Terminology and paradigms for fault tolerance

Terminology and Paradigms for Fault Tolerance

by

Henk Schepers

91/08

May, 1991

COMPUTING SCIENCE NOTES

This is a series of notes of the Computing Science Section of the Department of Mathematics and Computing Science Eindhoven University of Technology. Since many of these notes are preliminary versions or may be published elsewhere, they have a limited distribution only and are not for review. Copies of these notes are available from the author or the editor.

Eindhoven University of Technology
Department of Mathematics and Computing Science
P.O. Box 513
5600 MB EINDHOVEN
The Netherlands

Terminology and Paradigms for Fault Tolerance[‡]

Henk Schepers

Department of Mathematics and Computing Science
Eindhoven University of Technology
P.O. Box 513, 5600 MB Eindhoven
e-mail: schepers@win.tue.nl

Abstract

For the purpose of a clear discussion, this paper will introduce terms and paradigms used for fault tolerance. In this paper first the notions system, component, environment, design, structure and behaviour are presented. Then, the notions failure and fault are defined. After observing that designing a fault tolerant system is based on the use of redundancy and fault hypotheses, the fundamental techniques to achieve fault tolerance are identified. Then, a number of paradigms that are popular for fault tolerance are discussed.

1 Introduction

According to Laprie (cf. [12]) fault tolerance is the property of a system "to provide, by redundancy, service complying with the specification in spite of faults having occurred or occurring." This paper will define notions for fault tolerance, in order to obtain a clear terminology. After discussing typical paradigms it will present two case studies.

The paper is organized as follows: in section 2 the notions system, component, environment, design, structure and behaviour are presented and a terminology for fault tolerance is defined. In section 3 an extensive classification of failures is given. In section 4 the various stages of the process of treating faults are mentioned and it is discussed what designing for fault tolerance is about. In section 5 typical paradigms for fault tolerance are discussed. Section 6 presents, as a case study, the design of a stable storage. Lastly, section 7 presents, also as a case study, the design of a reliable broadcast mechanism.

2 Failures and Faults

A system consists of components which interact as described by a design. There is no conceptual difference between a system and a component: the system is simply the component under discussion. For the scope of this paper it is not important whether a

[‡] This research is supported by the Dutch STW under grant number NWI88.1517: "Fault Tolerance: Paradigms, Models, Logics, Construction."

component is of physical nature (i.e. a hardware component) or of non-physical nature (i.e. a software component). A system receives input from and delivers output to the environment, i.e. a system provides service in response to requests from the environment. Since it is also true that a component provides service in response to requests from the system, there is no conceptual difference between a system and an environment either. Following Wieczorek and Vytopil (cf. [21]) 'system' designates the entity at the current level of consideration, 'environment' designates the entity at the superordinate level and 'component' designates an entity at the subordinate level.

A system is characterized by its behaviour and its structure in terms of components and their interrelations. The behaviour of a system can be separated into two distinct categories: behaviour in accordance with the specification and behaviour *not* in accordance with the specification. A *failure* of a system occurs when the behaviour of the system deviates from that required by its specification [15]. An extensive classification of failures is presented in section 3. The failure of a component, but also the failure of the environment and an incorrect design, will be referred to as a *fault*. Besides the intended, standard behaviour the behaviour that is part of the process of treating faults should be part of the system's specification.

When discussing hardware defects, the notions transient and permanent are well established [1]. A *transient* defect is present for only a limited period of time (no longer than some threshold) after which it spontaneously disappears. Any defect which is present for longer than that threshold period is said to be *permanent*. Analogous to this, a system may fail transiently or permanently.

3 A Classification of Failures

Assuming the system is a discrete event system, the specification of the system's behaviour consists of time, value and location requirements. A time, value and/or location failure occurs if the system's behaviour does not conform to those requirements. It is natural to divide time failures in late behaviour, which can lead to omission, and early behaviour, which can lead to overrun. Likewise, since value requirements are usually formulated in terms of format and range, it is natural to distinguish a violation of the format from a violation of the range, i.e. underflow or overflow.

If it is possible to deduce from assertions about the system's behaviour that some failure has occurred, we call that failure detectable. Different failure models arise from the assumptions about the correctness of the time (T), value (V) and location (L) behaviour, and, in case that behaviour is not assumed to be correct, the detectability of time, value and location failures. Then, such models have the format:

$$T\left\{\begin{array}{c} C \\ D \\ U \end{array}\right\} V\left\{\begin{array}{c} C \\ D \\ U \end{array}\right\} L\left\{\begin{array}{c} C \\ D \\ U \end{array}\right\},$$

where C, D and U denote correct, detectably incorrect and undetectably incorrect respectively.

In the literature failure models such as the omission model [6] appear. With the above notation the omission model is equal to the TDVCLC model. The Byzantine models that appear in the literature (see for example [11] and [18]) apply to the distribution of messages. For such models it is necessary to consider the inconsistencies that may arise from value failures, i.e. they do not allow the assumption that every destination receives the same value.

4 Treating Faults

As mentioned in the introduction, fault tolerance is concerned with providing the specified service, even in the presence of faults. To do so, fault tolerance depends upon the effective deployment and utilization of redundancy[1].

Of course, a fault tolerant system can tolerate only a limited number of certain types of faults. Important for the design are the *fault hypotheses* stipulating the hypothetical character of possible faults. Under certain fault hypotheses, the system is designed as if the hypothetical faults are the only faults it can experience and measures are taken to tolerate (only) those *anticipated* faults. Other faults will generally cause the system to fail.

The most rigorous way to tolerate a fault is to use so much redundancy that it can be masked, for instance the triple modular redundancy paradigm presented in section 5.3. But this kind of redundancy is generally too expensive.

If faults cannot be masked, then our first concern is how to identify an anticipated fault (*fault detection*). Before the system can be allowed to continue to provide its service, *fault diagnosis* must be applied and the fault's — unwanted — consequences must be made undone. Leaving incorrect structures and environmental faults out of consideration for now, the fault diagnosis must identify the components that are responsible for the fault and also whether that fault is transient or permanent.

If the fault is only transient, the fault's consequences can be made undone by simply restarting the system[2], i.e. by putting it in some initial state, or, in case a valid system state is regularly recorded as a checkpoint, by bringing the system back to its last checkpoint and then continue the operation from that state. These techniques are called forward and backward error recovery respectively. After that the system can continue to provide its service.

If the fault is not transient but permanent — this is for instance the case with fail-stop components [16] — the system needs repair first. The faulty component can be replaced,

[1] In the literature there is a classification by what kind of element (for instance component and information) is replicated. This classification, however, is not orthogonal (for instance component redundancy also means information redundancy).

[2] This only helps, of course, if the application allows the involved delay; for real-time applications this usually is not the case.

in which case the system can deliver its service unmodifiedly, or other components must take over the faulty component's tasks in addition to their own tasks, which leads to a degradation of the service (*graceful degradation*). Replacing a faulty component can be done either physically or logically by means of *reconfiguration*, where, using (logical) switches, a faulty component is taken out of action and a spare, which is already present in the system, is put to action.

5 Some Paradigms for Fault Tolerance

To familiarize the reader with the fault tolerance field, a few typical paradigms will be presented and analyzed.

5.1 Consistency Check

Consistency check paradigms apply to those cases where the output of a component is checked with respect to its specified functionality. Especially when a component performs a mathematical function such paradigms are used, for instance by verifying whether the result conforms to the specified format (*syntax checking*), by verifying whether the result lies in the specified range (*range checking*) or by verifying whether the application of the reverse function to the result yields the input again (*reversal checking*).

5.2 Duplication With Comparison

If consistency checks are not feasible or too expensive, then the most rigorous way to detect the failure of a component is to duplicate it. Both components receive the same input and perform the same tasks. Their output is compared and only passed on if there is a match. Such a design leads to a fail-stop system: if one component fails, i.e. produces output that differs from the correct output, the system does not output anything. Under the assumption that if both components fail they produce different incorrect output, the system delivers correct output or none at all even if both components fail. When this paradigm is used to design fault tolerant hardware, for which it is very popular, the components are usually synchronized (see fig. 1). This synchronization is less stringent when used to design fault tolerant software.

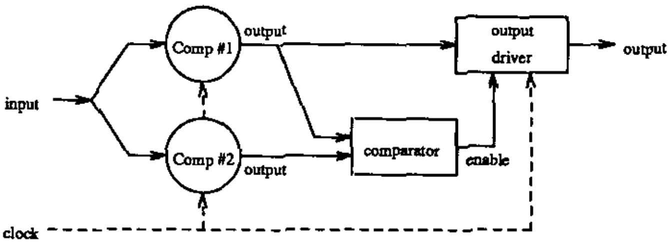

Figure 1: Duplication with comparison

5.2.1 Analysis of the Duplication With Comparison Method

The redundancy consists of an extra component plus the comparator and the output driver. The correctness hypothesis stipulates that all components must function correctly.

Since a component sends its output via one link only, there is no distinction between the failure of a link and the failure of the component using it. The failure of the comparator or the output driver results in the failure of the system. It may seem as if the system has become merely less reliable because of the larger amount of components, but because of the relative simplicity of both the comparator and the output driver, their failure is far less likely than the failure of one of the duplicated components.

5.3 Triple Modular Redundancy

The above mentioned duplication with comparison is capable of preventing the failure of a system, but if a failure of one of the duplicated components occurs the system outputs nothing. If the component is triplicated and another component acts as a *voter*, which passes the majority vote of the outputs of the individual components, the system can still produce correct output even when one of the triplicated components fails: its failure can be masked. This is known as the triple modular redundancy paradigm (see fig. 2). Again, the synchronization is less stringent when used to design fault tolerant software (for instance the SIFT system [20]).

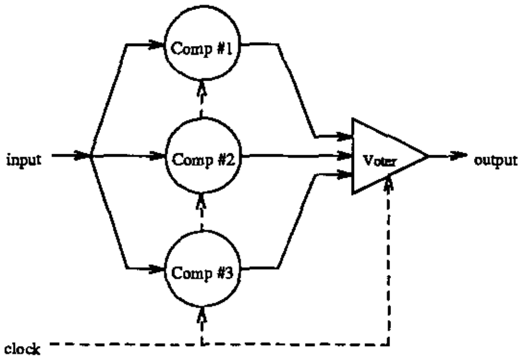

Figure 2: A triple modular redundant component

The triple modular redundancy paradigm can be generalized to N-modular redundancy ($N \geq 3$). In case the output of an N-modular redundant system is used as input for an M-modular redundant system, M voters are used.

It should be noted that instead of N equal components, N similar components can be used. N-version programming, a well known paradigm to minimize the consequences of programmer faults, typically uses N different implementations for the same functionality. For efficiency reasons those N versions are executed concurrently and the ultimate result is determined by a voting mechanism. The major drawback of N version programming is that the N versions are always more or less correlated and correlated faults are usually not detected. This drawback can be removed by (1) the use of self-checking versions, or (2) by applying a consistency check to the result of the first version, and, if the result does not pass, by executing the second version and applying a consistency check to the second result and so on, until a result passes the consistency test — the recovery block paradigm. Typically, a checkpoint is established before executing the first version.

5.3.1 Analysis of the Triple Modular Redundancy Method

Here the redundancy consists of the two replicas of the given component and the voter. Analogous to section 5.2.1 the failure of the links is left out of consideration.

The voter uses a majority vote on the outputs of the three components; this is possible as long as the outputs of at least two components are identical. The voter is usually designed to output nothing if no two of its inputs are identical. Clearly, when two components fail identically, incorrect output is produced.

From the above, it can easily be seen that the correctness hypothesis stipulates: "the voter does not fail and at most one of the three components fails."

5.4 Coding

A popular and effective method to protect data against corruption during transmission is the use of coding: a dataword is transformed into a codeword which contains some redundant bits. The use of coding is not restricted to communication, for instance it can be used to design a fault tolerant memory.

For two (binary) codewords of the same length, the *Hamming distance* [7] is the number of bit positions in which the two codewords differ, i.e. the number of single bit errors that are needed to convert one codeword into another. For example, the Hamming distance between the codewords 0000000000 and 1111111111 is ten. The Hamming distance of a complete code is equal to the minimum Hamming distance of all pairs of codewords in the code. For example, the Hamming distance of the code consisting of the codewords 0000000000, 0000011111, 1111100000 and 1111111111 is five.

Now, if a code has Hamming distance h, $h-1$ single bit errors cannot transform one codeword into another codeword. That code is thus capable of detecting up to $h-1$ single bit errors. Furthermore, if no more than $\lfloor\frac{1}{2}(h-1)\rfloor$ single bit errors occur, the original codeword is still closer than any other codeword. Hence, up to $\lfloor\frac{1}{2}(h-1)\rfloor$ single bit errors can be corrected. For the above given code, up to 4 single bit errors can be detected, and up to 2 single bit errors can be corrected.

In the following subsections, Hamming coding, the well known error correcting coding paradigm, and cyclic redundancy coding, which is a very popular error detecting coding paradigm, will be presented and analyzed.

5.4.1 Hamming Coding

To correct a single bit error in a codeword, a dataword is converted into a codeword by inserting check bits at the bit positions that are powers of 2, where the leftmost bit position has number 1 [7].

Every bit position p can be rewritten as a sum of powers of 2, e.g. $5 = 2^0 + 2^2$. A checkbit at position 2^i forces the parity of the group of bits whose position contains a term 2^i, thus including itself, to be odd or even. For example, the dataword 11011011 is converted into the codeword $1_c1_c11_c1011_c1011$, where the subscript c denotes a checkbit and even parity is used.

Analysis of the Hamming Coding Method As stated before, Hamming coding can only be used to correct single bit errors. In an n-bit Hamming codeword there are $\lceil {}^2\log(n+1) \rceil$ redundant bits, or the relationship between the m data bits and the n bits of the codeword is $n = m + \lceil {}^2\log(n+1) \rceil$. To correct single bit errors, the Hamming distance of the code must be 2, or, in other words, the n-bit bit strings at Hamming distance 1 from a legal codeword are illegal. Since there are n such bit strings, there are $n+1$ n-bit bit strings dedicated to each m-bit dataword. Because there are 2^m m-bit datawords and there are 2^n n-bit bit strings, it is necessary that $2^n \geq (n+1)2^m$. Thus, the Hamming coding method achieves the lower bound.

Now, assume that a single bit error occurred. The check bits at the positions that occur as a term of the position of the corrupted bit disagree with the parity. For instance, the corruption of the bit at position 5 results in incorrect checkbits at positions 1 and 4. It can easily be seen that the sum of the positions of the incorrect check bits equals the position of the corrupted bit.

5.4.2 Cyclic Redundancy Coding

An n-bit dataword can be regarded as the list of coefficients, where the coefficients are 0 or 1, of a polynomial $M(x)$ with n terms, ranging from x^{n-1} to x^0. The basic idea of cyclic redundancy coding is to append a checksum to the end of the dataword, such that the polynomial $C(x)$ represented by the checksummed dataword is divisible, using modulo 2 arithmetic, by a generator polynomial G(x).

Let g be the degree of $G(x)$. The algorithm for computing the checksummed dataword consists of three steps:

1. Append g zero bits to the end of the dataword, resulting in a bit string of $n+g$ bits which represents the polynomial $x^gM(x)$.
2. Divide the bit string from step 1. by the generator polynomial $G(x)$ using modulo 2 division. This can easily be implemented in hardware, i.e. by repeatedly shifting and exclusive oring. The remainder is a bit string consisting of at most g bits.
3. Subtract the remainder generated in step 2. from the bit string of step 1. using modulo 2 subtraction. Again, this can easily be implemented in hardware, i.e. by

exclusive oring. The result is the checksummed dataword which is divisible by the generator.

Consider the dataword 11010 and the generator 101. Step 1 produces the bit string 1101000. Step 2 yields the remainder 01. Subtracting 01 from 1101000 results in 1101001 being transmitted.

Analysis of the Cyclic Redundancy Coding Method Suppose that instead of a bit string representing $C(x)$, a bit string representing $C(x)+E(x)$ is received, where $E(x)$ is the error polynomial. $E(x)$ has the same degree as $C(x)$ and a coefficient equaling 1 means that the corresponding bit is inverted, i.e. erroneous.

In case of a single bit error $E(x) = x^i$, where i determines which bit is in error. If $G(x)$ contains more than one term, it does not divide $E(x)$ and hence it does not divide $C(x) + E(x)$. Thus, if $G(x)$ contains more than one term, a single bit error is always detected.

In case of a double bit error $E(x) = x^i + x^j$ $(i > j)$, or $E(x) = x^j(x^{i-j} + 1)$. If we assume that $G(x)$ does not contain a factor x — which is simply satisfied if the lowest order bit of the generator is 1 — all double bit errors are detected if $G(x)$ does not divide $x^{i-j} + 1$ for any $i - j$, i.e. for $i - j$ up to the length of $C(x)$.

In case of an odd number of errors $E(x)$ contains an odd number of terms. Evaluating $E(1)$ thus yields 1 (modulo 2). Since $E(1)$ would be zero if $E(x)$ contained a factor $(x + 1)$, an odd number of errors is detected if $G(x)$ has a factor $x + 1$.

In case of a burst error of length b $E(x) = x^{i+b-1} + ... + x^i$, or $E(x) = x^i(x^{b-1} + ... + 1)$. Under the assumption that $G(x)$ does not contain a factor x and that the coefficient of its lowest order term, x^0, is 1, $G(x)$ cannot divide $E(x)$ if the degree of $G(x)$ is greater than the degree of $E(x)$, i.e. if $g > b - 1$, or $b < g + 1$. If $b = g + 1$ then $G(x)$ can only divide $E(x)$ if $E(x) = G(x)$. The most and the least significant bit of a burst are 1 by definition, so that, assuming that 0 and 1 have equal probability, the probability that a burst error of length $g + 1$ is not detected is $\frac{1}{2}^{g-1}$. If $b > g + 1$ then $G(x)$ can only divide $E(x)$ if $E(x) = A(x) \cdot G(x)$. Because the least significant bit of both $E(x)$ and $G(x)$ is 1, the least significant bit of $A(x)$ is 1. Since the degree of $A(x)$ is $b - 1 - g$, there are 2^{b-2-g} different undetectable burst errors. Because the total number of different burst errors of length b is 2^{b-2}, the probability that a burst error of length b is not detected is 2^{-g} or $\frac{1}{2}^g$. Thus, if $G(x)$ does not contain a factor x and the coefficient of x^0 is 1, the fraction of burst errors of length b that is not detected is 0 if $b < g+1$, $\frac{1}{2}^{g-1}$ if $b = g+1$ and $\frac{1}{2}^g$ if $b > g+1$.

Three generator polynomials have become international standards:

- CRC–12 $= x^{12} + x^{11} + x^3 + x^2 + x + 1$
- CRC–16 $= x^{16} + x^{15} + x^2 + 1$
- CRC–CCITT $= x^{16} + x^{12} + x^5 + 1$

5.5 Conclusion

The redundancy, on which fault tolerance is based, can be split up in replication of the component whose failure has to be tolerated, and the necessary additional overhead.

In all paradigms that were presented there is an arbiter. For consistency checks and the recovery block paradigm the arbiter determines whether the result is valid. When using duplication with comparison, the arbiter is just a comparator. In the N modular redundancy case it is a majority voter. When using coding techniques, an arbiter uses the Hamming distance to see if there have been errors, or even to determine the codeword that was corrupted.

6 Case Study I: Design of a Stable Storage

6.1 Introduction

A paradigm to construct fault tolerant systems is the atomic action concept. An action is atomic if it is executed completely or not at all. To implement atomic actions, the system state is recorded before entering an atomic action, creating a recovery point (RP) or checkpoint. This recovery point is the prior state that can be restored in case of a fault during the execution of that atomic action. This approach requires that two assumptions hold [17]:

(*RP1*) Faults are detected before the results of invalid state transformations are saved as a recovery point.

(*RP2*) Recovery points are unaffected by faults.

In this section an effort is made to design a stable storage which can be used to store and retrieve recovery points and which, in order to help satisfy (*RP2*), is not affected by faults.

6.2 Requirements and Decisions

As a first step one tries to meet the basic requirement of needing a mechanism to store and retrieve recovery points in the most reliable way possible. With regard to environmental faults (e.g. a defect power supply) memory is partitioned in volatile and non-volatile, so that the use of non-volatile memory is preferred. A method to provide for the non-volatile storage and retrieval of data is the use of *disks* (e.g. [5], [8], [10] and [19]). Therefore:

($Design\ Decision_1$) A disk will be used as a basic building block for the stable storage.

The use of such a *physical* disk results in a physical disk layer. Because a physical disk's read and write operations are usually implemented in terms of physical sectors, we require for the physical disk layer:

($Requirement_{PD}$) The physical disk (PD) layer provides an array of physical sectors with a read and a write operation on physical sectors.

(*RP2*) stipulates that the recovery points must be unaffected by faults. Although we have been very careful about the environment, we must now consider in what ways a disk may fail. For the scope of this case study, the following two fault hypotheses are of interest:

(FH_1) Due to damages of the disk surface, the contents of the physical sectors are corrupted.

and:

(FH_2) Due to faults of the disk control mechanism, the contents of a particular sector may be read or written at a wrong location.

Then, a first requirement is that it can be detected that the contents of a given physical sector are corrupted or that the contents of a wrong physical sector are returned. It is common practice to implement a cyclic redundancy check (CRC) mechanism to detect the corruption of information on a disk and to encode the number of the physical sector in the physical sector to detect that the contents are from a wrong physical sector:

($\mathit{Design\ Decision}_2$) A CRC mechanism is used to detect the corruption of information on physical sectors.

and:

($\mathit{Design\ Decision}_3$) To detect that the contents of a wrong physical sector are returned the number of the physical sector is encoded in the physical sector.

This results in a physical and a logical disk layer (see fig. 3), where the abstraction relation consists of the CRC decoding algorithm, i.e the removal of the redundant CRC bits, and the removal of the physical sector number.

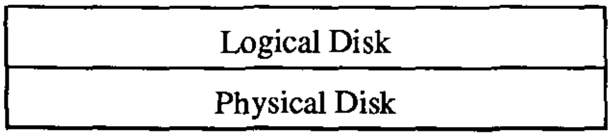

Figure 3: Physical and logical layer

For the correctness of the design it has to be assumed that:

($\mathit{Assumption}_1$) The CRC mechanism is always able to detect that a physical sector is damaged and hence that its contents are corrupted.

and[3]:

($Assumption_2$) If the control mechanism fails to write a particular sector at the correct location, all subsequent read operations for that sector are performed at the same incorrect location. Hence the address checking mechanism always detects that information is written on the wrong physical sector.

Such assumptions have to be included in this or an upper layer's fault hypothesis. Indeed, ($Assumption_1$) applies to an upper layer's fault hypothesis, whereas ($Assumption_2$) is an extension of (FH_2). Exceptions can be raised as soon as it is detected that an assumption no longer holds.

The above taken measures primarily make the read operation reliable. The write operation could be made reliable as well by means of a read after write mechanism. Since the rate of physical sectors getting damaged is low, the increased overhead is not considered worth-while:

(*Design Decision*$_4$) Only read operations are made reliable.

Then, we have the following requirement for the logical disk layer:

($Requirement_{LD}$) The logical disk (LD) layer presents an array of logical disk sectors with a reliable read and a normal write operation on logical sectors.

Since a CRC mechanism is only capable of detection and not of correction, we cannot mask the corruption: when a physical sector is damaged its contents are lost. The replacement of the CRC mechanism by a mechanism that is capable of correction is considered to be too expensive in terms of disk usage. Furthermore, a failure of the disk control mechanism causes one or more physical sectors to become inaccessible. A possible way to provide the data availability is to partition the disk in 2 sections and have each section contain the same data. However, there is little independence between the faults of physical sectors on one disk.

To guarantee the data availability the *mirrored disk* concept can be used (e.g. [5] and [19]): a second disk with identical contents is maintained, so that, in case some information can no longer be retrieved from one disk, the information is still available on the other one. That way operation is not stopped even if all physical sectors of one disk are damaged. The mirrored disk concept can be generalized to N (N $\geq$2) disks (e.g. [8] and [10]):

(*Design Decision*$_5$) To guarantee the data availability N disks with identical contents are used.

As we will see in section 6.3.3, the improvement of the performance that can be achieved outweighs the cost of the extra disks. To control the N disks we add the stable storage layer (see fig. 4), with the obvious requirement:

[3] If the control mechanism writes an update for a particular sector at a wrong location and performs a subsequent read operation for that sector at the correct location, the contents read are incorrect. Such transient faults can be detected if version numbers are administrated and encoded in the physical sector, but that is beyond the scope of this case study.

(*Requirement*$_{SS_1}$) The stable storage (SS) layer presents an array of logical sectors, the contents of which are unaffected by faults, and a reliable read and a normal write operation on logical sectors.

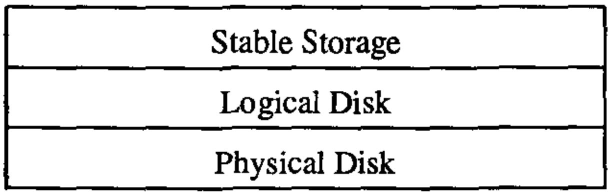

Figure 4: Layered structure of the stable storage

With the obvious assumption:

(*Assumption*$_3$) The N physical sectors with the same physical sector number are at no time during operation all damaged or inaccessible.

To guarantee that the multiple disks contain the same information, a write request must be processed on every disk:

(*Requirement*$_{SS_2}$) A write request must be processed on every disk.

If a sector is damaged it can no longer be used. Then the system degrades in the sense that it is not guaranteed that there are N copies of a logical sector at any time during operation. To avoid that degradation (*Requirement*$_{LD}$) becomes in terms of the damaged or undamaged status of the individual physical sectors:

(*Requirement*$_{LD}$') The logical disk (LD) layer presents an array of logical disk sectors that is independent of the status of the individual physical sectors. It also presents a reliable read and a normal write operation on logical sectors.

It is of course impossible to repair a damaged physical sector, but a thus far spare sector could take its place if the mapping of logical sectors to physical sectors is not fixed [3].

(*Design Decision*$_6$) The spare sector concept will be used to enable the reconfiguration of physical sectors.

Then, the abstraction relation between the physical disk layer and the logical disk layer is a combination of the CRC decoding algorithm, the removal of the physical sector number and the inverse of the mapping. Because the N physical sectors representing the same logical sector are now forming a dynamically changing set (*Assumption*$_3$) becomes:

(*Assumption*$_3$') The N physical sectors representing the same logical sector at some time during operation are not all damaged or inaccessible before the logical sector is read again.

We have to assume that:

(*Assumption*$_4$) There are always enough spare sectors.

and:

(*Assumption*$_5$) The mapping information is stored in such a way that it is not affected by faults.

This is a typical example of the case where exceptional behaviour can be specified to deal with the no longer holding of the assumption: an exception can be raised as soon as the disk has run out of spare sectors, i.e. as soon as too many physical sectors were damaged.

Just like it is not possible to repair a damaged physical sector, it is not possible to repair a failed disk control mechanism. For the sake of simplicity we decide:

(*Design Decision*$_7$) If the disk control mechanism has failed, the entire disk has failed.

If a logical disk layer signals to the stable storage layer that a disk has run out of spare sectors or that the disk control mechanism failed, that particular disk must be taken out of operation since it can no longer completely present the array of logical sectors. We have to assume that:

(*Assumption*$_6$) There are always at least 2 disks operational.

Again, exceptional behaviour can be specified to indicate that this assumption no longer holds.

The resulting logical system representation is shown in fig. 5. The functionality that controls the multiple disks is called 'serializer.' This functionality is discussed in more detail in section 6.3.3.

6.3 Description of the Layers

6.3.1 Physical Disk Layer

The format of a physical sector is denoted by *PhySec*. The physical disk layer presents an array PS of physical sectors, PS = array [0..PSNMax] of *PhySec*, plus read and write operations for physical sectors. The physical sectors are referred to by using a physical sector number PSN $\in$ [0, PSNMax].

6.3.2 Logical Disk Layer

The format of a logical sector is denoted by *LogSec*. The logical disk layer presents an array of LSNMax logical sectors, plus read and write operations for logical sectors. The logical sectors are referred to by using a logical sector number LSN $\in$ [0, LSNMax].

The use of a CRC mechanism and an address encoding mechanism requires an intermediate format, say *AdrSec*, which is the concatenation of a physical sector number (of type

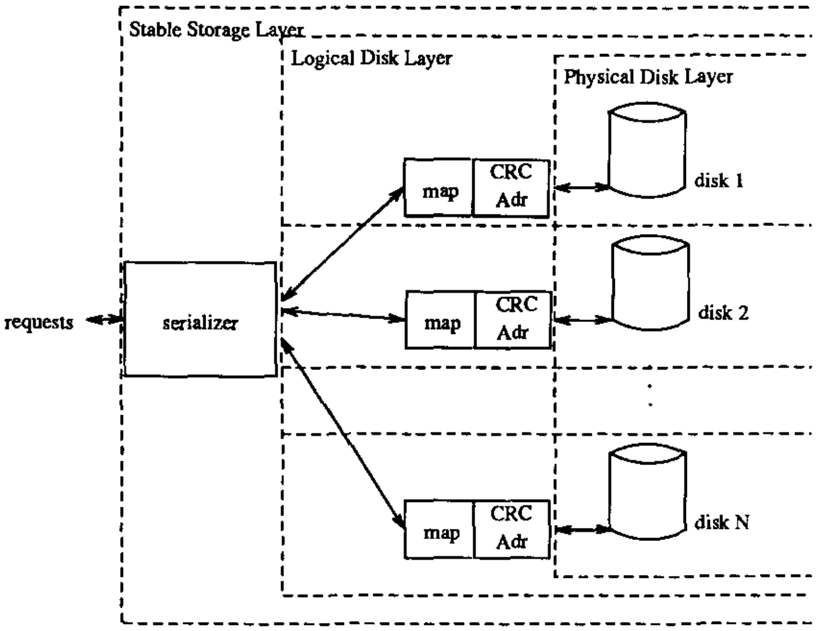

Figure 5: Logical representation of the system

PSN) and a logical sector (of type *LogSec*). Then, the use of a CRC algorithm results in an encoding function *CRCEncode*: *AdrSec* → *PhySec*, a decoding function *CRCDecode*: *PhySec* → *AdrSec*, and a check function *CRCCheck*: *PhySec* → bool. Likewise, the use of an address encoding mechanism results in an encoding function *AdrEncode*: PSN × *LogSec* → *AdrSec*, a decoding function *AdrDecode*: *AdrSec* → *LogSec*, and a check function *AdrCheck*: PSN × *AdrSec* → bool.

In case the *CRCCheck* function returns "false" the physical sector has been damaged. We implement the *spare sector* concept: when there are more physical sectors than logical sectors (PSNMax > LSNMax), the surplus creates spare sectors. If a physical sector is damaged, a spare sector can take its place. Doing so, the array of logical sectors is not affected by faults. We introduce the function *remap*: (LSN → PSN) → (LSN → PSN), to specify onto which physical sector a logical sector should be mapped and a function *lp*: LSN → PSN which, for a given logical sector number, returns the physical sector number according to the current mapping. Doing so, the abstraction relation between the physical disk layer and the logical disk layer also contains the mapping between the logical sector number and the physical sector number, besides the already mentioned CRC Decoding algorithm.

We represent the spare sectors as a set of physical sector numbers (SS = set of PSN), initially containing the numbers LSNMax+1,....,PSNMax. We do not have to represent the damaged sectors, because they are implicitly specified by *lp* and SS.

In case the *AdrCheck* function returns "false" the disk control mechanism has failed.

The above leads to the following representation invariant (RI):

$$\begin{aligned}(\mathit{RI1})\ & (\forall l : 0 \leq l \leq \mathrm{LSNMax} : \mathit{CRCCheck}(\mathrm{PS}[lp(l)]) \Leftrightarrow \\ & (\mathit{AdrCheck}(lp(l), \mathit{CRCDecode}(\mathrm{PS}[lp(l)])) \Leftrightarrow \\ & (\mathrm{LS}[l] = \mathit{AdrDecode}(\mathit{CRCDecode}(\mathrm{PS}[lp(l)])))))\end{aligned}$$

Then, the following logical disk layer read operation (LDRead), where PDRead denotes the reading of the physical sector, is quite straightforward:

```
proc LDRead (?l: LSN; !sector: LogSec) =
|[var AdrOK, CRCOK: bool;
 var as: AdrSec;
 var ps: PhySec;
 var s: PSN;
 ps := PDRead (lp(l));
 CRCOK := CRCCheck (ps);
 if ¬CRCOK → if SS ≠ ∅ → s := a member of SS;
                          SS := SS\{s};
                          remap (l,s);
                          "raise an exception to signal the stable storage layer
                          to retrieve the contents of logical sector l"
             [] SS = ∅ → "raise an exception to signal the stable storage layer
                          that the disk has run out of spare sectors"
             fi
 [] CRCOK → as := CRCDecode (ps);
            AdrOK := AdrCheck (lp(l), as);
            if ¬AdrOK → "raise an exception to signal the stable storage layer
                          that the disk control mechanism has failed"
            [] AdrOK → sector := AdrDecode (as);
            fi
 fi
]|
```

The logical disk layer write operation is even more straightforward because it does not have to be reliable:

```
proc LDWrite (?l: LSN; ?sector: LogSec) =
|[PS [lp(l)] := CRCEncode (AdrEncode (lp(l), sector));
]|
```

6.3.3 Stable Storage Layer

The stable storage layer receives read and write requests in terms of *LogSec* and LSN, which the serializer sends to the disks.

As we saw before, a write request must be processed on all the disks.

To maintain the data consistency the serializer must follow the serializability rule: it must maintain the order in which read and write requests arrive, except that *consecutive* read requests may be executed in any order so as to improve the throughput.

($Requirement_{SS_3}$) The serializer must respect the order in which read and write requests arrive.

($Design\ Decision_8$) Consecutive read requests may be processed in any order.

Typically, a disk receives far more read requests than write requests. Although the reliability of the stable storage could be improved by processing a read request on three or more disks and taking a majority vote on the results, this is not considered to be as valuable as the increase of throughput that can be reached by allowing the parallel execution of consecutive read requests [13]. Therefore, a read request is processed on only one disk at a time.

($Design\ Decision_9$) A read request is processed on only one disk at a time.

If a physical sector is discovered to be damaged, the logical disk layer signals the stable storage layer to retrieve the lost contents. Because all disks present the same array of logical sectors, the serializer only has to perform the same logical disk layer read operation on another disk. Upon reception of the logical sector contents, the serializer passes it to the requesting application and performs a logical disk layer write operation on the first disk. Recall that that first disk has already reconfigured the physical sectors.

6.4 Conclusion

In this section the design of a stable storage was presented, starting from the fault hypotheses (FH_1) and (FH_2). During the design a number of assumptions were made, which were the result of certain design decisions. As said before, such assumptions have to be included in some layer's fault hypothesis.

As said before, ($Assumption_2$) is an extension of (FH_2). The fault hypothesis for the physical disk layer becomes therefore:

$$(FH_1) \wedge (FH_2) \wedge (Assumption_2)$$

The assumptions ($Assumption_1$), ($Assumption_4$) and ($Assumption_5$) have to be included in the fault hypothesis for the logical disk layer, which therefore becomes:

$$(Assumption_1) \wedge (Assumption_4) \wedge (Assumption_5)$$

Lastly, the fault hypothesis for the stable storage layer consists of:

$$(Assumption_3\,') \wedge (Assumption_6)$$

7 Case Study II: Design of a Reliable Broadcast Protocol

7.1 Introduction

Broadcasting is a communication technique that enables the sending of a message to all destination nodes in the network simultaneously. It is used in distributed systems, for example to perform operations on distributed databases.

In the presence of faults it is possible that a broadcast message is not delivered to all destination nodes. This can lead to problems such as inconsistency. Furthermore, it is impracticable if it is not known which nodes receive the message. A broadcast protocol that guarantees the delivery of a broadcast message to all correctly functioning nodes in the network[4] is a *reliable* broadcast protocol. To be precise, a reliable broadcast protocol is a protocol that satisfies the following two requirements:

(*Requirement 1*) All correctly functioning nodes decide on the same message.

(*Requirement 2*) If the transmitter functions correctly, all correctly functioning nodes decide on the message that was broadcast by the transmitter.

These requirements are known as the 'interactive consistency' requirements [11]. For the scope of this example we are not interested in reaching agreement in the presence of Byzantine failures; we are only interested in getting a message to all correctly functioning destinations despite network faults. The various destinations do not interact to establish the validity of a message.

To make sure that all correctly functioning nodes receive the broadcast message, every such node sends an acknowledgement to the broadcasting node to acknowledge the receipt. The broadcasting node starts a timer after issuing the broadcast and retransmits the message to the nodes from which it hasn't got an acknowledgement when the timer expires or, in other words, a *time-out* occurs. Usually a maximum number of retransmissions applies.

In many distributed applications every node can start broadcasting at any moment. For the correctly functioning nodes to maintain consistency with such asynchronous broadcasts we additionally require that the messages are processed in the same order by every correctly functioning node:

(*Requirement 3*) All broadcast messages are processed in the same order by every correctly functioning node.

This means that there has to be a global ordering of messages. Usually this is achieved by first delivering the broadcast message to all correctly functioning nodes and secondly establishing the global order of the broadcast message (e.g. [4], [14]). For this case study a node buffers a received message until its order is determined; only then may a received

[4] Because one defect node would never allow the successful completion of a broadcast to 'all nodes', 'all nodes' is weakened to 'all correctly functioning nodes'.

message be processed. In [9] it is noted that if all broadcasts are issued from the same node, and if that node, which is therefore called 'sequencer,' generates a sequence number for each of those broadcasts, such a second phase is not necessary, because the sequence numbers generated by the sequencer a priori constitute the desired global ordering. We choose such a centralized broadcast protocol as a basic building block for the reliable broadcast protocol.

(*Design Decision*) A centralized broadcast protocol is the basic building block for the reliable broadcast protocol.

For this case study the nodes are interconnected by an (unreliable) broadcast network. An advantage of a broadcast network over a point-to-point network is that a broadcast network makes simultaneous distribution of a message to all other nodes *physically* possible. A disadvantage is the fact that the different acknowledgements have to be sent via one and the same channel. However, to avoid overloading the channel and the sequencer, *piggybacking* can be used: acknowledgements are attached to a message that has to be sent anyway. If there is no message to be sent within some interval of time, a separate acknowledgement is sent. Another disadvantage is that if the broadcast network goes down communication is no longer possible; if sections of broadcast networks are coupled by repeaters and/or gateways then partitions can occur. The latter case is beyond the scope of this case study.

We assume that the processors are fail-stop: they are either up, i.e. they function correctly, or they are down and when they are down they do not send any messages. This is stipulated by the following fault hypothesis:

(FH_{Node}) A node that is down does not send any message.

Doing so, Byzantine failures are excluded. Furthermore it is assumed that due to a fault of the network messages are lost; if they are not lost they are received correctly[5]:

($FH_{Network}$) Due to a fault of the network messages are lost.

7.2 A Reliable Broadcast Protocol

The centralized broadcast protocol [9] consists of two steps. If processor P wants to broadcast its i^{th} message m_{P_i}, it sends request $\mathcal{R}_{m_{P_i}}$ to sequencer S. S then broadcasts message $\mathcal{S}_{m_{P_i}}$, consisting of m_{P_i} plus a sequence number. When a node receives a broadcast message, it examines the sequence number. If the node has missed one or more messages, it will notice a gap in sequence numbers between the last message it received and the message it just received. In such a case the node sends a *negative* acknowledgement to the sequencer for every message it has missed. To speed up the completion of the broadcast, negative acknowledgements are sent as separate point-to-point messages. If there is no gap, the node can immediately pass the message to the application. As

[5] The fact that messages are lost because the buffers are full, although a big problem in practice, is left out of consideration to keep this presentation concise. It is assumed that an underlying mechanism (e.g. a CRC mechanism) rejects corrupted messages.

said before, acknowledgements are piggybacked. It is sufficient to acknowledge only the last broadcast message that was received in order; that acknowledgement acknowledges all previously received messages — which have lower sequence numbers — implicitly.

In the following sections the reliability aspects of the broadcast protocol will be considered.

7.2.1 The Broadcast Step — Protocol for a Sequencer

After S broadcasts a message $\mathcal{S}_{m_{P_i}}$, it expects an acknowledgement for $\mathcal{S}_{m_{P_i}}$ from every correctly functioning node. Therefore, S keeps a set CF of correctly functioning nodes. The sending of a broadcast is a non-blocking operation: several broadcasts may overlap. We assume that all nodes send regularly a request to the sequencer. Therefore, if S does not receive a message from a node $Q \in CF$ — either a request, a separate acknowledgement or a negative acknowledgement — before it times out, S concludes that that node is down and removes it from CF:

($Assumption_1$) A correctly functioning node, which is not the sequencer, will send a message to the sequencer before the sequencer times out.

For the scope of this case study recovery takes place at a higher level than the broadcast: the sequencer raises an exception to signal an upper layer that a node is down.

If a node that receives a broadcast message sees a gap in sequence numbers, it sends a negative acknowledgement $\mathcal{N}_s$ for every s it apparently misses. S stores a broadcast message until it has been acknowledged by every node in the set CF. Also the originating node must acknowledge the broadcast message because, although it usually does not process the message, it must know the sequence number the message got to avoid the origination of a gap.

To provide for the coming up of nodes after they were down, a node that comes up again broadcasts a "Hello network" message. Only S responds to that broadcast with an "I'm here" message containing the sequencer's identity and — because a higher layer deals with the recovery of the messages the node missed while it was down — the number of the next message the recovered node is supposed to receive. Should the recovered node get no response to that message, for instance because the sequencer is down, then it has to rebroadcast its "Hello network" message. This may go on indefinitely, because a recovered node may not do anything until it receives an "I'm here" message.

7.2.2 The Request Step — Protocol for a Non-Sequencer

In order to notify the receipt of $\mathcal{R}_{m_{P_i}}$, S could acknowledge the request, but since it is going to broadcast m_{P_i} anyway, P just has to verify that it receives $\mathcal{S}_{m_{P_i}}$ (design decision). That way we do not have to deal with the acknowledgement of the sequencer getting corrupted and hence lost.

If P does not receive $\mathcal{S}m_{P_i}$ before it times out, P will send $\mathcal{R}m_{P_i}$ to S again. There are three possibilities:

1. S did not broadcast $\mathcal{S}m_{P_i}$, because it is down.
   Then the retransmission of $\mathcal{R}m_{P_i}$ will have no effect.

2. S did not broadcast $\mathcal{S}m_{P_i}$, because it did not receive $\mathcal{R}m_{P_i}$.
   Then S is likely to receive $\mathcal{R}m_{P_i}$ now.

3. S did broadcast $\mathcal{S}m_{P_i}$, but P did not receive it.
   Then S sends P $\mathcal{S}m_{P_i}$ in a reliable point-to-point manner which means that S will retransmit $\mathcal{S}m_{P_i}$ to P until P has acknowledged it or until S decides that P is down.

Now, if we assume that a message is not lost more than some maximum number of times, say max_{lost}, then we may assume that it takes not more than a maximum number of (re)transmissions of $\mathcal{R}m_{P_i}$, say max_{tries}, before the sequencer broadcasts the message:

($Assumption_2$) A message is not lost more than max_{lost} times.

($Assumption_3$) If the sequencer functions correctly it doesn't take more than max_{tries} number of (re)transmissions of the request, before the sequencer broadcasts the message or sends it to the requesting node.

Thus, if after max_{tries} (re)transmissions of $\mathcal{R}m_{P_i}$ P still does not receive $\mathcal{S}m_{P_i}$, either as a broadcast or as a point-to-point message, it concludes that S is down. P signals an upper layer that the sequencer is down, but, because a sequencer is vital to this reliable broadcast protocol, the reliable broadcast protocol takes care of the initialization of a new sequencer.

7.2.3 The Initialization of a New Sequencer

Before a new sequencer can be initialized, first a node must be selected to become the new sequencer. If the node that detected the fault of the sequencer were to be the next sequencer, a conflict would arise if two or more nodes could simultaneously detect the fault of the sequencer. An obvious solution is to select a beforehand known node to become the new sequencer. We assume that the number of nodes is known a priori and that those nodes have successive identifiers. Hence, the identifier of the new sequencer can be obtained by incrementing the identifier of the current sequencer, modulo the number of nodes. All correctly functioning nodes know which node is currently the sequencer and can hence determine the identifier of the new sequencer. If some node detects that the sequencer is down, it sends a "You're it" message to the thus beforehand known node, which may either be up or not.

But what about the messages for which the broadcast has not been terminated? Note that when they are rebroadcast they must get the same sequence numbers because broadcast messages that are received in order are processed immediately. The natural solution is to let every node keep the messages it has received until they have been acknowledged by every correctly functioning processor. Then a node can still process the messages it received in order immediately and there is a (distributed) backup for messages for which the broadcast has not been terminated.

The problem is that though the sequencer knows what nodes function correctly and what correctly functioning nodes have acknowledged certain broadcast messages, the other nodes, and thus also the new sequencer, do not have that knowledge. Because of the asymmetry of the protocol, broadcasting such information (for instance with regular broadcast messages) does not help to reach global consensus because when a node receives a message, it cannot tell whether the other nodes have received that message also. In order to let nodes clean up their buffers the sequencer can broadcast which messages have been acknowledged by all correctly functioning nodes. However, no conclusions regarding the present state of the network of nodes can be taken from such information. This is especially the case for the set CF.

Thus, a new sequencer must initialize its own CF. If a node gets a "You're it" message, it starts by broadcasting an "I'm your new sequencer" message. Every correctly functioning node replies with the sequence number of the last message it received in order, i.e. the value it would send at that moment as acknowledgement. With its reply it also sends all outstanding requests because it is possible that the old sequencer started broadcasting them. The new sequencer expects a reply from all the nodes on the network and if it times out it sends the "I'm your new sequencer" message via reliable point-to-point communication to those nodes that haven't responded yet. Analogous to section 7.2.2. we assume that a correctly functioning node will have replied after max_{tries} (re)transmissions of the "I'm your new sequencer" message:

($Assumption_4$) It takes not more than max_{tries} (re)transmissions of the "I'm your new sequencer" message, before a correctly functioning node replies.

Thus, after up to max_{tries} (re)transmissions of the "I'm your new sequencer" message to all other nodes the set CF has been initialized. In addition, the sequencer learns for which messages the broadcast has not been terminated and also which nodes have not received those messages. If the new sequencer should miss messages itself, it can get them from nodes that did receive them. To pass the role of sequencer to the node with the most messages would require an extra round to notify all nodes about that decision and is therefore not considered to be an improvement.

To avoid problems such as congestion, node identifier dependent waits can be introduced: a node must wait some time before it may reply to the "I'm your new sequencer" message and it must also wait some time before a new request may be sent to the new sequencer.

Lastly there is the problem of the new sequencer being down or going down while the initialization is in progress. The fact that the new sequencer is down can be detected by combining a time-out mechanism w.r.t. the reception of an "I'm your new sequencer" message with a maximum of max_{tries} (re)transmissions of the "You're it" message:

($Assumption_5$) It takes not more than max_{tries} (re)transmissions of the "You're it" message, before a correctly functioning new sequencer broadcasts the "I'm your new sequencer" message or sends it to the detecting node.

The fact that the new sequencer goes down after it has broadcast an "I'm your new sequencer" message is detectable by the broadcast request mechanism as described in section 7.2.2. Note that this mechanism does not become effective until the (optional) waiting periods before new requests may be sent to the new sequencer are over. Then yet another node — which is also known beforehand — gets a "You're it" message. Note that no conflicts can arise because a recovered node must first get an "I'm here" message before it may do anything.

7.3 Conclusion

In this section a total of five assumptions were made. ($Assumption_1$), ($Assumption_3$), ($Assumption_4$) and ($Assumption_5$) can be combined with (FH_{Node}); ($Assumption_2$) can be combined with ($FH_{Network}$):

(FH'_{Node}) A node, which is not the sequencer, is down if it does not send a message to the sequencer before the sequencer times out or if it does not reply to the "I'm your new sequencer" or "You're it" message after max_{tries} (re)transmissions. The sequencer is down if it takes more than max_{tries} (re)transmissions of the request, before the requesting node receives the message. Generally, a node that is down does not send any message.

($FH'_{Network}$) Due to a fault of the network messages are lost, but a message is not lost more than max_{lost} times.

For broadcast protocols which are designed to deal with Byzantine failures (e.g. [2]), complexity is expressed in terms of the number of rounds of communication and the number of nodes that are necessary to deal with such failures. The protocol presented here does not deal with Byzantine failures, so that a comparison with the above mentioned class of protocols is not appropriate.

For protocols such as the one discussed here, complexity is usually expressed in terms of the number of messages necessary to perform a single broadcast because, for a broadcast network, point-to-point messages and broadcast messages attribute equally to the complexity. Because one request can implicitly acknowledge several messages, the derivation of such a performance model is, however, very hard. A similar problem arises in [14], where only best-case results are derived because very optimistic assumptions have to be made.

Under the assumption that every node sends a request very often and that the sequencer does not fail, a broadcast requires $1 + s + 1 + 2 * f$ messages, where f is the number of times a node does not receive a broadcast message and s is the number of times the sequencer does not receive the request; the case that the sequencer did receive the request but the sender did not see the message being broadcast is covered by f — note that the sequencer expects an acknowledgement from the requesting node.

The number of messages that are necessary to perform the initialization of a new sequencer is of itself quite incalculable, let alone the number of messages that are necessary to broadcast a message when the sequencer does fail. Simulations can be used to determine the average case performance.

Acknowledgements

The author is grateful to Jos Coenen, Peter Coesmans, Dieter Hammer, Frans Kaashoek, Peter van der Stok, Martin Wieczorek and Hanno Wupper for their remarks during the elaboration of this paper.

References

[1] A. Avižienis and J.C. Laprie, 'Dependable Computing: From Concepts to Design Diversity,' *Proceedings of the IEEE, Vol. 74, No. 5, May 1986, p. 629–638.*

[2] Ö. Babaoğlu and R. Drummond, 'Streets of Byzantium: Network Architectures for Fast Reliable Broadcasts,' *IEEE Trans. on Software Engineering, Vol. 11, No. 6, June 1985, p. 546–554.*

[3] K.H. Bates and M. TeGrotenhuis, 'Shadowing boosts System Reliability,' *Computer Design, April 1985, p. 129–137.*

[4] K.P. Birman and T.A. Joseph, 'Reliable Communication in the Presence of Failures,' *ACM Trans. on Computer Systems, Vol. 5, No. 1, February 1987, p. 47–76.*

[5] F. Cristian, 'A Rigorous Approach to Fault-Tolerant Programming,' *IEEE Trans. on Software Engineering, Vol. SE-11, No. 1, January 1985, p. 23 – 31.*

[6] P.D. Ezhilchelvan and S.K. Shrivastava, 'A Characterisation of Faults in Systems,' *Proc. IEEE Symp. on Reliability in Distributed Software and Database Systems, 1986, p. 215–222.*

[7] R.W. Hamming, 'Error Detecting and Error Correcting Codes,' *Bell Syst. Tech. Journal, Vol. 29, April 1950, p. 147–160.*

[8] E.S. Harrison and E.J. Schmitt, 'The Structure of System/88, a Fault Tolerant Computer,' *IBM Systems Journal, Vol. 26, No. 3, 1987, p. 293–318.*

[9] M.F. Kaashoek, A.S. Tanenbaum, S. Flynn Hummel and H.E. Bal, 'An efficient Reliable Broadcast Protocol,' *Operating Systems Review, Vol. 23, No. 4, October 1989, p. 5–20.*

[10] J. Katzman, 'A Fault Tolerant Computing System,' in: *D.P. Siewiorek and R.S. Swarz (eds.), 'The Theory and Practice of Reliable System Design,' Digital Press, 1982.*

[11] L. Lamport, R. Shostak and M. Pease, 'The Byzantine Generals Problem,' *ACM TOPLAS, Vol. 4, No. 3, July 1982, p. 382–401.*

[12] J.C. Laprie, 'Dependable Computing and Fault Tolerance: Concepts and Terminology,' *Proc. 15th Int. Symp. on Fault Tolerant Computing Systems, June 1985, p. 2–11.*

[13] N.S. Matloff, 'A Multiple-Disk System for Both Fault Tolerance and Improved Performance,' *IEEE Trans. on Reliability. Vol. R-36, No. 2, June 1987, p. 199–201.*

[14] P.M. Melliar-Smith, L.E. Moser and V. Agrawala, 'Broadcast Protocols for Distributed Systems,' *IEEE Trans. on Parallel and Distributed Systems, Vol. 1, No. 1, January 1990, p. 17–25.*

[15] B. Randell, P.A. Lee and P.C. Treleaven, 'Reliability Issues in Computing System Design,' *ACM Computing Surveys, Vol. 10, No. 2, June 1978, p. 123–165.*

[16] R.D. Schlichting and F.B. Schneider, 'Fail-Stop Processors: An Approach to Designing Fault-Tolerant Computing Systems,' *ACM Transaction on computer systems, Vol. 1, No. 3, p. 22–238.*

[17] F.B. Schneider, 'Abstractions for Fault-Tolerance in Distributed Systems,' *Information Processing '86 (IFIP '86), p. 727-733.*

[18] H.R. Strong and D. Dolev, 'Byzantine Agreement,' *Proc. IEEE Spring CompCon 1983, p. 77–81.*

[19] W.N. Toy, 'Fault Tolerant Design of Local ESS Processors,' *Proceedings of the IEEE, Vol. 66, No. 10, October 1978, p. 1126–1145.*

[20] 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: Design and Analysis of a Fault-Tolerant Computer for Aircraft Control,' *Proceedings of the IEEE, Vol. 66, No. 10, Oct. 1978, p. 1240–1255.*

[21] M.J. Wieczorek and J. Vytopil, 'Towards a Language and Notation for the Specification of Reliable Real-Time Systems, Part IV: Requirements and Design Specification Language,' *Technical Report no. 90-6, University of Nijmegen, Department of Informatics, Faculty of Mathematics and Informatics, March 1990.*

In this series appeared:

89/1	E.Zs.Lepoeter-Molnar	Reconstruction of a 3-D surface from its normal vectors.
89/2	R.H. Mak P.Struik	A systolic design for dynamic programming.
89/3	H.M.M. Ten Eikelder C. Hemerik	Some category theoretical properties related to a model for a polymorphic lambda-calculus.
89/4	J.Zwiers W.P. de Roever	Compositionality and modularity in process specification and design: A trace-state based approach.
89/5	Wei Chen T.Verhoeff J.T.Udding	Networks of Communicating Processes and their (De-)Composition.
89/6	T.Verhoeff	Characterizations of Delay-Insensitive Communication Protocols.
89/7	P.Struik	A systematic design of a paralell program for Dirichlet convolution.
89/8	E.H.L.Aarts A.E.Eiben K.M. van Hee	A general theory of genetic algorithms.
89/9	K.M. van Hee P.M.P. Rambags	Discrete event systems: Dynamic versus static topology.
89/10	S.Ramesh	A new efficient implementation of CSP with output guards.
89/11	S.Ramesh	Algebraic specification and implementation of infinite processes.
89/12	A.T.M.Aerts K.M. van Hee	A concise formal framework for data modeling.
89/13	A.T.M.Aerts K.M. van Hee M.W.H. Hesen	A program generator for simulated annealing problems.
89/14	H.C.Haesen	ELDA, data manipulatie taal.
89/15	J.S.C.P. van der Woude	Optimal segmentations.
89/16	A.T.M.Aerts K.M. van Hee	Towards a framework for comparing data models.
89/17	M.J. van Diepen K.M. van Hee	A formal semantics for Z and the link between Z and the relational algebra.

90/1	W.P.de Roever- H.Barringer- C.Courcoubetis-D.Gabbay R.Gerth-B.Jonsson-A.Pnueli M.Reed-J.Sifakis-J.Vytopil P.Wolper	Formal methods and tools for the development of distributed and real time systems, p. 17.
90/2	K.M. van Hee P.M.P. Rambags	Dynamic process creation in high-level Petri nets, pp. 19.
90/3	R. Gerth	Foundations of Compositional Program Refinement - safety properties - , p. 38.
90/4	A. Peeters	Decomposition of delay-insensitive circuits, p. 25.
90/5	J.A. Brzozowski J.C. Ebergen	On the delay-sensitivity of gate networks, p. 23.
90/6	A.J.J.M. Marcelis	Typed inference systems : a reference document, p. 17.
90/7	A.J.J.M. Marcelis	A logic for one-pass, one-attributed grammars, p. 14.
90/8	M.B. Josephs	Receptive Process Theory, p. 16.
90/9	A.T.M. Aerts P.M.E. De Bra K.M. van Hee	Combining the functional and the relational model, p. 15.
90/10	M.J. van Diepen K.M. van Hee	A formal semantics for Z and the link between Z and the relational algebra, p. 30. (Revised version of CSNotes 89/17).
90/11	P. America F.S. de Boer	A proof system for process creation, p. 84.
90/12	P.America F.S. de Boer	A proof theory for a sequential version of POOL, p. 110.
90/13	K.R. Apt F.S. de Boer E.R. Olderog	Proving termination of Parallel Programs, p. 7.
90/14	F.S. de Boer	A proof system for the language POOL, p. 70.
90/15	F.S. de Boer	Compositionality in the temporal logic of concurrent systems, p. 17.
90/16	F.S. de Boer C. Palamidessi	A fully abstract model for concurrent logic languages, p. p. 23.
90/17	F.S. de Boer C. Palamidessi	On the asynchronous nature of communication in logic logic languages: a fully abstract model based on sequences, p. 29.

90/18	J.Coenen E.v.d.Sluis E.v.d.Velden	Design and implementation aspects of remote procedure calls, p. 15.
90/19	M.M. de Brouwer P.A.C. Verkoulen	Two Case Studies in ExSpect, p. 24.
90/20	M.Rem	The Nature of Delay-Insensitive Computing, p.18.
90/21	K.M. van Hee P.A.C. Verkoulen	Data, Process and Behaviour Modelling in an integrated specification framework, p. 37.
91/01	D. Alstein	Dynamic Reconfiguration in Distributed Hard Real-Time Systems, p. 14.
91/02	R.P. Nederpelt H.C.M. de Swart	Implication. A survey of the different logical analyses "if...,then...", p. 26.
91/03	J.P. Katoen L.A.M. Schoenmakers	Parallel Programs for the Recognition of *P*-invariant Segments, p. 16.
91/04	E. v.d. Sluis A.F. v.d. Stappen	Performance Analysis of VLSI Programs, p. 31.
91/05	D. de Reus	An Implementation Model for GOOD, p. 18.
91/06	K.M. van Hee	SPECIFICATIEMETHODEN, een overzicht, p. 20.
91/07	E.Poll	CPO-models for second order lambda calculus with recursive types and subtyping, p.
91/08	H. Schepers	Terminology and Paradigms for Fault Tolerance, p. 25.
91/09	W.M.P.v.d.Aalst	Interval Timed Petri Nets and their analysis, p.53.
91/10	R.C.Backhouse P.J. de Bruin P. Hoogendijk G. Malcolm E. Voermans J. v.d. Woude	POLYNOMIAL RELATORS, p. 52.

18