements are seldom used up until their reliability tends to zero. Most of the time, only the left portion of the above curve is of practical importance. One is interested in the reliability of an element at the end of a certain **mission time**, which is the interval during which one depends on the good function of the element. The probability that the mission succeeds is therefore:

$$R = R(\text{mission time}).$$

Alternatively, one can define an Accepted Reliability Level ARL below which the mission is considered too risky and should be stopped. The mission time is therefore:

$$\text{mission time} = R^{-1}(\text{ARL})$$

To express the reliability of an element, one often uses the Mean Time To Fail, or MTTF of that element, which is the average lifetime of the element. It is generally expressed as:

$$\text{MTTF} = \int_0^{\infty} R(t) dt$$

In the above Figure, the MTTF corresponds to the surface below the plot. The term MTBF (mean time between failures) is sometimes used instead of MTTF in the case of repairable elements - we will come back to it.

9.1.2 Determining Reliability

The reliability of one particular, non-repairable element cannot be determined experimentally before the element fails, in which case the Figure becomes quite useless for that element. The problem is the same as to determine whether a match can be lighted: one cannot say before trying to light it, but after that, the only indication one gains is that the match **WAS** good, not how reliable the match **IS**. To determine the reliability, two ways are chosen: experimental reliability or previsional reliability:

- **Experimental reliability** (or empirical reliability) is determined by studying the behaviour of a great number of similar parts and gaining clues on the reliability of an element. For instance, the reliability of a light bulb can be deduced from studying large samples, as we will see.
- **Previsional reliability** calculates the reliability of a complex system by dividing it into parts, whose reliability has been previously determined experimentally, and by analysing their interrelationship. This will be discussed later when considering a collection of elements.

9.1.3 Experimental Reliability

Experimental reliability is determined by gathering experimental data of the reliability of a large quantity of similar elements, and deducing the reliability of a typical element out of it.

Example:

we study a large quantity, 6500 light bulbs. At a fixed interval Δt – for instance of one hour, we check how many good bulbs remain and plot this number in a curve like Figure 9-3 shows:

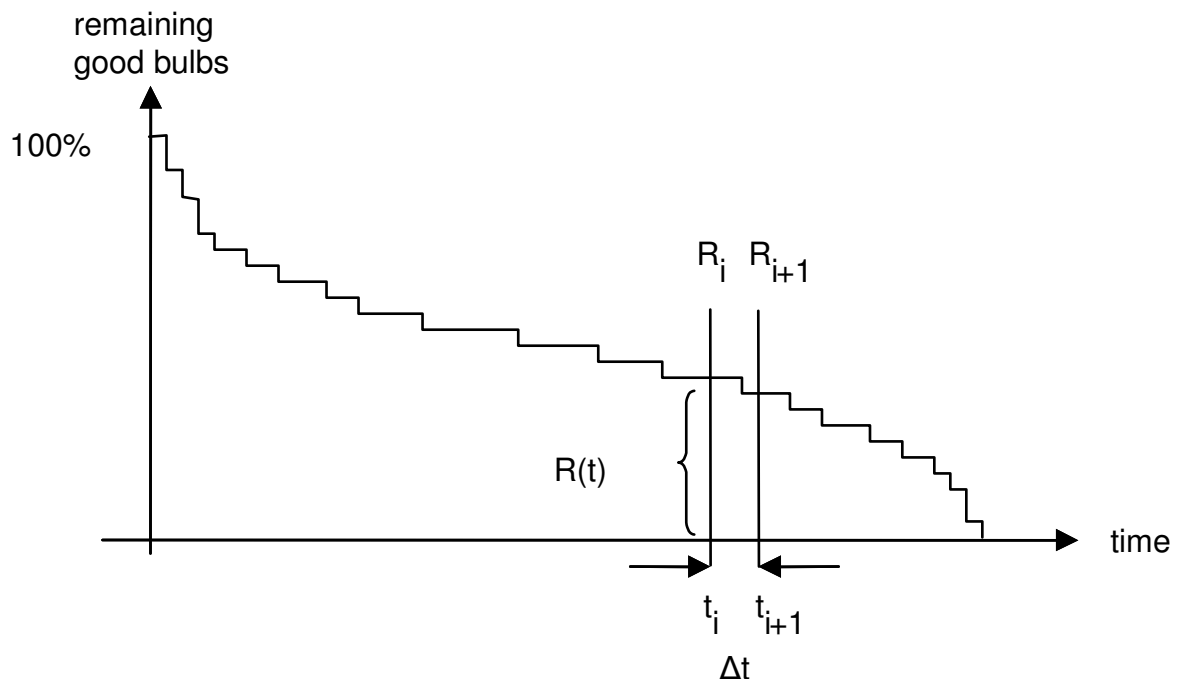


Figure 9-3: Experimental Reliability

One then draws a conclusion from this curve to the reliability of a single element, by observing the **duality rule** that the proportion of failed elements in a large sample is equal to the probability of failure of one element of the sample.

The curve in Figure 9-3 expresses therefore the reliability of a single bulb as well as the proportion of remaining good bulbs in a large sample.

An important Figure is the number of bulbs that fail in each time interval. It is clear that this number depends on how many bulbs are considered, since the more bulbs there are, the more failed bulbs there will be. So, instead of plotting each hour the number of remaining bulbs, we now plot the proportion of bulbs that have failed since last hour, and we call this relation λ :

$$\lambda = \frac{\text{number of failed bulbs during the interval}}{\text{number of good bulbs remaining at the beginning of the interval}}$$

The result is the curve of the **failure rate**, like is shown in Figure 9-4:

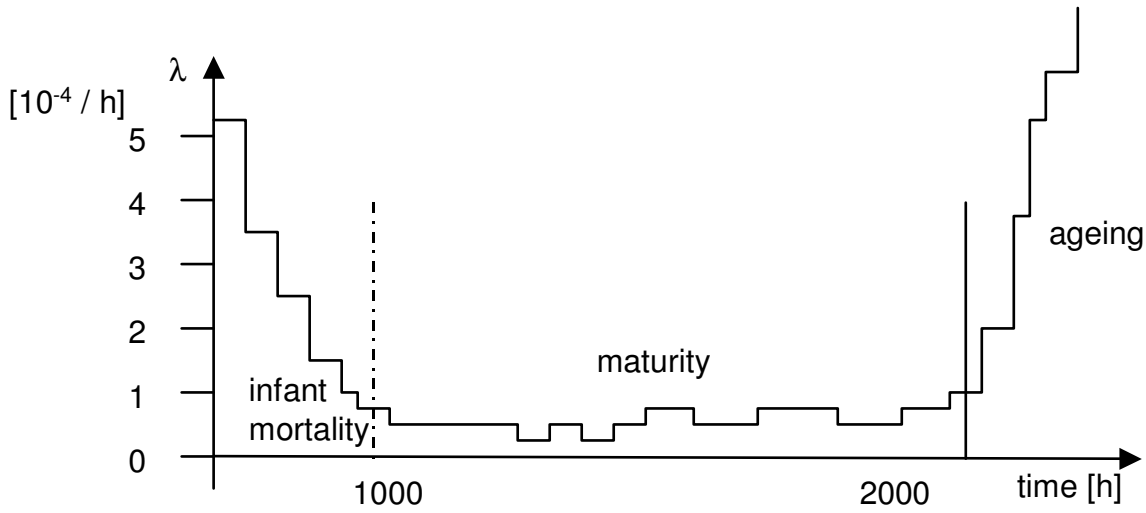


Figure 9-4: Experimental Failure Rate.

The above failure rate curve applies not only to many technical objects like light bulbs, but also to living organisms. It is commonly known as the "bathtub" curve: at the beginning of the life span, there is a large number of failures corresponding to the "infant mortality", which is generally related to manufacturing problems. The bottom part of the bathtub corresponds to the "maturity" phase, and the failure rate is about constant. At the end of the lifespan, the failure rate increases again because of ageing and wear. The failure rate of human beings is of course of special interest for the life insurance companies in calculating their rates.

The failure rate, at each instant t , defines the proportion of bulbs that are likely to fail in the next small interval Δt . If R_i is the number of remaining bulbs when reaching interval i , this proportion is equal to:

$$\frac{R_{i+1} - R_i}{R_i} = -\lambda_i \cdot \Delta t$$

We can now express the relationship between reliability (Figure 9-3) and failure rate (Figure 9-4) by the relation:

$$\frac{R_{i+1} - R_i}{t_{i+1} - t_i} = \frac{\Delta R}{\Delta t} = -\lambda_i \cdot R_i$$

Since this number is proportional to the probability of a single element being good, we can let the time interval $\Delta t = t_{i+1} - t_i$ tend to zero and express:

$$\frac{dR(t)}{dt} = -\lambda(t) \cdot R(t)$$

Although the reason for the failing of an element is quite complex [Spectrum 81], the experimental curves obtained over a great number of elements seem to follow simple mathematical laws. Therefore, we would like to obtain an analytical expression to characterize the reliability and the failure rate instead of curves as in Figures 9-3 and 9-4. Several formulas have been proposed which match different kinds of failures, but we will only consider here the most important one, which is the assumption of the **constant failure rate**, also called the assumption of the **exponential distribution** of failures.

The assumption of the constant failure rate is valid for elements in their mature phase (bottom of the bathtub). It applies to elements which do not age, and for which early failures are detected and screened out. Such is a good approximation for semiconductors that have passed the infant mortality stage. This last assumption can be enforced by a mature manufacturing process, by a proper test and by a temperature cycling procedure of about one week after manufacturing, called burn-in.

The process of determining the failure rate from a certain number of samples is a matter of statistical methods, which are described in several books, such as [Birolini 85]. We will not consider this experimental gathering of data, but assume that for each element, we know the failure rate. There exist several reliability catalogues for electronic and mechanical components, the most common being for the US market the military reliability handbook, MIL-HDBK-217D. A companion document, MIL-STD-781, describes the procedures which apply to the acceptance testing of the equipment, under the assumption that the equipment obeys a constant failure rate.

The following table shows typical reliability values that can serve as a base for the evaluation of computers. The reliability is expressed by the failure rate. The MTTF is only relevant for a whole system, since it is usually so low for a component that it has no practical meaning.

The failure rate is expressed by failures in 1000 million hours (10^{+9} h^{-1}) or "**fit**" (**failure-in-time**)³

One fit corresponds roughly to one failure in 114'000 years. Using fit avoids handling numerous zeroes after the decimal point. The following Table xx shows typical failure rates of electronic components:

discrete resistor (1/4 W)	1 fit		
discrete semiconductor diode:	2 fit		
discrete silicon transistor (<1W)	3 fit		
discrete capacitor (< 10 uF)	10 fit		
TTL-LS SSI	150 fit		
HC-MOS SSI	250 fit		
TTL-LS MSI	350 fit		
integrated octal bus driver	500 fit		
LED	360 fit		
256 K DRAM	400 fit		
microprocessor 8085	1000 fit		
lamp	2000 fit		
disk controller	2500 fit		
microprocessor 8086	5000 fit		
microprocessor Pentium 1 GHz	100'000 fit	(MTTF =	10000 hours)
power grid:	114'000 fit	(MTTF =	8765 hours)
computer (including memory)	150'000 fit	(MTTF =	100'000 hours)
power supply	333'000 fit	(MTTF =	3000 hours)
dry cells (limited life)	1500'000 fit	(MTTF =	720 hours)
PC (industry computer)	10'000 fit		
gateway	4000 fit		
star coupler	1000 fit		

Table 9: typical failure rates of electronic components

The above measures vary widely depending on the batch, the technology, the environment (benign, industry, aircraft, helicopter) and depend especially on the temperature, which is after time the most important factor. A rule of thumb for electronic equipment is that reliability is inversely proportional to the power dissipated:

$$\lambda \sim 10^4 \text{ [fit per Watt]}$$

Note also the heavy dependence on the power supply side. Most outages of commercial computers are caused by power supply failures, well ahead of mechanical devices failures (disks, tapes) and memory failures. CPU failures are relatively rare. Indeed, in industrial process control, the reliability increases from the periphery (sensors) to the CPU, which is normally located in a benign and temperature controlled environment.

We recall the following about experimental reliability: The failure rate of an element can be deduced from empirical studies over a large number of similar elements, by applying statistical methods. For electronic equipment, one assumes generally a constant failure rate over the useful lifetime of each element. This assumption allows such a simplification of the mathematical treatment of reliability that it is also made when this assumption is questionable, for instance for electronic devices which do age, such as relays and vacuum tubes.

9.1.4 Exponential Distribution

It is worth spending a few words on the assumption of the constant failure rate or exponential distribution. The constant failure rate corresponds to the mature phase of the element (bottom of the bathtub) in Figure 9-4. Assuming a constant failure rate considerably eases the calculations. This is why one assumes constant failure rates even where this is questionable.

³ Originally, "fit" meant something different.

Solving the above differential equation with a constant failure rate ($\lambda(t) = \lambda = \text{constant}$) yields:

$$R(t) = e^{-\lambda t}$$

This curve is plotted on Figure 9-5. It matches approximately the middle part of Figure 9-4.

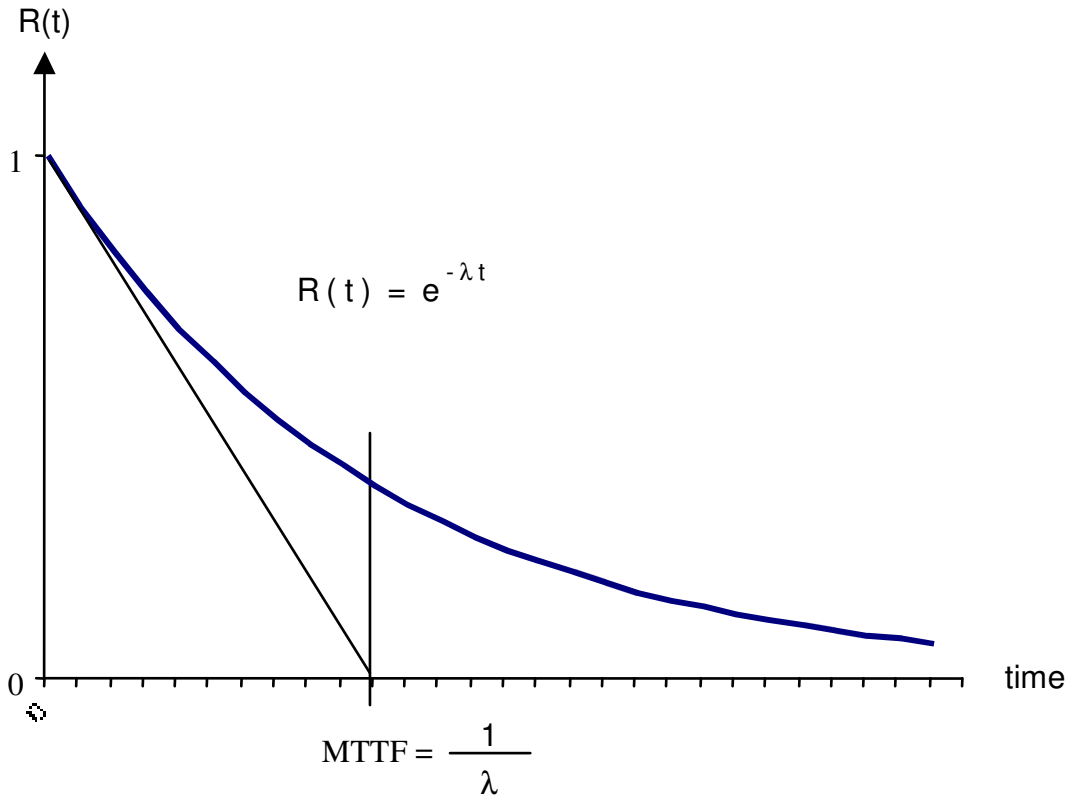


Figure 9-5: Reliability under the Assumption of a Constant Failure Rate (Exponential Distribution of Failures).

The Mean Time To Fail, or MTTF of that element is found by integration as:

$$MTTF = \int_0^{\infty} R(t) dt = \frac{1}{\lambda}$$

for a constant failure rate only.

The assumption of a constant failure rate seems quite simple, but we must bring it into accordance with our intuition. A constant failure rate means that the proportion of elements that will fail in the next small time interval dt is constant and independent on how long the experiment is under way.

For instance, if $\lambda = 0.001$ per hour, it says that in the next hour, 0.001 % of the remaining elements will fail, independently of how long these elements have been in service and how many elements have failed until now.

This means that every element that has not been observed to fail is considered "as good as new", just by noticing that it is still good. Our intuition would tell us that an element that has been in service a long time has a higher probability of failure than a new element. On the other hand, our intuition also tells us that an element that has been in service for a long time can be more trusted than a newly inserted element. In reality, these heuristic rules apply respectively to the end and to the beginning of the lifetime of an element which ages, and not to its mature phase. We consider a kind of exponential distribution when we choose flashlights off a shelf for a hike: some flashlights are exhausted and discarded, and we consider each of the remaining good flashlights as new, regardless of how many of the other flashlights we tried may have failed.

To make it easier to understand, we consider another distribution. The exponential distribution has a counterpart in a **discrete** distribution called the **geometrical** distribution. The geometrical distribution describes a roulette or other kind of hazard device which has a defined probability p that an event occurs at each throw. For instance, a dice has

a probability of $1/6$ of throwing a 2, the probability of picking an Ace of Hearts is $1/52$ in a deck of 52 cards. The geometrical distribution expresses the probability p that the event occurs at throw number i and has not occurred before. The form of this distribution is shown in Figure 9-6.

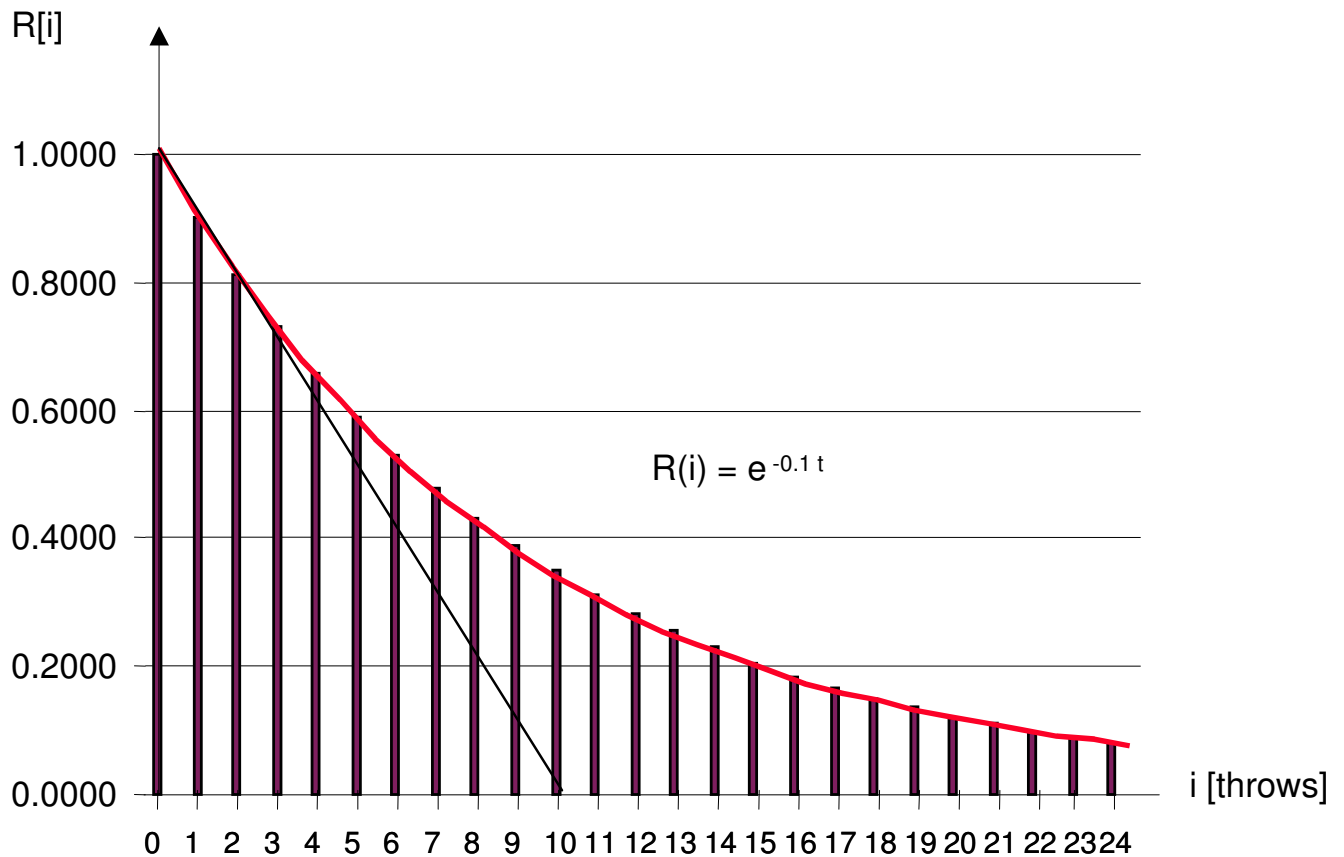


Figure 9-6: Geometrical Distribution
(Probability of not throwing a 6 with a 6-faced dice within a number i of throws)

Note the similarity with Figure 9-5. We can assume that each throw corresponds to a small time interval dt . The probability of failure at each throw is always the same and equal to p (constant failure rate). As time passes, the probability of failure diminishes because the element may have failed before. So, we may think that the life of an element with a constant failure rate is similar to a hazard game in which time is sliced and a dice is thrown at each small time interval, deciding whether the element dies or not. Or, like the German soldier song says:

"Das Leben ist ein Würfelspiel" - Life is a dice game.

Other distributions than the exponential distribution are sometimes considered. Some authors use the Poisson distribution. The Poisson distribution applies to a large number of identical devices, and expresses the probability that 0, 1, 2 or k devices would fail during time t . The case $k = 0$ is equal to the exponential distribution, and this is the only relevant case for us.

The assumption of a constant failure rate is questionable for instance for mechanical elements which age, and it does not describe the infant mortality processes. Here, more complex distributions like that of Weibull or Normal are taken. But we leave this to more appropriate textbooks, like [Shooman 68]. The exponential distribution serves our purposes well.

9.2 Reliability forecast by combinatorial means

When an element is unique or exists only in a small quantity, no experimental data exist for it. The reliability must be calculated on the base of the reliability of the different parts and their interaction.

The theory of reliability states that when a system depends on two elements to function, and the elements fail independently, then the reliability of the system is the product of the reliability of the parts. This can be extended to N elements. The reliability block diagrams express this, as Figure 9-7 shows.

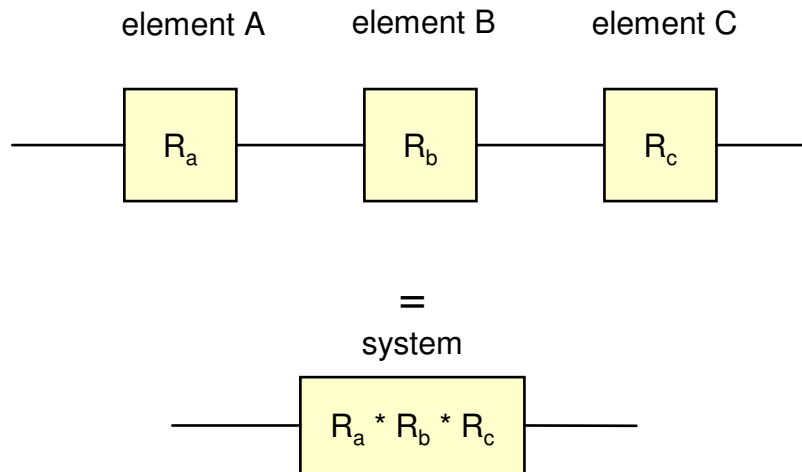


Figure 9-7: Reliability Block Diagrams.

When a system depends on N elements for its function, and there is no redundancy, the total reliability is the product of the reliability of the different elements:

$$R_{\text{total}} = R_1 \cdot R_2 \cdots R_n = \prod_{i=1}^n R_i$$

This expression only holds if the N elements fail independently, that is, when there is no relation between them. The consideration of the interdependency only brings additional complexity in the calculation, since the above expression gives the worst case.

Under the assumption that $R(t)$ is an exponential function, one just can sum the λ of the different factors:

$$R(t) = e^{-\lambda_1 t} \cdot e^{-\lambda_2 t} \cdots e^{-\lambda_n t} = e^{-(\lambda_1 + \lambda_2 + \cdots + \lambda_n) t}$$

For instance, to evaluate the reliability of a computer board, the elements are just listed by categories. The λ factor of each element is taken from the manufacturer's data sheets or from the handbooks, and the λ factors are simply summed. The total is the λ of the board. A finer analysis should of course consider the interaction between parts, but this is seldom required. The worst-case assumption that the board fails when any part of it fails is taken.

9.2.1 Non-repairable, redundant systems

Non-repairable, redundant systems are a class of fault-tolerant systems. Some of their parts may fail, but the service is still provided until the spares are exhausted. There is no repair process. This is a typical situation in applications where repair is impossible, such as for space probes or communication satellites. The paradigm of non-repairable, redundant systems is the matchbox: one can use it to light the fire as long as there is still one good match in it, the fact that one match is bad has no influence on the others.

Example

A space probe sent to Jupiter may have a mission time of 5 years, and, if the MTTF of the probe is also 5 years, the mission will only have a $1/e = 37\%$ probability of success. The statistic for communication satellites over the last years shows that the mission success is in excess of 95% (after successful launch), which implies that the MTTF of the satellites is well in excess of 5 years. This can be achieved through careful selection of the parts (fault avoidance) but also through redundancy (fault tolerance).

When the reliability of an element is too low to fulfil the mission, redundancy must be used. At this point, we do not detail how redundancy is obtained, either by parallel elements, switched elements or time redundancy. We do not consider either how the transition is done when a redundant component fails. We just assume that the system can detect the failure or mask it in some way. Redundancy is expressed in terms of reliability blocks by paralleling blocks, as in Figure 9-8:

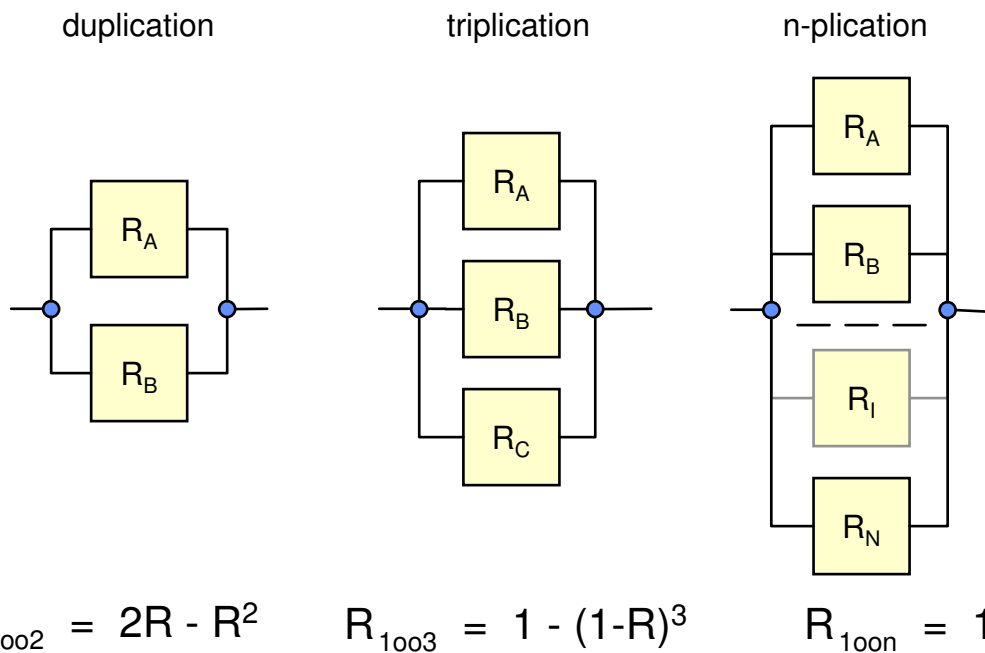


Figure 9-8: Redundancy.

The most common kinds of redundancy are **duplication** (1-out-of-2 or 1/2), **triplication** (1/3) in general, **n-plication**, as Figure 9-8 a,b and c shows.

The increase in reliability that results from redundancy will now be calculated for the most important cases.

9.2.2 Duplication

The duplicated system (also called one-out-of-two, or 1oo2, is expressed by the reliability diagram of Figure 9–9. The reliability of the whole is the probability that the two elements do not fail.

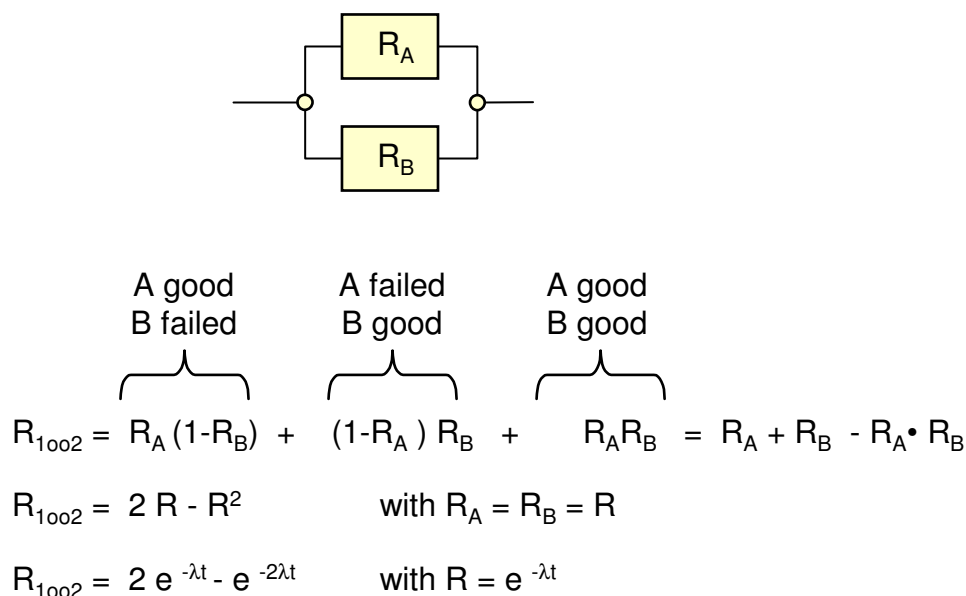


Figure 9-9: Reliability diagram and reliability of a Non-Repairable Dual System.

In principle, the increase in reliability provided by duplication is considerable:

Example:

If the reliability of one unit is .90, the redundant system will have a reliability of 0.99, i.e. the unreliability has decreased by a factor of 10.

However, if the elements are not reliable, the reliability of the dual system only increases slightly. if the reliability of an element is 0.5, the reliability of the redundant system will only be 0.75.

Assuming a constant failure rate for each device, λ_A and λ_B , and $\lambda_A = \lambda_B = \lambda$, the resulting reliability is:

$$R_{1002}(t) = 2e^{-\lambda t} - e^{-2\lambda t}$$

We now plot this curve along with the reliability of a single element (Figure 9-10):

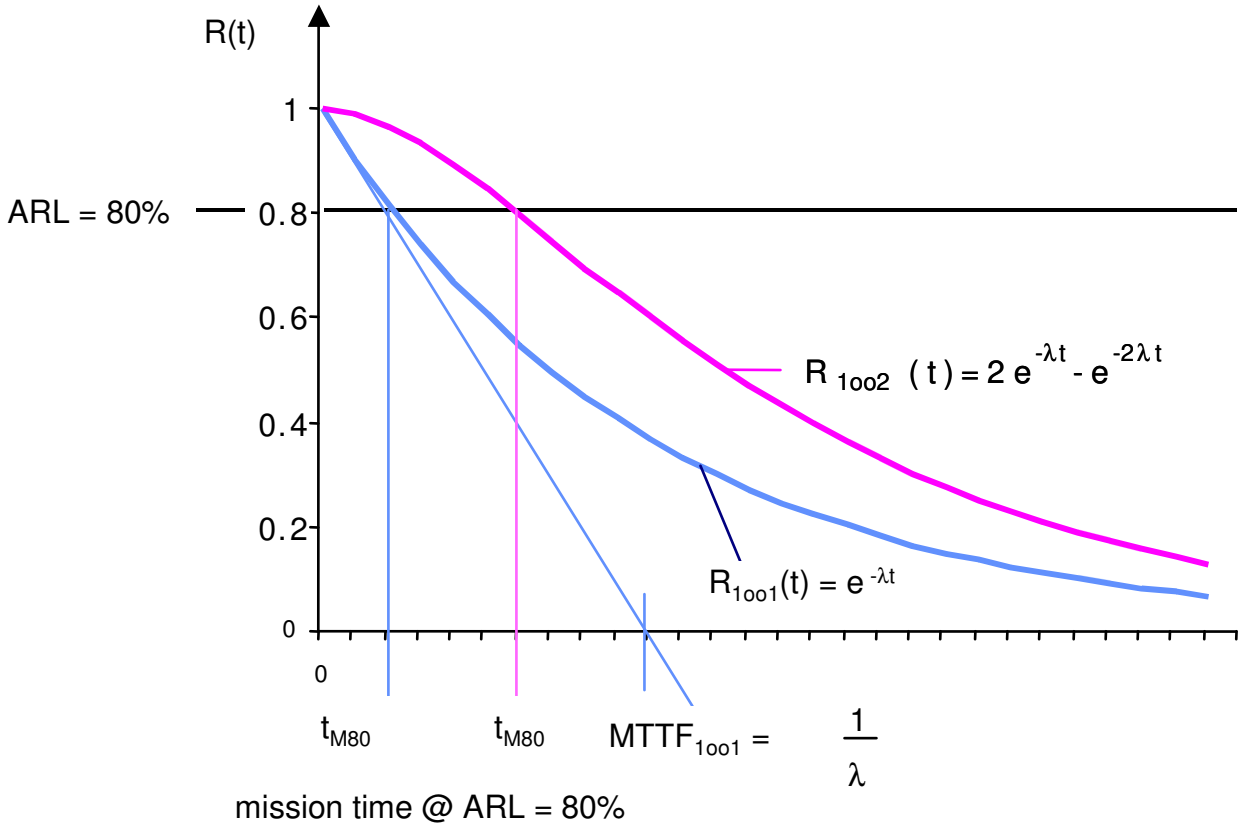


Figure 9-10: Reliability of Duplicated versus Single System.

Duplication increases the reliability especially near the origin, i.e. for short mission times.

The MTTF of the duplicated system is calculated as:

$$MTTF_{1002} = \int_0^{\infty} R(t) dt = \frac{2}{\lambda} - \frac{0.5}{\lambda} = 1.5 \cdot \frac{1}{\lambda} = 1.5 \cdot MTTF_{1001}$$

We come to a rather strange result: The MTTF of the duplicated system is only one and a half times that of the non-replicated one. This does not look encouraging: in fact, the MTTF of a part is seldom known with precision. Fluctuations of 50% on the value are common, depending on the manufacturer and the batch. This means that we cannot expect a significant increase in the lifetime of a system by duplication. This is quite easy to understand: if the redundant parts have the same failure rate, it is likely that both will fail within a given, long time.

Example:

This is why duplicating the internal circuits does not protect integrated circuits against solid faults. Unless one can detect and replace the failed circuit half immediately, the increase in reliability is negligible. However it may help against transient faults that leave the hardware intact.

Therefore, the utility of duplication without repair lies in its use on a small time span near the origin: there, the failure of a part is unlikely, and it is even more unlikely that the redundant part will fail also.

We see here that the MTTF is not an appropriate measure for non-repairable, fault-tolerant systems: it does not express the quality on a small time scale. So we should look for another measure.

When the mission time is undefined, one can then declare the mission terminated when a certain level of reliability is reached. Then, the criterion is the MIF, or **mission (time) improvement factor**, which is the ratio of the mission times obtained by the different methods for a certain level of reliability.

Example: If the acceptable reliability level is $ARL = 0.99$, the mission time increases from $0.01 \lambda \cdot t$ to $0.105 \lambda \cdot t$. The MIF obtained by a duplicated system is 10.5, which is appreciable. If the acceptable reliability level is 0.9, the MIF drops to 3.6.

However, the MIF may not be of great interest, since normally the mission duration of a device is known in advance. One chooses then to express the improvement as the RIF or **reliability improvement factor**, which is as the ratio of the unreliabilities:

$$RIF = \frac{F_{\text{single}}}{F_{\text{duplicated}}} = \frac{(1 - R_{\text{single}})}{(1 - R_{\text{duplicated}})}$$

Example:

A twin-engine aircraft has to cross the Atlantic in a 10 hours flight.

The failure rate of each motor is $= 0.001 \text{ h}^{-1}$. (MTTF = 1000 hours).

The probability of failure is 1% with one motor, 0.01% with two motors; the RIF is 100.

However, if the same plane would fly 1000 hours without repairing a damaged motor, the probability of success would drop to 0.60, about the same as without redundancy (0.37), the RIF drops to 1.5, an uninteresting value.

9.2.3 N-plication

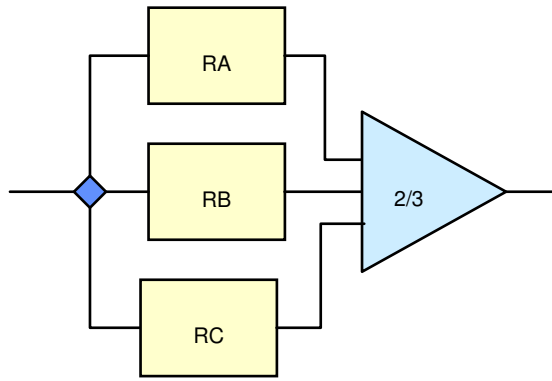
Triplication and in general N-plication are handled by the same methods as duplication. The simplest assumption is that the system requires at least one of N redundancies to function. The formula for that case is:

$$R_N = 1 - (1 - R)^N \approx N \cdot R \text{ for } R \approx 1$$

As for duplication, N-plication is most interesting when the reliability level is large anyway, i.e. near the origin.

9.2.4 Triple Modular Redundancy

Triple modular redundancy (TMR), or **2-out-of-3 (2oo3)** is often used in high reliability applications: Three identical units execute the same program at the same time. The results should be identical. A voter compares the results and takes that of the majority (Figure 9-11a).



$$R_{2003} = \underbrace{R_A \cdot R_B \cdot R_C}_{\text{none failed}} + \underbrace{[R_A \cdot R_B \cdot (1 - R_C)]}_{\text{C failed}} + \underbrace{[R_A \cdot (1 - R_B) \cdot R_C]}_{\text{B failed}} + \underbrace{[(1 - R_A) \cdot R_B \cdot R_C]}_{\text{A failed}}$$

$$R_{2003} = 3R^2 - 2R^3 \quad (R_A = R_B = R_C)$$

$$R_{2003} = 3e^{-2\lambda t} - 2e^{-3\lambda t} \quad (\lambda_A = \lambda_B = \lambda_C = \lambda)$$

Figure 9-11: Triplication and Voting (TMR) – block diagram and reliability

Assuming that each unit has a constant failure rate λ , the reliability of the whole is:

$$R_{2003} = 3e^{-2\lambda t} - 2e^{-3\lambda t}$$

The reliability plot of TMR is shown in Figure 9-12:

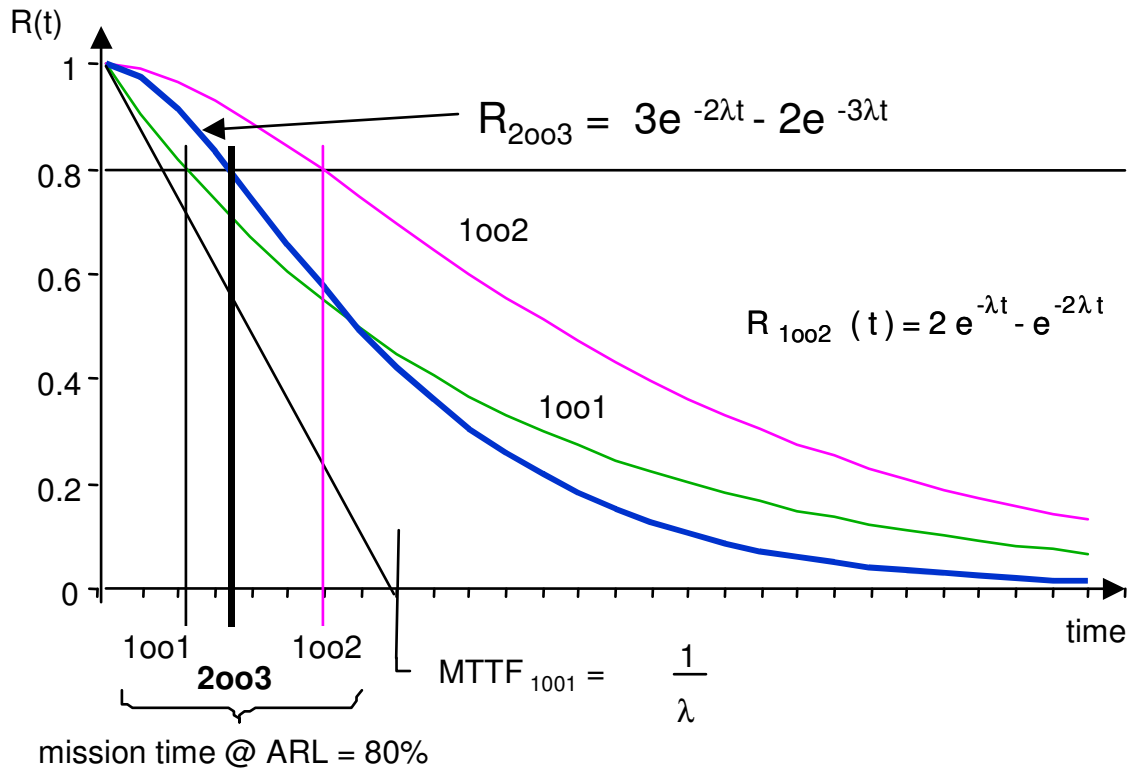


Figure 9-12: Reliability Plot of TMR

An interesting result of this plot is that the reliability of TMR drops **below** that of a simplex system when the mission time comes near to the MTTF of a single element. In fact, the total MTTF is equal to:

$$MTTF_{2003} = \frac{3}{2\lambda} - \frac{2}{3\lambda} = \frac{5}{6} \cdot \frac{1}{\lambda} = 0.8333 \cdot MTTF_{1001}$$

TMR has brought no improvement in the MTTF, on the contrary. This confirms that the MTTF is not relevant to describe long-living redundant systems. The important parameter is the Mission Time. Figure 9-12 shows the Mission Times that correspond to an acceptable reliability level of 90%, for the non-redundant and for the TMR case.

But even so, the MIF for a ARL = 0.99 is only 6 (it was 10 for a duplicated system). So, one can legitimately ask the question: what is a TMR system good for?

TMR has the advantage of combining redundancy and error detection. If we had to add the reliability of the components necessary for error detection and switchover in a duplicated system, the dual system would possibly be worse than TMR. In fact, we should consider the reliability of the voter in the above calculations also. The point is, that it is easier to design and build a reliable voter for TMR than an error detection and switchover logic for a dual system.


But the fact remains that static (non-repairable) TMR only makes sense when the mission duration is short with respect to the lifetime of the parts. When mission time becomes longer, redundancy does not offer much advantage and becomes even a source of unreliability - the larger the number of elements, the more likely the failure of one is.

Example: TMR has been used in the guidance computer of the Saturn V rocket. The mission time is below 30 minutes. There, redundancy offers a large improvement in reliability.

9.2.5 k-of-N systems

The case of TMR can be generalized to a k-of-N system, for which it is necessary that at least k units out of N redundant units be in working conditions to perform the required work. This situation occurs for instance in computers with multiple processors and memories [Siewiorek 78]. We will come back to its justification when considering graceful degradation. The reliability of such a k-of-N system is given by:

$$R_{koon} = \underbrace{R^n}_{\text{none failed}} + \underbrace{\binom{n}{1} + R^{n-1}(1-R)}_{\text{1 failed}} + \underbrace{\binom{n}{2} + R^{n-2}(1-R)^2 + \dots}_{\text{2 failed}} + \dots$$



 n - k terms

If we assume that all redundant units are identical and have a constant failure rate λ , we can plot the reliability of a system of four units as Figure 9-13 shows:

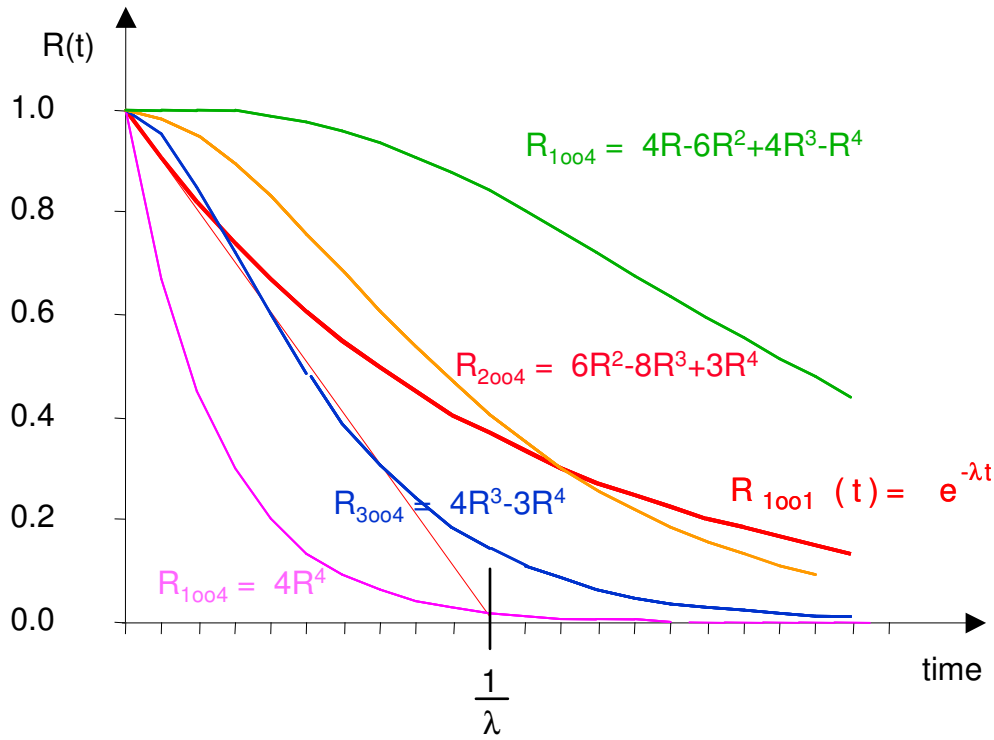


Figure 9-13: k-of-N Reliability.

Note that the reliability of a k-of-N system becomes in any case worse than the reliability of a single element after a long enough time.

A general conclusion can be drawn from this: the average life of redundant systems that depends on more than one redundancy to function is shorter than that of a non-redundant one. Non-repairable, redundant systems are only justified for short mission times.

9.2.6 Influence of Granularity

The question is how a system should be partitioned, and especially which is the optimum size of the replaceable units, assuming that the recovery takes place individually for each replaceable unit.

In principle, the smaller the replaceable unit, the higher the number of faults the system can tolerate before failure. If the system is partitioned into only two redundant RUs, it will fail for any fault in both units. If the system is partitioned into several RU, then it will only fail if both RU of the same function are faulty, as Figure 9-14 shows:

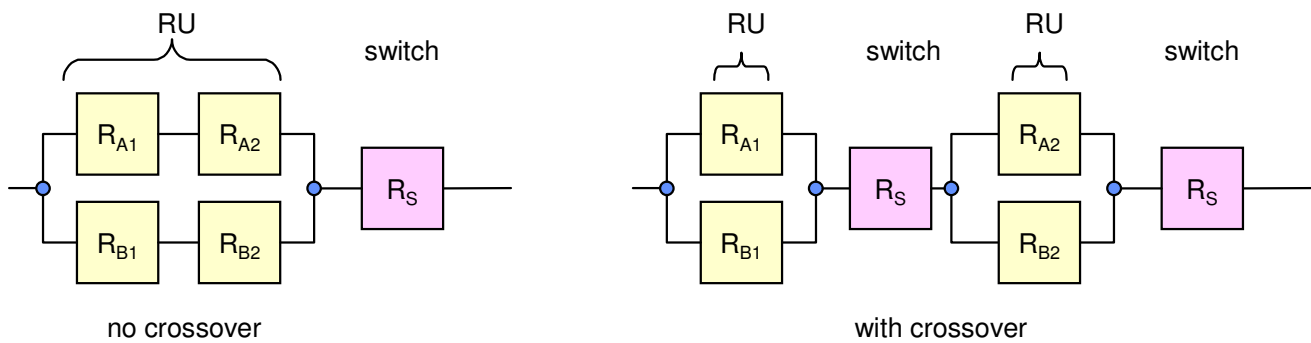


Figure 9-14: Large and Small Replaceable Units.

The reliability calculation will be shown assuming that the system consists of N units, each of which has a reliability R:

System consisting of one duplicated RU of N elements:

$$R_{1002} = 1 - (1 - R^n)^2$$

System consisting of N duplicated RUs:

$$R_{1002} = 1 - (1 - R^2)^n > 1 - (1 - R^n)^2$$

This result tends to show the advantage of small RUs. However, this assumes that the switching unit is a fully reliable element. Although the switching unit can itself be replicated, it introduces a new unreliability source. Further, we can include in the switching unit's reliability the common mode error probability due to the fact that the replicated units are closely coupled by the switch.

We shall consider that the reliability of the switching unit does not depend on the size of the RU. This is true anyway for logic comparators, voters and switches. In this case, the introduction of N switching units must be considered:

$$R_{\text{total}} = R_{\text{RU}} * R_s^n$$

The results are shown graphically in Figure 9-15

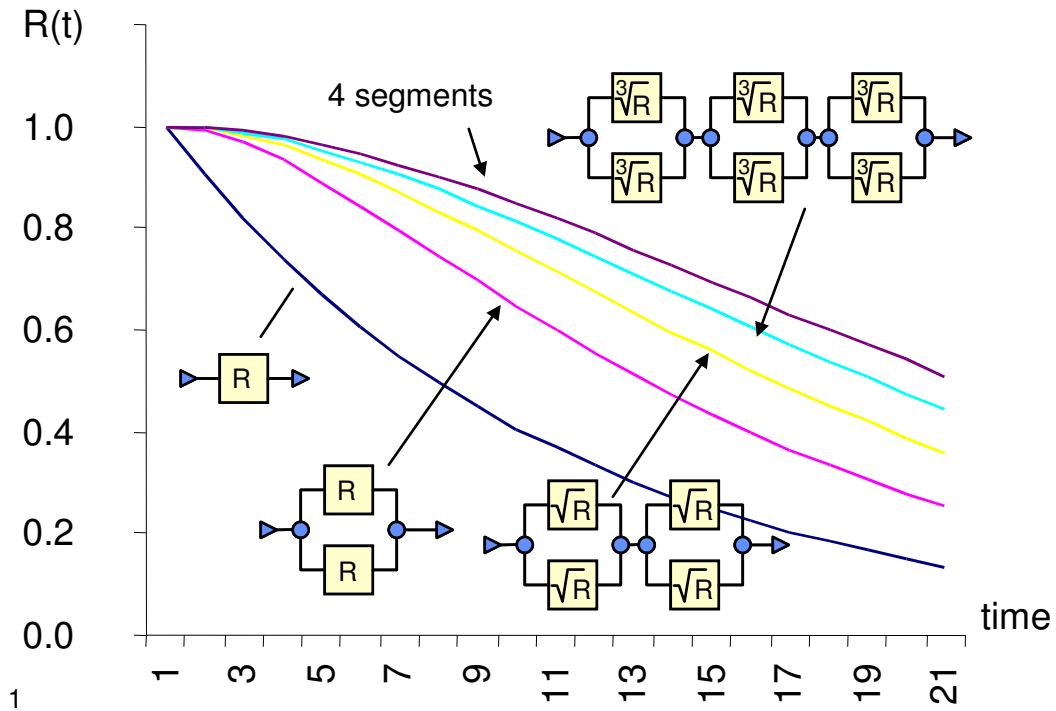


Figure 9-15: Influence of Granularity.

This shows that the reward in reliability obtained by reducing the size of granularity as getting each time smaller.

Furthermore, it has been shown that the system reaches its highest reliability when all RUs have the same failure rate. Thus, a balancing of the failure rates is a wise design decision.

Even so, these results assume that the switches are fully reliable and crossover is perfect. This is in practice difficult to achieve. In the end, it is only the reliability of the switches that matters.

9.3 Available systems

9.3.1 Definitions

Repair is the key to long mission times. Repairable, redundant systems are quite common. Most fault-tolerant computers for commercial applications belong to that category. The redundancy is there to increase the availability and also to stretch the maintenance interval.

When an element is **repairable**, the failure is not permanent. After a repair time, the element can be brought to operation again. Figure 9-16 shows the lifecycle of a repairable element:

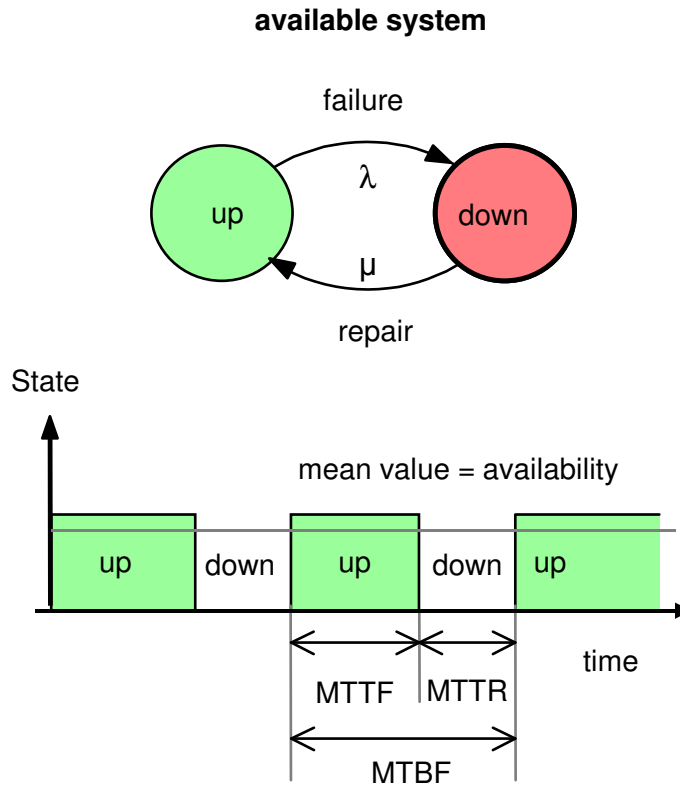


Figure 9-16: Lifecycle of a Repairable Element.

The element can be in one of two states, "work" (0 fault) or "failed" (1 fault). The terms "Up" for "good" and "Down" for "failed" are more appropriate when one wishes to emphasize that the system is repairable. A failure corresponds to the transition labelled λ . A repair corresponds to the transition labelled μ . The system oscillates between the states "Up" and "Down" forever.

We remember that this was not the case for the reliable, non-repairable, system of Figure 9-1. There, the system remained in the "Failed" state forever once it reached that state.

The **punctual availability** $A(t)$ is defined as the probability of the system being in the "Up" state at time t , provided that $A(t) = 1$ when $t = 0$. The availability is a function of time, but it reaches rapidly a stationary value called the **long-term availability** or short **"availability"**. The (long-term) availability is defined as:

$$A_{\infty} = A = \lim_{T \rightarrow \infty} \frac{1}{T} \int_0^T A(t) dt \approx \lim_{t \rightarrow \infty} A(t)$$

The average time during which the system is not operational ("Down" state) is called the Mean Down Time (MDT), as opposed to the Mean Up Time (MUT), which is the average time the element spends in the "Up" state.

The term MTBF, Mean Time Between Failure, is often used in the literature to express the average time between two successive transitions from "Up" to "Down". It is therefore equal to $MUT + MDT$. Since most systems spend much more time working than being repaired (hopefully), the MTBF is often equalled with the MUT.

The (long-term) availability can be expressed as the relation of operational time (MUT) to the total lifetime (MUT + MDT) of the system:

$$A = \frac{MUT}{MUT + MDT} = \frac{MUT}{MTBF} = \frac{1}{1 + \frac{MDT}{MUT}}$$

Like reliability, the numerical value of the availability is close to 1 for all practical purposes. Therefore, to avoid handling a large number of 9s after the decimal point, one prefers to express it as the unavailability, which is equal to:

$$\bar{A} = \frac{MDT}{MUT} = \frac{1}{A} - 1 \approx 1 - A$$

Example

to an availability of 0.995 corresponds an unavailability of 0.005, or 44 hours/year. It is found by multiplying the unavailability by 8765 hours (one year).

Some like to outline the time from mission start to the first failure as Mean Time To First Failure, MTTF.

The **maintainability** $M(t)$ of an element is the probability that this element is repaired within a time t after the failure occurred. It is commonly expressed by the MTTR, Mean Time To Repair, of that element. Although this is an approximation to reality, one usually assumes an exponential function for the time to repair. The repair rate is therefore a constant, μ , with:

$$\mu = 1 / MTTR$$

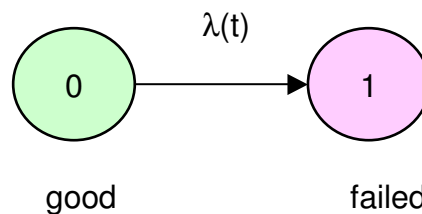
The reason is that the calculations become much too complicated with any other assumption. The MTTR differs only from the MDT if one considers the latency time, during which a failure exists, but is not detected.

9.3.2 Availability computation by Markov processes

We will introduce here the method to calculate the availability and the reliability of systems by considering the state-transition model. The bases of this theory are due to Markov, and the model we consider is termed a "continuous Markov model" of a system. There are other Markov models that we disregard. The calculation involves solving differential equations by using the Laplace transformation. It is assumed that the reader is familiar with the Laplace transform. A good introduction can be found in the Kleinrock's book on queuing theory [Kleinrock 75], a short form in [Siewiorek 82].

Remember the simple, non-repairable system with a constant failure rate.

The differential equation governing its behaviour was, with $R(t)$ as the probability of being in the "good" state P_0 . (Figure 9-16a)



$$\frac{dR(t)}{dt} = -\lambda \cdot R(t)$$

Figure 9-16a 1oo1 reliable system

More intuitively, we note that this differential equation is similar to that of a leaking water tank, since the flow rate is function of the pressure, i.e. the level of the tank. The rate defines the outflow (Figure 9-16b):

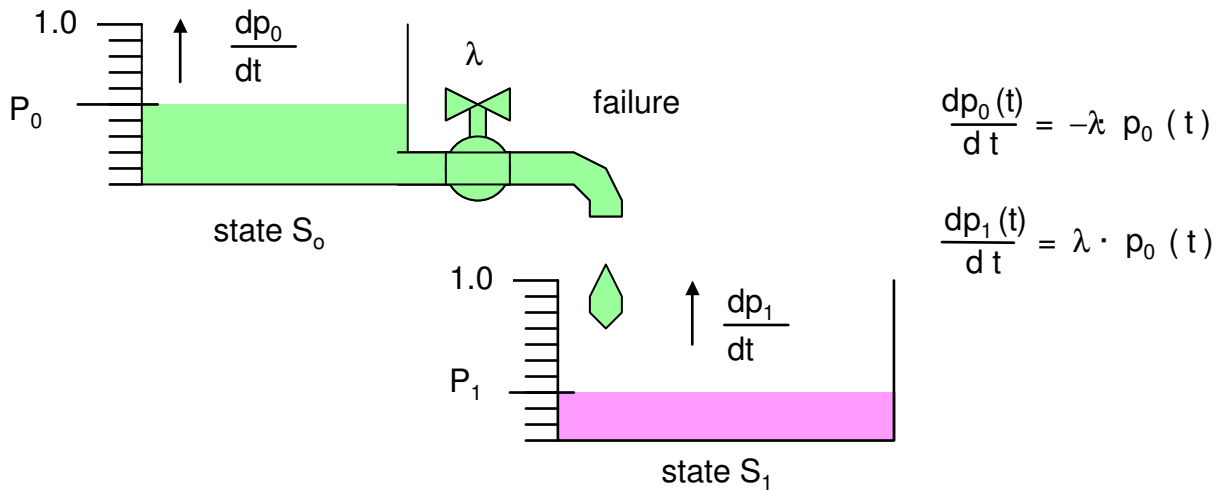


Figure 9-16b Reliability of 1oo1: water tank model

Solving this differential equation yields the know equation:

$$R_{1001}(t) = e^{-\lambda t}$$

For computing the availability, we extend this model by a repair rate, symbolized by a pump (cheating somewhat with the fluid –physics – the repair “flow” is supposed to be proportional to the tank level.

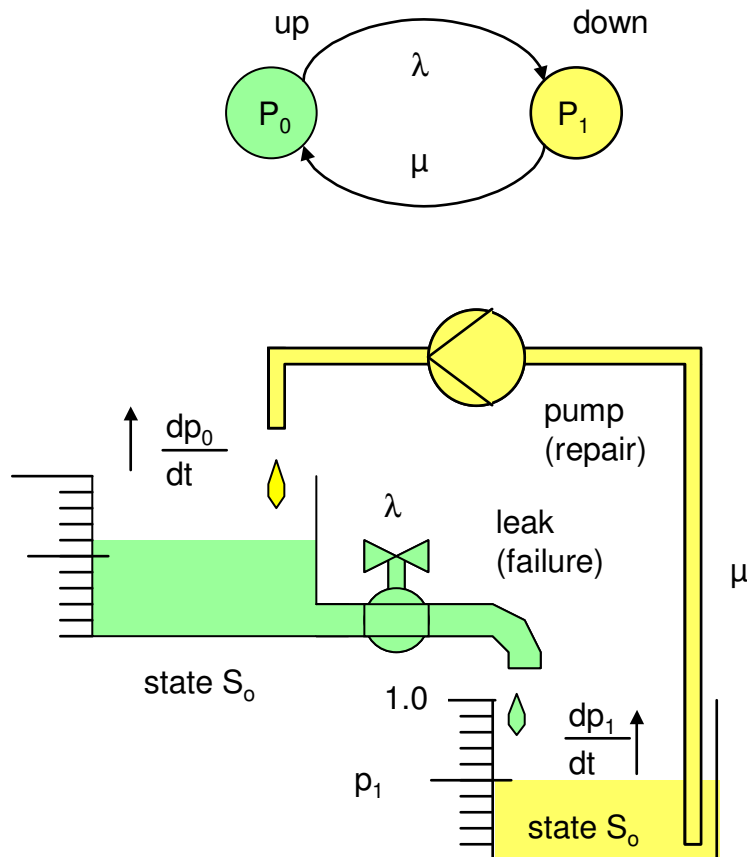
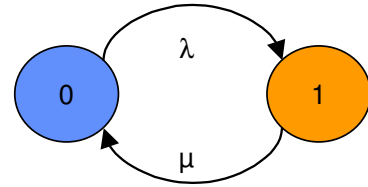


Figure 9-17: Fluid Analogy of an available system.

As an analogy, we can consider each state as a kind of fluid tank where the height of the fluid in a tank is proportional to the probability of being in that state. The transitions between states can be viewed as unidirectional pipes with a flow rate proportional to the failure rates multiplied by the quantity of fluid in the originating tank. The repairable system is described by the differential equations:

$$\begin{cases} \frac{dp_0}{dt} = -\lambda p_0 + \mu p_1 \\ \frac{dp_1}{dt} = +\lambda p_0 - \mu p_1 \end{cases}$$

initial conditions:
 $p_0(0) = 1$ (initially good)
 $p_1(0) = 0$



We can solve these equations by applying the Laplace transformation. With the initial conditions $P_0(0) = 1$ and $P_1(0) = 0$ (initially good) we solve the Laplace equations:

$$\begin{cases} s\tilde{P}_0 - 1 = -\lambda\tilde{P}_0 + \mu\tilde{P}_1 \\ s\tilde{P}_1 = +\lambda\tilde{P}_0 - \mu\tilde{P}_1 \end{cases}$$

and obtain:

$$\begin{cases} \tilde{P}_1 = \frac{\lambda}{s + \mu} \tilde{P}_0 \\ \tilde{P}_0 = \frac{1}{s} \cdot \frac{s + \mu}{s + \mu + \lambda} = \frac{\mu}{s(\mu + \lambda)} + \frac{\lambda}{s + \mu + \lambda} \end{cases}$$

We can transform these expressions back to the time domain by applying the method of the partial fraction expansion and performing the inverse transform, so we obtain:

$$p_0(t) = \frac{\mu}{\mu + \lambda} + \frac{\lambda}{\mu + \lambda} e^{-(\mu + \lambda)t}$$

Figure 9-18 plots the availability $A(t)$ = probability of being in the S_0 state as a function of time:

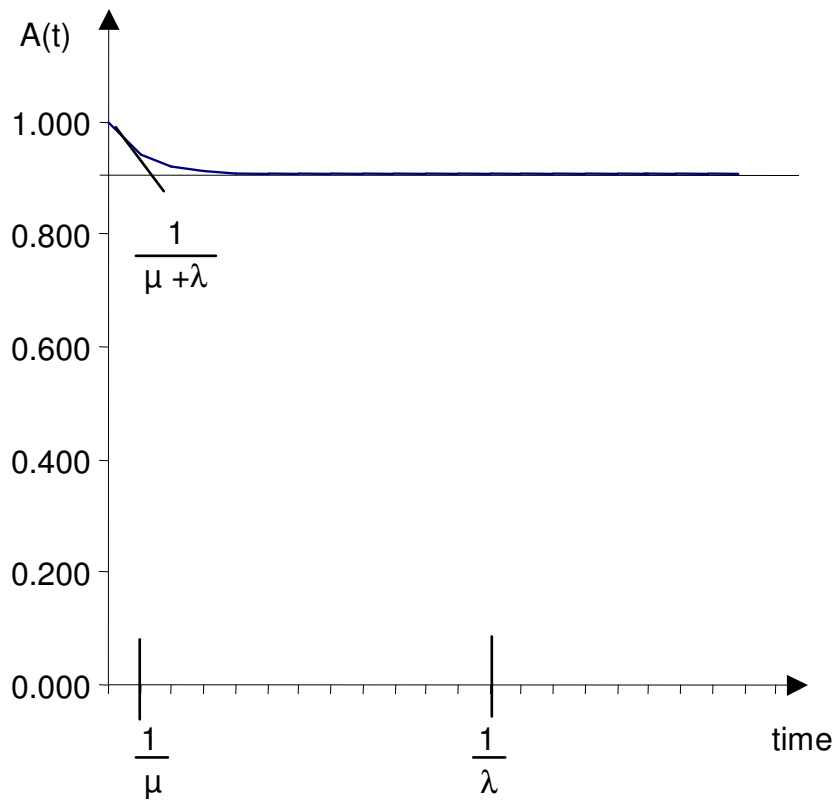


Figure 9-18: Availability in function of time

Although calculating the reverse Laplace transform is an interesting hobby, it is seldom necessary to determine the exponential time function, since the most important parameter is the steady-state value, which is reached in a very short time (with a time constant about equal to $1/\mu$, since normally $\mu \gg \lambda$).

Applying the end theorem of the Laplace transform that:

$$\lim_{t \rightarrow \infty} f(t) = \lim_{s \rightarrow 0} s F(s)$$

We find the long-term availability as:

$$A = 1 - \frac{\lambda}{\mu}$$

The unavailability is:

$$\bar{A} = \frac{\lambda}{\mu}$$

Note that the availability depends only on the relation of λ to μ , and not on their absolute values.

Example:

Given a mean time to repair of 10 hours, and a mean time to fail of 1000 hours, the unavailability will be 0.01, and the availability close to 0.9901%.

9.3.2.1 Influence of merging states

Let us consider a dual system with repair (Figure 9-19.a):

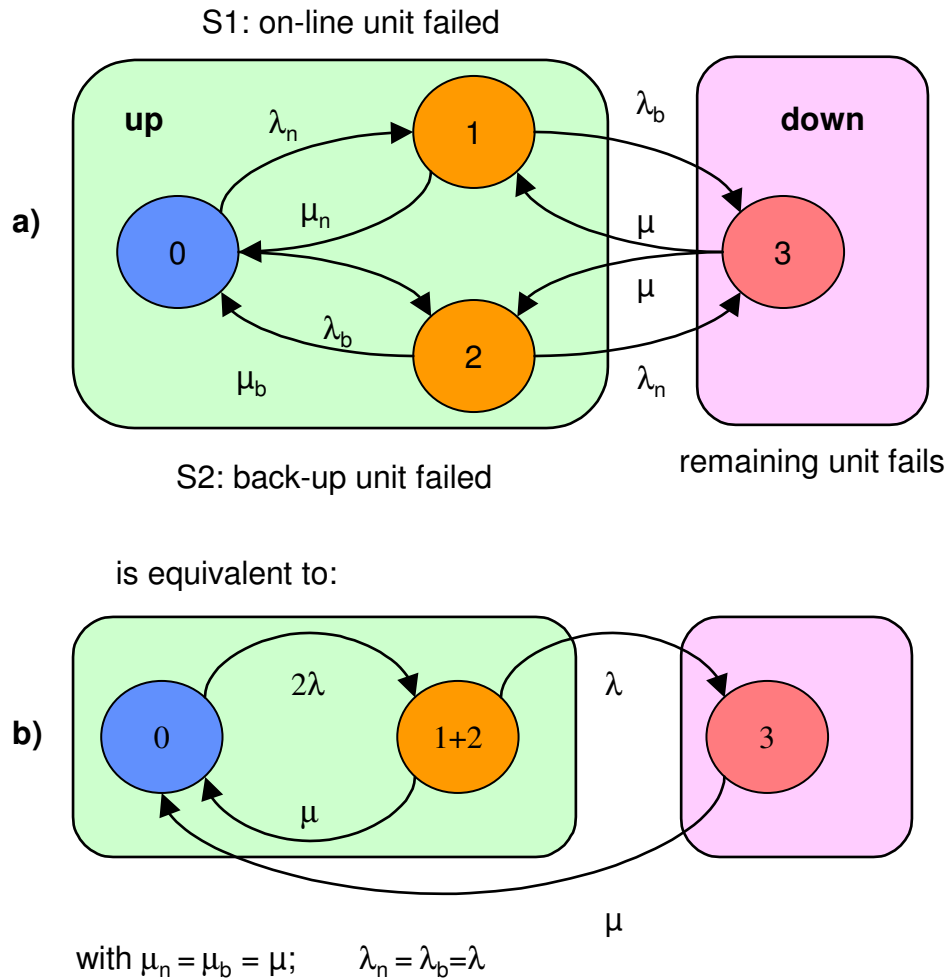


Figure 9-19: Dual system with repair

The states P1' and P1" of Figure 9-19.a can be collapsed into a single state P1 when both redundant units have the same λ and μ , like Figure 9-19.b shows.

We calculate the differential equations:

$$\frac{dp_0}{dt} = -2\lambda p_0 + \mu p_1 + \mu p_2$$

$$\frac{dp_{1+2}}{dt} = +2\lambda p_0 - (\lambda + \mu) p_{1+2}$$

$$\frac{dp_3}{dt} = +\lambda (p_1 + p_2)$$

We find the following expressions:

The above model assumed that the repair rates for both units are independent. This is only the case when there are two repair teams, for instance. Normally, only one unit at a time may be repaired. In that case, the repair rate from state P2 to state P1 is only μ , as suggested in Figure 9-19. The reason for this becomes obvious when one considers that the diagram of Figure 9-19.b was obtained by collapsing states P1' and P1" of Figure 9-19.a. If there is only one repair team, then the team must choose to repair unit A or unit B first, and either the transition from P2 to P1' or from P2 to P1" in Figure 9-19.a is missing. The availability of the dual system with a single repair team is then:

In general, for any system, we can set up the Markov transition diagram and the differential equations. Then, the probability of the "good" states are summed and the limit value is calculated. It makes no sense to calculate the

time function of available systems, since availability rapidly reaches an equilibrium point. To calculate the steady-state availability, it is not necessary to compute the Laplace Transform: a simple linear equation is sufficient, since in the stationary state, all dP/dt terms are zero. One must just add to the equation system the condition:

$$\sum_{i=0}^{N-1} P_i = 1$$

However, once the number of states exceeds three, the numerical calculations become rather tedious. Furthermore, the analytical solution is only interesting when one can intuitively view the influence of each factor. Where the expression becomes too complicated, it is better to proceed to a numerical solution and test the influence of the factors.

9.3.3 A simpler solution

In fact, we do not need Laplace to obtain the long-term availability, since we can assume that the system is stable after an infinite time and therefore all dp_i / dt terms are zero.

Just setting the left side of the equations to 0 is not sufficient, since the equation system is indeterminated – we have to introduce the initial conditions.

We can do this under the form that the sum of the state variables is always 1.

Therefore, the above equation becomes:

$$\begin{cases} 0 = -\lambda P_0 + \mu P_1 \\ 1 = P_0 + P_1 \end{cases}$$

and the availability is equal to the sum of the up states (there is only P_0 here), with the same result as above.

Note also that the MTTF is infinite in available systems – there is no absorbing state.

9.3.4 General solution for availability

After this introduction, we state the general solution for available systems

- a) decompose the system in states with defined transitions (failure and repair)
- b) for each state, determine the inflow and the outflow (Figure 9-19)
- c) set up the difference equations by letting the derivative term to zero
- d) remove one the equations and replace it by the expression: $\sum p_i = 1$

$$\frac{dp_i(t)}{dt} = \overbrace{\sum_{k=1}^{k=N, k \neq i} \lambda_{ki} p_k(t)}^{\text{inflow}} - \overbrace{\sum_{k=1}^{k=N, k \neq i} \lambda_{ik} p_i(t)}^{\text{outflow}}$$

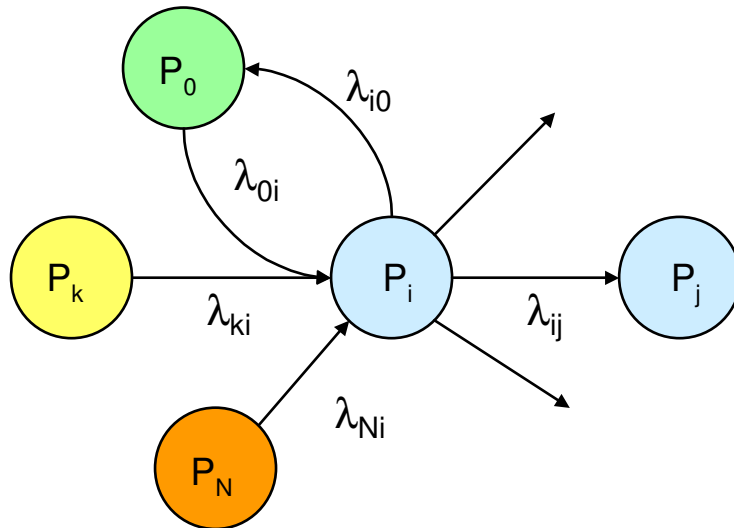


Figure 9-19 Inflow and Outflow

9.4 Example

Let's consider the availability of a repairable 1oo2 system, 1oo2r, such as Figure 9-20a depicts.

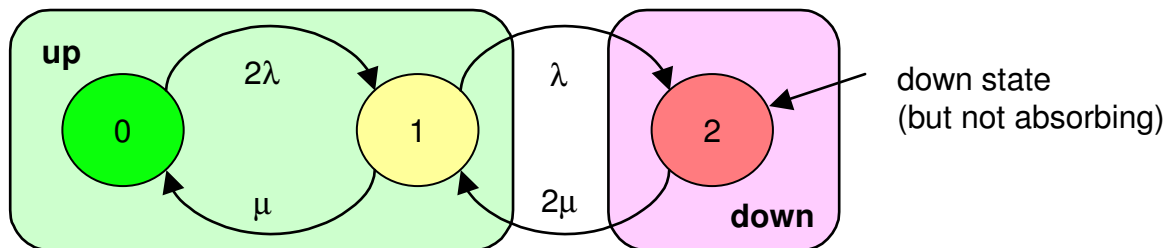


Figure 9-20a – repairable 1oo2 with two repair teams

The transition 2μ expresses that the repair of the failed parts takes place in parallel.

The equations are:

$$\begin{aligned} 0 &= -2\lambda P_0 + \mu P_1 \\ 0 &= +2\lambda P_0 - (\lambda + \mu) P_1 + 2\mu P_2 \\ 0 &= +\lambda P_1 - 2\mu P_2 \\ 1 &= P_0 + P_1 + P_2 \end{aligned} \quad \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix} = \begin{bmatrix} -2\lambda & \mu & 0 \\ 0 & \lambda & -2\mu \\ 1 & 1 & 1 \end{bmatrix} \begin{bmatrix} P_0 \\ P_1 \\ P_2 \end{bmatrix}$$

And the availability / unavailability are found as:

$$A = P_0 + P_1 = \frac{1}{1 + \frac{\lambda^2}{\mu^2 + 2\lambda\mu}} \quad D = \frac{\lambda^2}{\mu^2 + 2\lambda\mu}$$

9.5 Reliable systems

The same Markov models are used for reliable systems, although the equations are not identical, since a reliable system has at least one absorbing state.

9.5.1.1 Example: Reliability of a repairable 1oo2 system

Let's consider the reliability of a repairable 1oo2 system, 1oo2r, such as Figure 9-21a depicts.

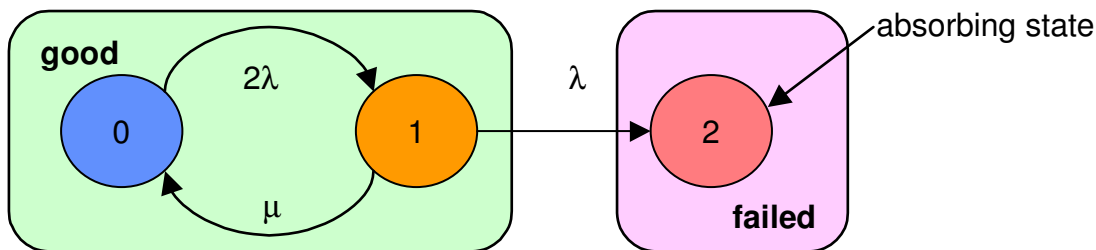


Figure 9-21a - State transition diagram of a 1oo2r system

The transition rate from the good state to the one unit failed is 2λ , since there are two units that can fail.

The repair rate is μ and the transition from the S1 to the S2 state is only λ , since there is only one unit that remains to fail. Figure 9-20b shows the corresponding hydraulic model, where a pump plays the role of repair.

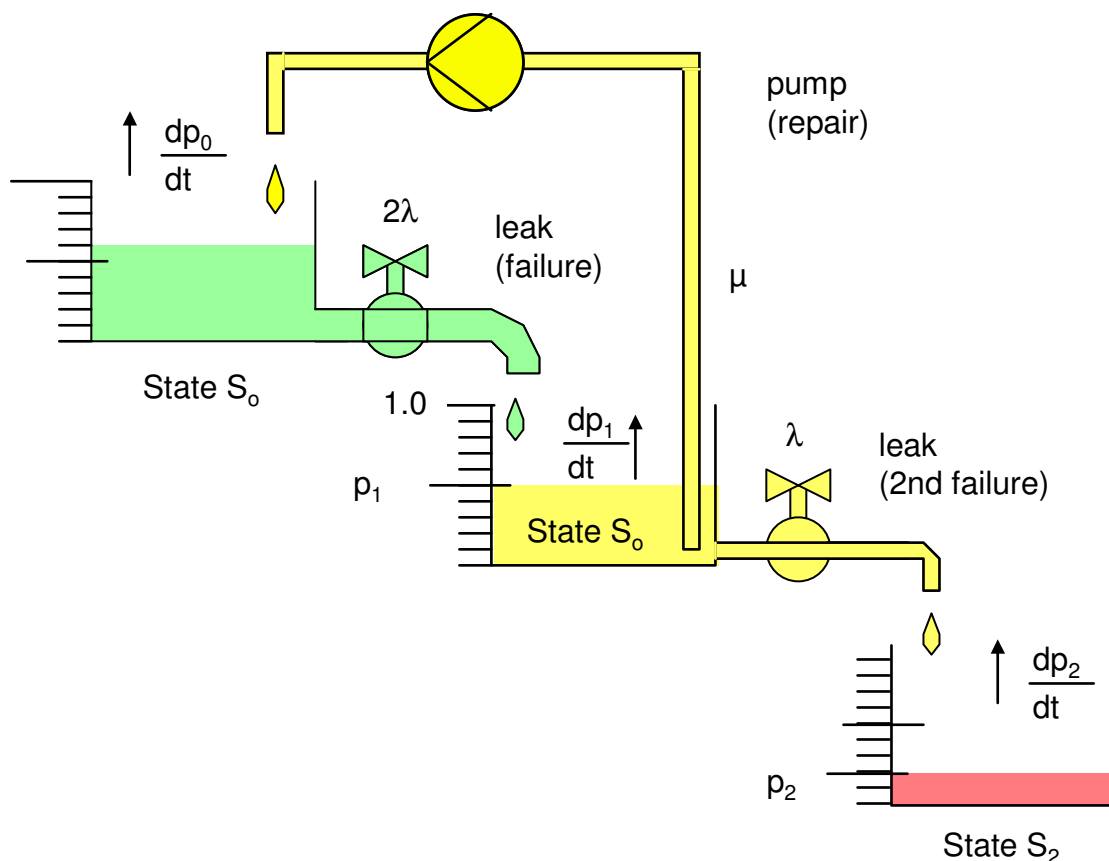


Figure 9-21b: hydraulic model of a 1oo2r system

The corresponding equations are:

$$\frac{dp_0}{dt} = -2\lambda p_0 + \mu p_1$$

$$\frac{dp_1}{dt} = +2\lambda p_0 - (\lambda + \mu) p_1$$

$$\frac{dp_2}{dt} = +\lambda p_1$$

initial conditions:

$$p_0(0) = 1 \text{ (initially good)}$$

$$p_1(0) = 0$$

$$p_2(0) = 0$$

The solution of these equation can obtained by a Laplace transformation (granted, it is tedious):

$$R(t) = p_0(t) + p_1(t) = \frac{(3\lambda + \mu) + W}{2W} e^{-(3\lambda + \mu - W)t} - \frac{(3\lambda + \mu) - W}{2W} e^{-(3\lambda + \mu + W)t} \quad \text{with: } W = \sqrt{\lambda^2 + 6\lambda\mu + \mu^2}$$

Figure 2-24 shows examples of results:

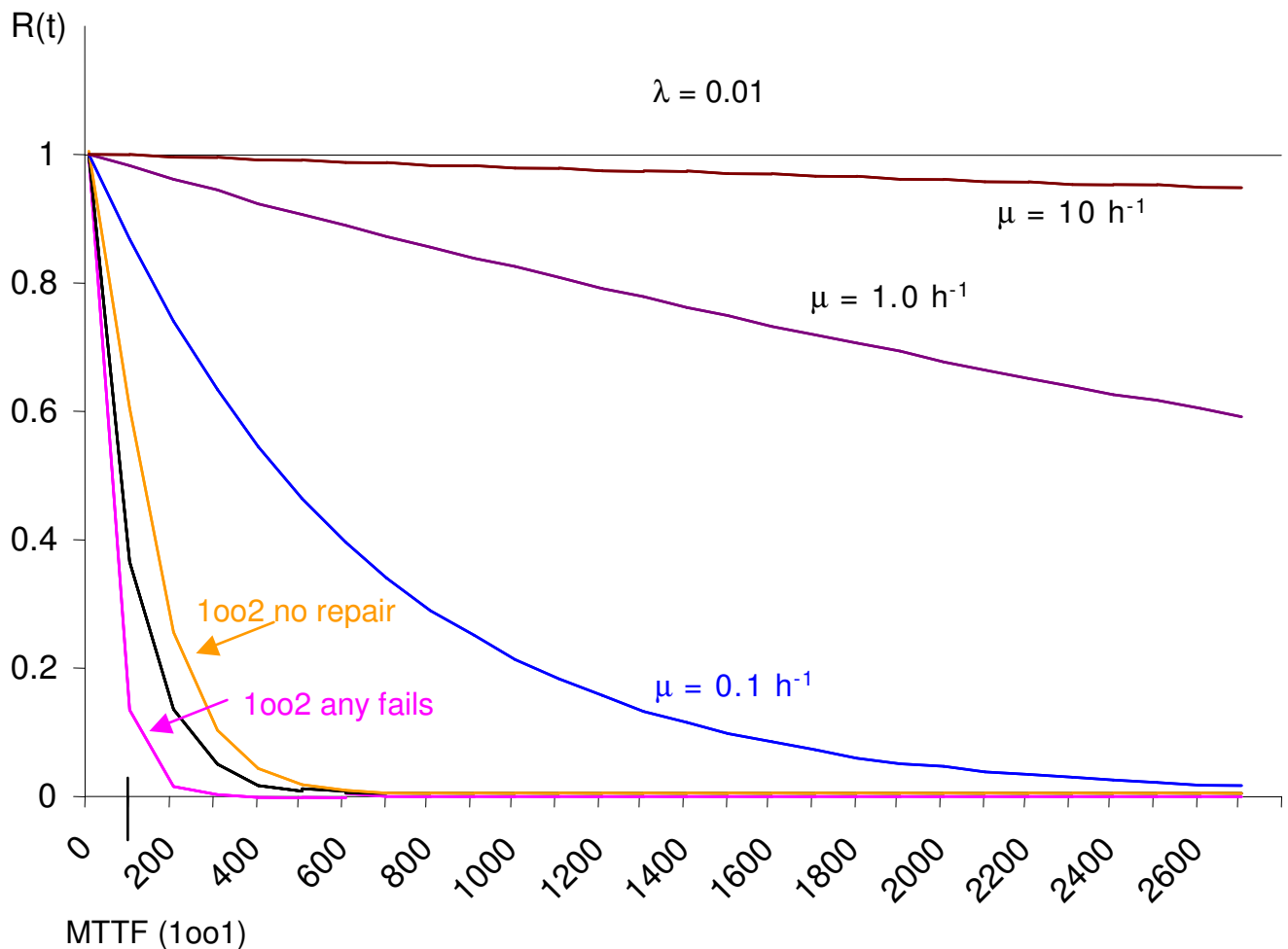


Figure 9-22: Reliability of a repairable 1oo2 system

9.5.1.2 Mean time to fail of repairable systems

An interesting consideration when looking at Figure 9-22 is that the plots look very much like a non-redundant system, only with a much high reliability. When μ tends to 0 (instant repair), reliability becomes perfect.

For repairable systems, it makes therefore sense to operate with the Mean Time to Fail.

This number is easy to compute with Laplace by applying the boundary theorem:

$$\lim_{T \rightarrow \infty} \int_0^T p(t) dt = \lim_{s \rightarrow 0} s P(s)$$

In the 1oo2 example, the equations become (we keep only two equations of the three)

$$\begin{aligned} -1 &= -2\lambda P_0 + \mu P_1 \\ 0 &= +2\lambda P_0 - (\lambda + \mu)P_1 \end{aligned}$$

and the MTTF is computed as:

$$MTTF = P_0 + P_1 = \frac{(\mu + \lambda)}{2\lambda^2} + \frac{1}{\lambda} = \frac{\mu/\lambda + 3}{2\lambda}$$

9.6 General solution for the MTTF of reliable systems

Like for the availability, there exist a simple way to compute the MTTF:

- e) decompose the system in states with defined transitions (failure and repair)
- f) for each state, determine the inflow and the outflow (Figure 9-19)
- g) remove one the equations
- h) set up the difference equations by letting the derivative term to zero, except the first
- i) calculate the sum of the non-absorbing states

9.6.1 Example:

Refer to the table in section 9.10. for a summary of available systems.

The diagram illustrates a 3-bit Markov chain for a system with three components (A, B, C). The system can be in a "system up" state (green background) or a "system down" state (pink background). Transitions are labeled with failure rates λ_A , λ_B , and λ_C .

System Up States (Green):

- ABC (all good)
- $\bar{A}BC$ (1 bad: A)
- $A\bar{B}C$ (1 bad: B)
- $AB\bar{C}$ (1 bad: C)

System Down States (Pink):

- $\bar{A}\bar{B}C$ (2 bad: A, B)
- $\bar{A}B\bar{C}$ (2 bad: A, C)
- $A\bar{B}\bar{C}$ (2 bad: B, C)
- $\bar{A}\bar{B}\bar{C}$ (all bad)

Transitions:

- From ABC to $\bar{A}BC$ (λ_A), $A\bar{B}C$ (λ_B), and $AB\bar{C}$ (λ_C).
- From $\bar{A}BC$ to $\bar{A}\bar{B}C$ (λ_B) and $\bar{A}B\bar{C}$ (λ_C).
- From $A\bar{B}C$ to $\bar{A}\bar{B}C$ (λ_A) and $A\bar{B}\bar{C}$ (λ_C).
- From $AB\bar{C}$ to $\bar{A}\bar{B}C$ (λ_C) and $A\bar{B}\bar{C}$ (λ_B).
- From $\bar{A}\bar{B}C$ to $\bar{A}\bar{B}\bar{C}$ (λ_C).
- From $\bar{A}B\bar{C}$ to $\bar{A}\bar{B}\bar{C}$ (λ_A).
- From $A\bar{B}\bar{C}$ to $\bar{A}\bar{B}\bar{C}$ (λ_B).
- From $\bar{A}\bar{B}\bar{C}$ to $\bar{A}\bar{B}\bar{C}$ (λ_A).

We may also be interested in the mean time between failures of a redundant system. To calculate the MTBF, one must change the "Down" states into trapping states. The probability of being in the "Up" states is determined, and the mean time between failures is obtained as the integral over time.

Figure 9-20: MTTF of a Dual, Repairable System.

 $\lambda = 1/1000$ hours, $\mu = 1/10$ hours.

MTTF with repair: 201'500 hours

The mean time between failures of **elements** of the system is of course much shorter. It can be calculated by making every other state than the fault-free state a trapping state. In the above diagram, the MTBF of any of the part is equal to $\mu / 2$, that is, half of that of a single system. This is not surprising, since there is the double number of parts that can fail.

189

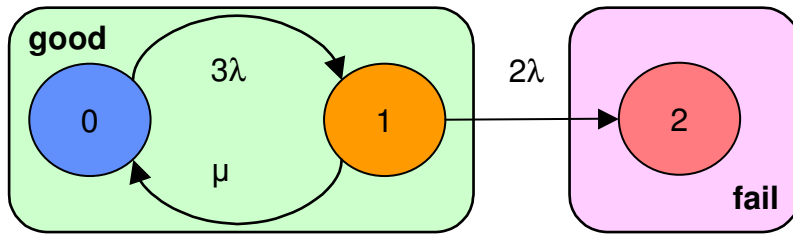


Figure 9-21: Markov diagram of a (reliable) 2/3 voting system (TMR)


The reliability of that system can be computed (using Laplace) as:

$$\tilde{P}_0 = \frac{s + \mu + 2\lambda}{s^2 + s(\mu + 5\lambda) + 6\lambda^2} \quad \tilde{P}_1 = \frac{3\lambda}{s^2 + s(\mu + 5\lambda) + 6\lambda^2}$$

$$d = \sqrt{(\mu + 5\lambda)^2 - 6\lambda^2} \quad s_{1,2} = \frac{-(\mu + 5\lambda) \pm d}{2}$$

$$R(t) = \frac{s_1}{d} e^{-s_1 t} + \frac{s_2}{d} e^{-s_2 t}$$

$$MTTF = \frac{1}{\lambda} \left(\frac{5}{6} + \frac{\mu}{\lambda} \right)$$



 repair term

Numerical

example:

$$\lambda = 0.001, \quad \mu = 0.1, \quad MTTF = 34'166 \text{ hours}$$

Refer to the tables in section 9.10 for a summary of reliable systems. Examples of the calculations for fault tolerant systems are given in [Muron 76] and [Siewiorek 77].

9.6.3 Influence of imperfect redundancy

The above calculations show that the reliability or the availability can be boosted by repair, practically in proportion with the repair rate to failure rate. In reality, the increase is far lower, due to the fact that redundancy is seldom perfect.

Until now, we assumed that the failure of a unit would not influence the functionality as long as one redundant unit is still available. This may be true for a matchbox, but in computing systems, the masking of an error or the switching in of a spare unit is an operation that is bound with a certain risk. For instance, in a dual system, we must take into account that the probability of successful switchover to the spare unit is less than one. This probability is called the **coverage**.

More formally, coverage is defined as "the conditional probability that, given the existence of a failure in the operational system, the system is able to recover" by [Bouricius 69]. The word "coverage" is also used in conjunction with error detection. Here it is defined as the percentage of errors detected within the allocated error latency time in relation to the total number of errors.

In fact, if we assume that any detected error can be corrected, and no other uncertainties remain, then the error coverage is identical to the recovery coverage. The error detection coverage is indeed the most important factor in coverage, and therefore coverage can be enhanced significantly by improving the error detection mechanisms.

Let's consider a dual system. We assume that only one of the redundant units is performing calculations, while the other remains as hot stand-by. When an error occurs, it is detected with a certain probability and switchover takes place. The probability that the switchover successfully takes place is the coverage (c). Figure 9-22 shows the corresponding Markov diagram.

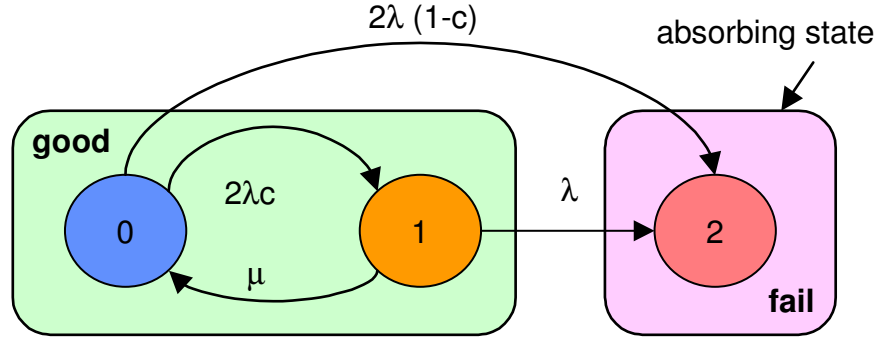


Figure 9-22: Dual redundant, reliable system with coverage $c < 1$.

Note that the coverage does not apply to a failure of the stand-by unit, since it is assumed that a failure of the standby is not critical.

The resulting equations are:

$$\begin{cases} -1 = -2\lambda P_0 & + \mu P_1 \\ 0 = +2\lambda c P_0 & - (\lambda + \mu) P_1 \end{cases}$$

$$MTTF = P_0 + P_1 = \frac{(1+2c) + \mu/\lambda}{2(\lambda + \mu(1-c))}$$

As an example, take the above case of the redundant system with

$$\lambda = 10^{-3}/h, \mu = 1/10/h.$$

MTTF without repair:	1500 hours
MTTF with repair:	201500 hours
MTTF with $c = 0.9$:	8575 hours

We see that even a small lack of coverage is sufficient to let the reliability drop drastically. In fact, it makes no sense to increase the reliability of the system by reducing the failure rate or increasing the repair rate as long as c is large, since the MTTF reaches rapidly a limit that is independent from the repair rate μ :

$$\lim_{\lambda/\mu \rightarrow 0} MTTF = \frac{1}{\lambda(1-c)}$$

On the other hand, it makes no sense to increase the coverage when the repair rate is low, since the MTTF tends to the value:

$$\lim_{\mu \rightarrow 0} MTTF = \frac{1}{\lambda} \left(\frac{3}{2} + \frac{\mu}{2\lambda} \right)$$

An optimum is reached when: $(1-c) \approx \lambda/\mu$

9.7 Dependability and performance

Until now, we assumed that the system was either working or failed. We can imagine several variations in between, especially under the aspect of multiprocessor computers. **Graceful degradation** expresses the fact that a redundant system goes through several stages of functionality until it eventually fails.

Graceful degradation applies especially to multi-processor systems, in which the tasks are executed by a pool of processors, which are interchangeable with respect to the tasks they execute. The general case considers that the processing power is a function of the number of intact processors. When all processors are working, the computing power is highest, and diminishes as the processors fail. Only when the last processor is down, does the system fail.

Although conceptually simple, graceful degradation is not easy to implement. It supposes that:

The redundant unit is an active resource (participates in the function);

Activities can be paralleled so as to take advantage of the availability of multiple resources;

Reconfiguration is reliable (e.g. means exist to save the data held by a failed processor).

The second point is probably the most complicated to solve. Indeed, multiprocessors increase the computing power, provided that the application can be divided into parts that can run in parallel. No general solution exists to perform the parallelisation of applications, and many algorithms do not lend themselves to a parallel execution.

Therefore, graceful degradation is rather complex to implement, but its calculation is simple. In fact, as far as redundant systems without repair are concerned, calculations rely solely on the assumption that k-of-N units are working (see paragraph 9.3.5). We can then assume that there is a weighting factor for each number of working processors.

The Markov diagrams that describe the failure model determine the probability of being in a particular state. The availability of computation power is then built as a weighting function over the existing states.

A general model has been given by Baudry [Baudry 77].

9.7.1 Safety evaluation

The safety of a system has been defined as the probability that a disruption will not cause damage in excess of an acceptable value. Safety can only be defined with a specific application in mind, for which safe and unsafe states can be defined.

In a computer, we consider that two kinds of behaviour can endanger the plant: either a loss of function for a time longer than an acceptable time, or the output of erroneous data.

9.8 Tables

We now summarize the calculation for each of the most frequent cases:

9.9 Summary of the definitions

Dependability: a subjective measure of the user's trust in the system.

Non-repairable systems:

R(t): Reliability: the probability that a non-repairable item has not failed at time t, given it was in a working condition at time $t = 0$.

I: Integrity: the probability that a computing element does not produce false data which cannot be recognized as such (called Credibility in [Meraud]).

MTTF: Mean Time To Fail: the expected lifetime of a non-repairable system.

Repairable systems:

MTBF: Mean Time Between Failure. The mean interval between two successive failures of the same repairable item.

MTTR: Mean Time To Repair. The average time it takes to repair an item.

MUT:	Mean Up Time: The mean time during which a repairable item is operational.
MDT:	Mean Down Time: The mean time during which an item is out of service because of failure.
A:	Availability: the relation of the MUT to the total lifecycle of the system, $MUT + MDT$.
U:	Unavailability: the ratio of MDT to MUT in the steady state.
Fault-tolerant Systems:	
MTTMF:	Mean Time To Mission Failure is the life expectancy of a reliable fault-tolerant system. It is identical to MTTF for the non-redundant case.
MTBEF:	Mean Time Between Element Failures is the average time between two successive failures of elements (but not necessarily the same ones) of the system. The MTBEF depends on the MUT/MDT or MTTF of the individual parts and on the amount of redundant parts. MTBEF is identical to MTBF in the non-redundant case.
MTTER:	Mean Time To Element Repair: (applies only to a repairable unit) the mean time it takes to repair a failed element of an on-line repairable system. During that time, the system can remain operational because it is fault-tolerant, but further failures can bring it down if redundancy is exhausted.
MRT:	Mean Recovery Time: This is the mean time needed by a fault-tolerant computer to recover from a failure and resume operation. If the computer is unable to recover during a time T_{domax} , then the system will be considered as failed. Further failures during the recovery time may lead to a mission failure. This time is essentially identical to the MTTER in the case of automatic repair.

9.10 References

- [Baudry 77] M.D. Baudry,
"Performance related reliability measures for computing systems",
7th Int. Symposium on Fault-tolerant Computing, FTCS-7, Los Angeles, California, June 29-30, 1977
- [Birolini 85] A. Birolini,
"Qualität und Zuverlässigkeit technischer Systeme",
Springer-Verlag, 1985
- [Bouricius 69] W.G. Bouricius, W.C. Carter and P.R. Schneider,
"Reliability Modelling Techniques for self-repairing computer systems",
pp. 295..309, Proc. 24th Annual Conference of the ACM, 1969
- [Hopkins 78] A. Hopkins et al.,
"FTMP - A Highly Reliable Fault-Tolerant Multiprocessor for Aircraft",
Proceedings of the IEEE, Vol. 66, No. 10, OCT 1978, pp. 1221 - 1239
- [MIL 82] Dept. of Defence, U.S.A,
Military-Handbook-217D, MIL-HDBK-217D,
"Reliability Prediction of Electronic Equipments",
January 15, 1982
- [Muron 76] O. Muron, C. Meraud, F. Browaeys,
"Markov Reliability Modelling of a reconfigurable system applied to the COPRA computer",
SAGEM, 6 Avenue d'Iena, 75783 PARIS CEDEX
- [Shooman 68] M. Shooman,
"Probabilistic Reliability: an Engineering Approach",
Mc Graw Hill Company, LCC 68-13099

- [Siewiorek 77] D. Siewiorek,
"Multiprocessors: Reliability Modelling and Graceful Degradation",
INFOTECH-State-of-the-Art Conference on System Reliability, London, 1977
- [Siewiorek 78] D. Siewiorek et al. ,
"A Case Study of C.mmp, Cm*, and C.vmp, part 1 and 2",
pp 1178 .. 1220, IEEE Proceedings, October 1978
- [Siewiorek 82] D. Siewiorek & R. Swarz,
"The Theory and Practice of Reliable System Design",
DIGITAL Press, 1982
- [Spectrum 81] IEEE Spectrum,
Special Issue on Reliability,
October 1981

10 Appendix A Linguistic Note on Terminology

A precise terminology is essential for a common understanding, especially in the literature. Unfortunately, the terms used in reliable computing are strongly biased by their everyday use and by predefined formulas. We speak of fault tolerance, while in some cases it would be preferable to speak of error tolerance. Further, different languages express subtleties that do not exist in others. The results are endless discussions on terminology [Laprie 85]. We try here to reconcile the common sense of the words and a precise definition. We need a set of terms to describe the following situations:

- An item ceases to provide the required service: the transition from service to lack of it is a **failure** of that item. The word "failure" makes no assumption with regard to the duration, which can be definitive or temporary: "a failure occurred".
- An item is not working because of a failure: this is a **malfunction**, or a **breakdown**. While a malfunction is momentary, a breakdown implies a lengthy repair with human intervention. The terms "malfunction" and "breakdown" express a state, not an event like "failure". However, in normal usage, "failure" also refers to the period after the failure, without assumptions about the duration, but this usage should be avoided.
- If the malfunction is so long that the mission cannot be fulfilled any more, this leads to a **mission failure**, which is definitive. The English language does not distinguish "mission failure" and "failure". The opposite of "mission failure" is "mission success", the opposite of "failure" is "function";
- A failure is caused by something. The cause of a failure is a **fault**. The fault may be an external perturbation or violence, a programming mistake, an exceeded temperature, a bonding fault, but it can also be the result of a poor design. One can argue whether the design itself has a flaw, or whether the design incorrectly protected the system against external threats. Such an expression as "we are looking for the failure" is therefore incorrect, what is looked for is a fault. A fault does not immediately nor necessarily cause a failure;
- A fault may itself be caused by something which prepared the conditions for its occurrence. A **defect** is something that impairs the quality of a system. A defect can be so bad that the system does not work any more, it can be something which can lead the system to fail at some later time, it can be just some cosmetic defect. A fault may be a defect or an external action. Conversely, an external action can cause a defect. Unfortunately, fault and defect are sometimes used in the same sense ("this gem has a fault").
- In computing or logical elements, a fault expresses itself as a deviation from the intended state, which is especially visible at the output. This is called an **error**. An error is a manifestation of a fault. The error may originate from a faulty hardware (hardware fault), or also from a programming mistake (design fault). A design error or programming mistake introduces a defect in the program. This defect can remain unnoticed for a long time (lurking fault) until it manifests itself and causes data errors. These errors may be permanent or appear from time to time (intermittent). If these errors cannot be tolerated or corrected, a mission failure follows.

Now, the terminology gets somewhat more complicated because of the nested nature of the phenomenon. A failure is due to a fault, which is itself caused by a failure, and so on.

For instance, a data error is due to a fault in the floating point unit, which is due to the failure of a circuit, which is due to a design fault (a failure of the designer). Or, an error is due to a programming fault, which is itself the consequence of an error of the programmer.

Consider a computer as an abstract machine, consisting of a physical layer (hardware), a logical layer (signals), an algorithmic layer and an application layer. This hierarchical view sheds a different light on these terms [Avizienis 82]:

At the physical level, a departure from the specifications is seen as a physical fault. For instance a physical fault would be that transistor burns open because of overheating. The failure of the transistor introduces a fault in the circuit.

At the logical, or signal level, a failure of a circuit occurs when a logical value is turned into its opposite. The fault is generally a shorted or open circuit, for instance a burned transistor, a bridge or an open-circuit. Since this is the level one can view with measuring instruments, the word "**fault**" has become a special meaning: since disturbances in logical circuits and power outages are caused by unintended current flows, "fault" describes such phenomenon. Circuit designers use the terms "bonding fault" or "stuck-at-one fault" (which, more correctly, should be called stuck-at-high fault), power systems engineers use terms like "grounding fault". So fault has both the meaning of "origin of an arbitrary failure" and "origin of an electrical failure".

At the algorithmic level, one speaks of errors, which are incorrect data items (data error, error correction, etc..). The errors are caused by faults in the logic or by design faults. The same term is used to describe higher-level errors (algorithm errors). Note that an error may show up a long time after the fault took place;

At the application level, the failure of a part is a fault in the system, which can cause a mission failure if the system is not fault-tolerant. Here also, a long time may elapses between occurrence of the error and mission failure, possibly because the erroneous parts were seldom used.

The following Figure A1-1 shows the hierarchy of faults and failures:

APPLICATION (MISSION) FAILURE
 ALGORITHMIC (DATA) ERROR
 LOGICAL (LOGICAL) FAULTS
 PHYSICAL (COMPONENT) FAILURE

Fig. A1-1. The four universes of reliability

Of course, when one attributes a fault to a design fault, then this is considered an error of the designer, thereby treating the designer as a (faulty) logical machine.

When we try to port these definitions into several languages, the difficulty is that there is no direct correspondence between the terms. We choose here a correspondence of terms which tries to reflect the usage of these terms in the literature. Note that some terms occur several times, since some languages use them in different meanings (Figure A1-2):

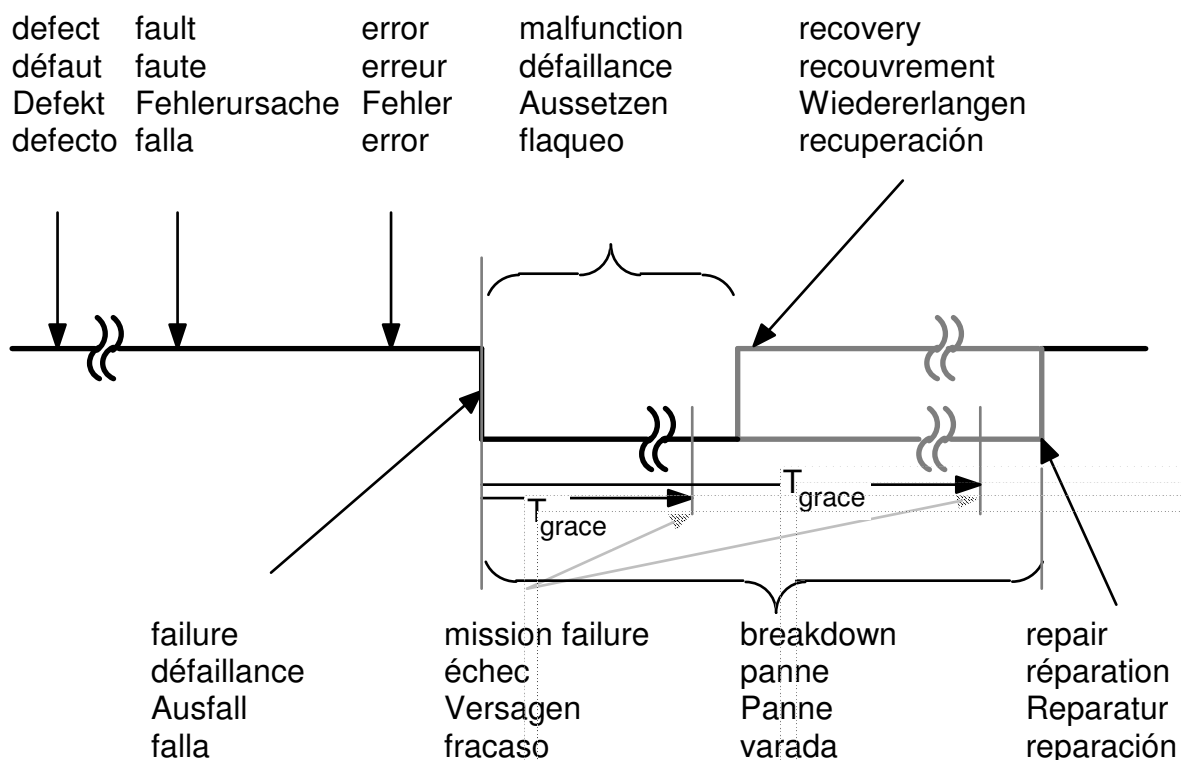


Fig. A1-2. Fault History

The most important terms are summarized in this table:

English	Français	Deutsch
reliance	dépendence	Verlass
dependability	sûreté de fonctionnement	Verlässlichkeit
reliable	fiable	zuverlässig
reliability	fiabilité	Zuverlässigkeit
available	disponible	Verfügbar
availability	disponibilité	Verfügbarkeit
failure	défaillance	Ausfall
mission failure	échec	Versagen
failure rate (de défaillance)	taux de pannes	Ausfallrate
malfunction, (défaillance)	panne	Aussetzen
breakdown	panne	Panne
damage	dommage	Schaden
defect, flaw	défault	Defekt
fault	faute	Fehler(-ursache)
error	faute, erreur	Fehler, (Irrtum: bei Menschen)
safety	sécurité	Sicherheit
security	sûreté	Schutz
susceptibility	susceptibilité	Anfälligkeit
persistence	persistance	Beharrlichkeit

nur

10.1 References of Appendix 1

- [Avicienis 82] A. Avicienis,
"The four-universe information system model for the study of fault-tolerance",
Proceedings of the 12th Symposium on Fault-Tolerant Computing FTCS-12,
pp 6..11, Santa Monica, California June 1982
- [Laprie 85] J.C. Laprie,
"Dependable Computing and Fault-Tolerance: Concepts and Terminology",
Proceedings of the 15th Symposium on Fault-Tolerant Computing FTCS-15,
Ann Arbor, Michigan, pp. 2..9,
June 1985

