Chapter 14

Software-Fault Tolerance

At the beginning of the Delta-4 project, a design assumption was made to the effect that only physical faults were to be taken into account when providing fault-tolerant mechanisms: the possibility of design faults could therefore be neglected. In the subsequent years, recognizing that software design faults are becoming a major source of system service disruption, it was decided to study how to provide the Delta-4 architecture with specific provisions to deal with this kind of faults.

Since the term software-fault tolerance may assume different meanings, let us also say that we intend here to deal with ways to tolerate design faults in software. The tolerance of hardware design faults will only be considered as a side issue. The general consideration applies that the effectiveness of fault-tolerance techniques is not usually limited to a precisely defined class of faults, and hardware design faults, software bugs and transient hardware faults often lead to similar behaviour, as discussed in section 6.4.2.

It should be noticed that design faults may be present in Delta-4 hardware, operating system software, and applications. Application-level software-fault tolerance can help against all three kinds of errors (assume an operating system error that causes messages to be delivered in the wrong order: an application will often be able to recognise this, based on the expected contents of the messages), but is mainly directed against errors resulting from faults in the application itself.

This chapter, after briefly recalling the main techniques presented in the literature to tolerate software design faults, focusses on the problem of applying some of these techniques in the Delta-4 architecture. Support mechanisms and structuring concepts are presented. It should be pointed out that the solutions shown below are still in the specification phase — no implementation has yet been carried out.

14.1. A Brief History — the State of the Art

14.1.1. The Problem of Software Dependability

Software engineering has offered many improvements in the way software is produced, but no radical solution for producing dependable software is in sight [Brooks 1987]. Formal proofs of correctness are still unfeasible for many real-world products, and suffer from some inherent limitations (with respect to specification faults and/or ambiguities, for instance). Testing suffers from basic limitations, best stated as the fact that it can discover faults, but not prove absence of faults (sometimes not even upper bounds on the probabilities of failures caused by residual faults [Hamlet 1987, Miller 1986b]). Development methodologies and tools have improved, but none are known to avoid software faults.

From a management point of view, increasing the expense in fault avoidance methods beyond some threshold yields diminishing returns, because of technological limits, increased overhead and human communication problems.

Since creating perfect software components seems impossible anyway, to increase reliability, and in general dependability, of current, complex computing systems, we need strategies for coping with software design faults and techniques suitable to tolerate their effects.

According to [Gray 1986], for instance, once disc storage is duplicated, software faults become by far the most common cause of errors (except operator errors) in a transaction-oriented computer system. According to Giloth and Prantzen [Giloth and Prantzen 1983], about one fourth of the down time in the 4ESS electronic switching system was due to software problems. So was the January 1990 nationwide outage in the AT&T telecommunication network.

Software-fault tolerance, besides improving reliability, for a given cost, is thought to have the potential of: i) decreasing the cost of operational testing of the final product, through on-line error detection; ii) thus, shortening the teething problems of new products; iii) obtaining reliability levels that would be impossible by other means, regardless of the accepted cost, for applications where the cost of failure is very high.

Of course, software-fault tolerance techniques, given their cost, can only be cost-effective if they use reasonably reliable individual modules in their redundant configuration.

14.1.2. Software-Fault Tolerance Methods

Early proposals for software-fault tolerance began to appear in the mid 70s [Avizienis 1975, Horning et al. 1974, Kopetz 1974, Randell 1975, Shaw 1976, Yau and Cheung 1975]. By the late 70s, some methodological proposals had been defined. We will quote here:

1) *General concepts* about error detection and treatment (independent of the origin of errors): executable assertions, exception handling and such. These are proposals about language constructs, and corresponding program-structuring criteria, that would make it easier to: i) state conditions that the state of the computation must satisfy at given points in the execution, and ii) describe the actions that must be taken if an assertion is not satisfied [Cristian 1987].
2) Structured software-fault tolerance: *Specific proposals* for structuring the addition of redundant code in programs in a simple and coherent way:
   2a) *Recovery blocks* [Randell 1975], where a block[1] of code contains, besides the "primary" routine for the specified computation, i) an acceptance test to be performed on its results and ii) back-up ("alternate") routines, functionally equivalent to the primary, to be invoked, on the state of the data prior to the execution of the primary, if the results of the primary fail the acceptance test. So, the recovery block concept is based on error-detection through executable assertions plus backward error recovery; only if no alternate produces an acceptable result the block fails, and the resulting exception must be handled at a higher level.
   2b) *N-version programming* [Chen and Avizienis 1978], or "multiple version software": replicated, diverse, functionally equivalent software modules, referred to as "variants", are executed concurrently (e.g., on the replicated processors of an N-modular redundant system), and their results are compared

1 "Block" has the same sense here as in "block-structured languages".

and voted to mask errors and possibly correct the internal states of the variants themselves: it is a form of masking redundancy, with forward error recovery.

2c) *N-Self-Checking Programming* [Laprie et al. 1987]: N functionally equivalent self-checking components are executed concurrently; one is considered as being acting and delivers the results, the other self-checking components are considered as hot spares. A self-checking component can be either the association of an acceptance test to a variant or the association of two variants with a comparison algorithm.

Subsequent research has elaborated on these basic concepts; its topics may be divided as follows:

- *Architecture of fault-tolerant software.* A number of proposals have appeared, generally originating from one of the three concepts of recovery blocks, multiple-version software, and assertion checking, and often combining aspects of these techniques [Anderson 1986, Avizienis and Kelly 1984, Hecht 1976, Kim 1984, Kim and Ramamoorthy 1976, Strigini and Avizienis 1985] deal with how the component elements of those techniques can be subsumed in a more global vision of design possibilities. A series of contributions from the University of Newcastle (see [Randell 1987]) concentrates on the structuring of complex systems for fault-tolerance: nesting of recovery blocks, nesting of generic components and exception propagation, structuring fault-tolerance in distributed systems. An experimental supervisory system for running multiple-version software on Unix-like systems, called "Dedix" (Design Diversity Experiment), was developed at UCLA [Avizienis et al. 1985].
- *Ad hoc techniques for tolerating special classes of faults.* We shall quote the area of "robust data structures" [Taylor and Black 1985], dealing with techniques for protecting, through redundancy in their representation in storage, the "structural" information of data structures (e.g., the topology of a graph structure, regardless of the information located at the nodes).
- *Evaluation of software-fault tolerance strategies.* The effectiveness of software-fault tolerance techniques, as an improvement in the operational reliability or safety of software, has been studied along two main directions: experimental measurement and analytical estimation. Only very partial conclusions have been be reached so far.

Documentation, and additional references, about most of the experimental work and industrial applications can be found in [Littlewood 1987, Strigini 1990, Voges 1988].

14.1.3. Software-Fault Tolerance in Practice

14.1.3.1. Fully Redundant Software. Fully redundant software is not yet widely used and is at present limited to highly critical applications. Nonetheless, there are a number of interesting applications. Most of these are documented in the book [Voges 1988].

These examples are in aerospace, railway and atomic energy applications. All these areas are influenced by the presence of regulatory agencies whose approval is needed for the operation of products. *Fail-safeness* is usually required (the computer system can stop, provided it leaves the controlled system in a safe state), sometimes with continuous fault-tolerant operation as a long-term objective, but in avionics, as computers take on flight-critical tasks, the basic requirement is becoming *continuous fault-tolerant* operation.

Nuclear applications have been only experimental so far.

Avionic multiple-version software systems have been operational for several years. We can quote the Airbus A-310 flap and slat control, the autopilot for the Boeing 737/300, the Airbus A-320 fly-by-wire flight control system, and the Space Shuttle Flight Control System.

There are examples of dual version (fail-safe) systems in railway applications in Sweden [Voges 1988] and in Austria [Theuretzbacher 1986].

14.1.3.2. Other Software-Fault Tolerance Techniques. Two outstanding examples of systems able to cope with software faults without massive (diverse) software redundancy are the Electronic Switching Systems of the Bell System, in its several versions, and the Tandem systems. Both systems were primarily designed to tolerate hardware faults by an error detection and backward recovery scheme; but including in programs means to detect erroneous behaviour proved to be quite effective to handle software faults as well.

A detailed description of the techniques used in the No.5 ESS system to detect software errors is given in [Haugk et al. 1985]. The software system in the No.5 ESS is structured in functional components that communicate by messages and operate with duplicated units in an active-standby configuration; at run time, relatively short lived processes are created to perform specific tasks. Error detection is based on in-line defensive checks to ensure quick error detection, audit programs to verify consistency of data using data redundancy, special processes to detect program loops, scheduling problems, unavailability of critical resources, time-out on watchdog timers and such. For each detection mechanism, a specific error recovery procedure is designed.

In the Tandem system, the main features that assist in tolerating software faults [Gray 1986] are:

- software modularity (the structuring of application software into processes that interact via messages reduces error propagation);
- fault containment through fail-fast software modules (processes are made "fail-fast" by defensive programming, i.e. they run many consistency or reasonableness checks on their inputs, intermediate results and outputs, and abort themselves if an error is detected);
- atomic transaction mechanisms to establish natural checkpoint-rollback points, masking the actions of the aborted process to the rest of the world;
- process pair mechanisms to allow easy repetition of failed computations (for processes that cannot use the atomic transaction mechanism, automatic checkpointing is provided).

14.1.4. Effectiveness

All experiments to date have shown that software-fault tolerance improves the reliability of software to which it was applied, but it is difficult to evaluate its effectiveness in quantitative terms.

Several analytical estimations of the effectiveness of structured software-fault tolerance techniques (and sometimes of performance) have been published, differing in the models and analytical tools used and in the assumptions made. Unfortunately, they all depend, in order to estimate reliability, on the probability of errors common to redundant components (replicated modules, or the producer of a result and its verifier). This information must be obtained experimentally, but we are still far from being able to determine it with any confidence. Much experimental work is aimed at selecting development rules that make that probability as small as possible, but the comparison between alternative rules is done mostly by qualitative observations on the anecdotal history of each development.

The experience in aerospace, railway and atomic energy applications cannot be used to measure the effectiveness of software-fault tolerance techniques: the experimental sample is small, little information is published, and the operational record of the software usually shows few or no serious errors. Such software is built to very high standards, and the desired reliability level is so high that it could hardly be measured experimentally with any reasonable degree of confidence. These techniques are used because of an *a priori* belief that they help, although in a non-quantifiable way, to reduce the probability of failures whose cost would be extremely high.

Even for the 5ESS system and Tandem no specific figures for the effectiveness of software-fault tolerance techniques are known; the high reliability and availability demonstrated in the operation of these systems is an indication of success of the mechanisms used.

14.2. Software-Fault Tolerance for Delta-4

14.2.1. Fields of Application

We must first recall that the intended product of the Delta-4 project is not a computing system that can be used in life-critical applications, but computing systems that will provide a significant increase in reliability and availability compared to those currently used.

There is a common misconception that structured software-fault tolerance is worth applying only for *life-critical* applications, where the supposedly very high costs of software-fault tolerance are acceptable. Actually, software-fault tolerance can be applied in different degrees depending on the applications, to match the cost of the redundancy to the expected benefits. Moreover, cases can be found where even expensive forms of software-fault tolerance are justified.

It is probably safe to assume that no life-critical safety functions will be programmed on Delta-4. Safety systems for industrial plants must usually be built independently of the main control systems, and they must be very simple and easy to validate. Hence, even in cases where they are implemented in software, this probably would not be run on a Delta-4 system.

However, in many applications that are not deemed life-critical, large amounts of money are staked on the dependability of the software. In some cases, the software can directly cause damage to valuable items (either information or physical objects); in others, the software may cause unavailability of a computer system, which in turn causes the financial loss.

Some examples of intended uses of Delta-4 systems and related risks are listed below.

- *High level planning and management functions* (in industrial applications). This means that faults may cause large inefficiencies in the allocation of resources and disruption in the production process. The distributed computer system is also bound to be involved in plant-wide emergency contingency plans, and so indirectly have life-critical functions, although the operation of individual machines in the plant is safeguarded by reliable safety systems.
- *Control of individual peripherals*. A chemical reactor can be induced to ruin a batch of an expensive or profitable product. Robots used in material handling or processing will be prevented from running into each other or rolling over people by safety interlocks, but they can still damage the objects they handle, if given wrong orders or information.
- *Electronic funds transfer and banking applications*. There have been several recent episodes of huge losses to banks and security brokers due to computer error or unavailability [Risks].

14.2.2. General Criteria

Software-fault tolerance support should be provided in Delta-4 according to the following criteria:

1) Consider the software component as the basic unit to which software-fault tolerance techniques may be applied (following a general Delta-4 rule).
2) Exploit basic Delta-4 fault tolerance mechanisms, e.g., atomic multicast, checkpointing, etc.
3) Offer a coherent set of support mechanisms, enabling the implementation of a wide spectrum of software-fault tolerance techniques. The user is left the choice of the technique best suited to the application, by trading cost against dependability improvement, as well as considering real-time requirements and other factors, e.g., error coverage. Simple techniques, featuring the notification of an internal error to abort the computation, up to structured solutions like N-Version-Programming and N-Self-Checking Programming can be implemented, as well as new paradigms, not bound to "classical" proposals.

   In fact, in some critical applications, there is a well-defined set of catastrophic failure modes that will invite the use of structured software-fault tolerance protections as an obvious and economical precaution. In others, audit and cross-checking procedures will play the some rôle. In some applications, the complexity of the system may make it desirable to apply a "blanket" approach, such as global replication, to reduce the residual failure probability.
4) Allow software-fault tolerance and hardware-fault tolerance to be configured independently. In the general case, however, an integral approach to fault tolerance has to be followed, to minimize the redundancy (and cost), to extend the coverage, and to facilitate fault diagnosis.
5) Obtain ease of use. An important feature for any reliability-increasing technique is the possibility of applying it without changing the representation of the system at some level of abstraction. This allows the application programmer to concentrate on the functional specifications of the product, and another designer to add redundancy without complicating the program.

   By contrast, many of the possible ways of providing software-fault tolerance require the application developer to think up tests based on the semantics of each individual module, to worry about whether the use of diverse modules might cause problems, etc.: in short, the application of redundancy complicates the development process.

   As this lack of transparency is a necessary consequence of the willingness to check for deviations from the specifications of the individual applications, as opposed to the specifications of the machine supporting the applications, it is desirable that at least the individual application programmers be shielded from the need to take care of software-fault tolerance at the same time as they program the functional parts of the application. The division of the software into work jobs assigned to different programmers should take into account the desirability of diversity, and configuration tools should make the task of assembling the redundant software parts relatively easy and error free. For instance, it must be possible for a configuration manager to vary the number of software variants in a redundant module without interfering with the individual variants.

In a similar way as for hardware-fault tolerance, strategies for tolerating software faults in Delta-4 can be developed for:

a) use of software components that are either self-checking or fail-uncontrolled; in the latter case error detection is accomplished through comparison with independently obtained results, or by separate components (consumers of results, auditors).

b) use of deterministic or non-deterministic replicas. Non-deterministic programs must be self-checking: non-deterministic non-self-checking programs can be made tolerant of software faults by ad hoc techniques that hardly suggest mechanisms of general use.

c) use of backward or forward recovery of faulty components.

The choice of how the basic support mechanisms are combined into a coherent strategy to cope with software faults is left to the application designer. The next section presents some typical combination schemes, that are thought to have wide applicability, and could be incorporated in tools for assisting the designer in this task.

In the presented solutions it is assumed that the hosts are *fail-uncontrolled.*

14.3. Support Mechanisms and Features

A number of specific mechanisms should be provided to support software-fault tolerance in Delta-4 in the several activities that it involves, namely error detection, error processing, recovery of damaged state in faulty variants; they are described in the following subsections.

14.3.1. Error Detection

Errors in a computation can be detected essentially in one of two ways: by checking that its results have some properties required by its specification (*acceptance test* or *executable assertion*), or by comparing them with independent results (here *independence* refers to the computation failure modes). These techniques are dual to self-checking and replicate-and-compare circuits, respectively, for hardware-fault tolerance (an acceptance test can be included in the same software component together with the code to be checked).

14.3.1.1. Self-Checking Software Components. Detection of internal faults during the execution of a program can be carried out by a variety of techniques, from simple data consistency checks (e.g., non-null pointers, range checks), computation of the inverse function with respect to the main program computation, up to, conceivably, execution of a different implementation of the specifications and comparison of results (inside the component).

Self-checking software components, when an error is detected that is not recoverable inside the component[2], are allowed to behave in two ways:

a) simply omit sending due messages (like fail-silent hardware) and abort;

b) explicitly notify the error occurrence to their rep_entities (cf. section 6.4.4).

The second alternative allows the system to react sooner, since there is no need to wait for a time-out to expire; it also offers some benefits for error processing, as shown later.

A self-checking software component is characterized by its use of error-detection mechanisms and by appropriate reactions to the detection of errors.

Error detection mechanisms include:

- tests explicitly programmed by the application programmer;

---

2 Backward recovery *inside* a component can be done transparently to the rest of the system, but must satisfy the constraint of replica determinism and must not have side-effects outside the component.

- mechanisms existing in the virtual machine supporting the application component: hardware (divide-by-zero trap), operating system software (overflow of system data structures, illegal system calls), language support (array overflow checks).

The latter error detection mechanisms, in general-purpose computers, are typically configured to abort the application process, in a way transparent to the application programmer. However, many systems give the application programmer some limited control of the reactions to exceptions raised by the support virtual machine. Such control may be offered by the operating system (e.g., the UNIX *signal* system call) or as part of a language (programmed exception handling, as in Ada, for example).

For Delta-4, it is desirable that, to the largest degree allowed by the native LEX used, exceptions be made available to Deltase and/or the application program. Deltase must provide a default reaction, which must not only abort the erroneous component but also notify Delta-4 system administration.

The application programmer can instead customize the reaction to a detected error based on his knowledge of the context where it arose. Depending on the severity of the fault, two different actions can be performed:

a) If the software component cannot be (internally) recovered to an operational state, an exception-notifying message is generated; a subsequent system recovery and/or reconfiguration is required.

b) If a single output result is recognised as erroneous, but the internal state of the component still allows continued operation (e.g., an internal recovery action has been successful), the expected output message is generated, with the special format *<ABSTENTION>*. This allows the normal message flow to be maintained, avoiding the need for system recovery. Several *<ABSTENTION>* messages can be sent out by a software component during an execution.

14.3.1.2. Error Detection by Comparison. In the Delta-4 fail-uncontrolled host paradigm, detection of hardware-induced errors is accomplished by comparing the results produced by two or more identical software components, running on separate hosts. This method relies on the assumption that faults in physically autonomous electronic circuits exhibit statistical independence: in fact, the assumption closely matches the actual behaviour, with limited exceptions (e.g., upon occurrence of large electromagnetic disturbances).

To detect errors caused by software bugs it is necessary to compare results generated by program modules, which are ideally free from common design faults. The assumption of statistically independent failures in redundant software modules has sometimes been used. Although convenient for mathematical analysis, this hypothesis represents neither reality, according to experiments, nor an ideal goal (an ideal goal would be zero probability of common error, i.e., negative correlation). It has been shown in the literature [Eckhard and Lee 1985, Littlewood and Miller 1987a, Littlewood and Miller 1987b] that programs developed "independently" (in the sense that they are extracted randomly and independently from a population of possible programs) do fail independently on individual inputs, but may exhibit failure correlation, on a population of inputs, if some inputs are more likely to induce failures than others. Anyway, there is evidence that forcing methodological diversity in development is generally better or no worse than not forcing it.

When error detection is based on comparison of results, the problem arises that diversely implemented replicas of a software component (variants) will often produce different correct results; such components will be referred to as *divergent variants*. For instance, diverse implementations of real-numbers arithmetic functions will usually produce results that differ in

the least significant digits (specifications that allow multiple, very different correct solutions are not considered here).

For this reason, the possibility of inexact comparison is a requirement for practical systems. Otherwise, many redundant applications would always fail on a "no agreement" situation.

Since the result comparison is inherently connected to find out the "good" result from the set produced by several variants, the recommendations for its use in Delta-4 will be discussed below, together with the latter problem.

14.3.2. Error Processing

Several error processing strategies, allowing different levels of dependability, must be available.

The simplest way of processing an error condition, upon detection, is to abort the affected computation and discard the faulty modules. This solution is appropriate, of course, for applications having almost no requisite of dependability. The only support needed for such an action is the capability of self-aborting by application programs, which is ordinarily supplied by LEXes. However, such an occurrence has to be signalled to system administration; this can be implemented by intercepting the LEX's abort call.

As the next step in dependability levels, Delta-4 supports a range of applications that do not have stringent time constraints and allow the use of backward error recovery techniques.

To support applications with more demanding requirements in terms of real-time response, the voting mechanism (introduced in section 14.3.1.2 for the purpose of error detection and error masking) can be effectively exploited to determine a "correct" value out of N results obtained by diverse, concurrent, redundant variants.

While in the backward recovery technique both the correct result and a consistent computation state are obtained by the single recovery action, only the correct value can be obtained by voting. The state recovery of failed components is (optionally) obtained by ad–hoc means.

Backward error recovery and the problem of choosing a "good" result (*adjudication)* are discussed below.

14.3.2.1. Backward Recovery. Error-handling techniques based on backward error recovery can be reduced to a common scheme: the state of the computation is saved from time to time (checkpointing). In case of a fault detected by error detection mechanisms, the component execution is stopped, and a backward recovery action is taken, by restoring the state saved in the last checkpoint and the same computations are repeated (using the *same* code). For example, consider the case where an exception is raised by the operating system, due to some internal problem (say, overflow of the process table). The programmer can invoke the rollback primitive, since he may expect the problem to clear itself in short time. Notice that the implementation of the rollback primitive must i) be such as not to be likely to worsen the problems that may have caused the original exception (in this example, must not require new space in the overflowing table, but wait for the overflow to be resolved), and ii) allow a clean abort of the application component, with proper notification to system administration, if a roll-back does not succeed.

This technique recovers only those faults (whether in hardware or in software), whose effects are expected not to last until the subsequent re-execution. To recover from solid hardware faults, the checkpoint data need to be stored on a redundant host (as in the Delta-4 passive replica model). There is a class of software faults that cause errors having a "transient"

appearance: quoting from [Gray 1986], the residual bugs in good software are often "Heisenbugs" rather than "Bohrbugs" (see section 6.4.2).

Backward recovery techniques for software faults without *design diversity* are recommended on mature software only (where Heisenbugs only are likely to be left). The software components shall be equipped with extensive run-time error detection mechanisms, based on: defensive programming techniques; consistency and reasonableness checks on inputs, intermediate results and outputs; program structuring and run-time organization able to confine the effects of errors (such as processes communicating through messages only, object-oriented systems; e.g., Deltase programs).

These techniques are an extension of the passive replica model for hardware-fault tolerance (see section §6.6). However, unlike the passive replica model, the checkpoint data can be stored in the same node where the software component is running, if application software-caused errors are deemed to be the principal source of system problems.

Another way of implementing backward recovery in Delta-4 is to use a transactional model of computation. Transactions implement recoverable atomic actions at the application level. By forcing a transaction abort in case of error, the component itself, as well any other component involved in the transaction, is brought back to a consistent state.

The processing of software-originated errors by backward recovery, possibly through the structuring of applications in transactions, needs the same support as the processing of hardware-caused errors as shown in section 6.6, on passive replication. However, the process of rolling back to a previously saved checkpoint can be triggered by an autonomous action of the software component itself (which, of course, will act upon internal detection of an error). Furthermore, there is no need to clone the component on a different node, with savings on hardware redundancy requirements.

Handling also "Bohrbugs" requires the execution of diverse variants; this is accomplished by the recovery block structuring concept. No new architectural support, other than those well known in the literature, is needed in Delta-4 when using the active replica protocol for hardware-fault tolerance. The case of passive replicas on fail-silent hosts will be shortly discussed in the remarks concluding this chapter.

14.3.2.2. The Adjudication Problem. Structured software-fault tolerance techniques, such as N-Variant Programming and N-Self-Checking Programming, can be considered instances of a general model [Anderson 1986], based on the use of diverse variants, whose outputs are used by an *adjudicator* sub-component. The adjudicator [Di Giandomenico and Strigini 1990], or generalized decision function [Avizienis and Kelly 1984], or collator [Cooper 1984], must produce the most probably correct result.

Several adjudication functions have been presented in the literature. Some of them use only the normal outputs of the replicas (e.g., exact (bit-by-bit) majority voting; median adjudication function; mean adjudication function). Others use additional information, such as: results of acceptance tests, reasonableness tests, maximum distance between consecutive results, etc.

A comprehensive discussion can be found in [Di Giandomenico and Strigini 1990]. Summarizing, the adjudication problem, is based on algorithms much more complicated than exact (bit-by-bit) voting as provided by MCS. This implies that the definition of the adjudication function may often be application-specific, making it difficult to create a standard adjudicator.

As a consequence of the above considerations, a number of support mechanisms could be provided in Delta-4 to allow the application programmer to include appropriate adjudication procedures in standard Delta-4 components.

1) MCS-level adjudicators: used only when identical results are specified for the variants, and/or with self-checking variants capable of sending abstention messages;
2) Deltase-level support: the user is given means to embody his own voting procedures into Deltase run-time support code, by using templates available in system libraries;
3) Application-level support: a software component generation tool able to automatically expand a software component into its redundant modules plus the adjudicator module is provided. A library of adjudicators should then be set up, at least for the basic objects available in a given system. To simplify the process, a generic template of adjudicator objects could be designed; a specific adjudicator would be built by linking a user-supplied voting routine to the template.

Self-checking software components require some discussion. In a typical configuration, two such components run concurrently as a replicated Delta-4 component. In the absence of faults, the results of both of them are correct: it is sufficient to choose one of them, e.g., on the basis of a record of past errors. When a fault occurs, the affected component either aborts, or sends an *<ABSTENTION>* message. In either case, the task of choosing the (unique!) correct value is indeed trivial, disregarding the different time performance. However, when an integrated approach to hardware and software fault handling is taken, the adjudication task may present subtle problems, as will be shown by an example in section §14.3.2.6. Anyway, the decision function can be expressed in a simple table form, where the table contents depends only on the system hardware and software configuration; this allows the implementation of the adjudicator at the MCS level.

14.3.2.3. Application-Level Adjudicator. A possible structure for a software-fault tolerant application in Delta-4 is shown in figure 1.

All the modules shown are standard Deltase objects, whose corresponding software components are produced using the Deltase software component generation system, Deltase/GEN. Both client and server replicated objects are presented to the rest of the system as non-replicated (but software fault-free!) ones. This is obtained by hiding replicated objects behind adjudicator objects (CA.adj, CB.adj, S.adj in figure 1). The adjudicator objects should be protected from hardware faults by means of standard, transparent Delta-4 replication.

The ordinary client-server interaction between client *CA* and server *S* through, say, the RSR primitive, sketched as:

$$CA \xrightarrow{\text{RSR}} S$$

becomes:

$$\begin{array}{l} CA.1 \xrightarrow{\text{RSR}} CA.adj \\ CA.2 \xrightarrow{\text{RSR}} CA.adj \\ CA.3 \xrightarrow{\text{RSR}} CA.adj \end{array}$$

CA.adj adjudges requests

$$CA.adj \xrightarrow{\text{RSR}} S.adj,$$

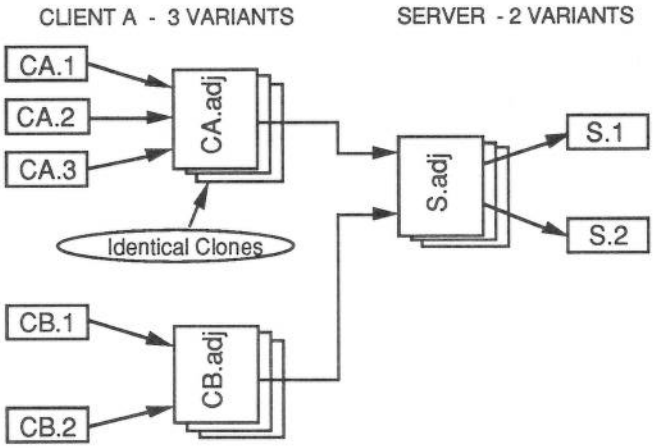
$$\left\{ \begin{array}{l} S.adj \xrightarrow{\text{RSR}} S.1 \\ S.adj \xrightarrow{\text{RSR}} S.2 \end{array} \right\},$$

S.adj adjudges replies

S.adj replies to *CA.adj*

$$\left\{ \begin{array}{l} \text{CA.adj replies to CA.1} \\ \text{CA.adj replies to CA.2} \\ \text{CA.adj replies to CA.3} \end{array} \right\}$$

(i) A DELTASE CLIENT-SERVER CONFIGURATION



(ii) REDUNDANT CLIENTS AND SERVERS

Fig. 1 - Separate Client and Server Adjudicators

Each server is accessed through a single adjudicator object (*S.adj* in figure 1), independent of the number of clients. The adjudicator should issue replicated service requests, avoiding the interleaving of requests coming from different clients. For this purpose there is no necessity of using atomic multicast; for better performance *S.adj* and *CA.adj* could use simple bi-point transmission to send messages to *S.1*, *S.2* and to *CA.1*, *CA.2*, *CA.3*.

In this solution, there is no need for special architectural support. However, a software component generation subsystem able to automatically expand a software component into its redundant modules plus the adjudicator could be very helpful to the application programmer. A library of adjudicator objects should be set up, at least for the basic objects available in a given system. Further, a generic template of adjudicator objects could be designed; a specific adjudicator would be built by linking a user-supplied adjudication function to the template.

No special restriction on software modules is assumed, i.e., divergent, non-self-checking components should be accommodated. Inexact voting procedures, which heavily depend on the specific application, can be provided.

The above configuration suffers some drawbacks:

1) The performance penalty imposed by the two added message levels between client and server.

2) The fairly large number of objects, considering that (as shown in the figure) the adjudicators need to be replicated to mask hardware faults.
3) The facilities already provided by the Delta-4 architecture, notably the atomic multicast communication protocol, are not exploited.

14.3.2.4. Adjudicator as a Unique Deltase Object. In this second solution, the adjudication of replicated client requests, as well as that of replies from replicated servers, are accomplished by a special Deltase object *S** (see figure 2). *S** is interposed between clients and servers by the Deltase subsystem for software component generation and installation (Deltase/GEN). A different adjudicator object *S** is required for each client-server pair.

Synchronization among replicas and error reporting are also implemented in this object.

*S** has a number of identical interfaces to the replicated clients; the number *N* of such interfaces, i.e., clients, actually linked to *S** can be dynamically changed. In a similar way, multiple interfaces to the replicated servers are provided.

For protection against hardware errors, *S** should be specified as a standard Delta-4 replicated object obeying the validate-before-propagate paradigm.

A template of a generic adjudicator object, including the communication interfaces, can be set up in a library, leaving to the application programmer the task of providing the procedures implementing specific adjudication algorithms. In the software component generation phase, the generic template should be linked with these procedures.

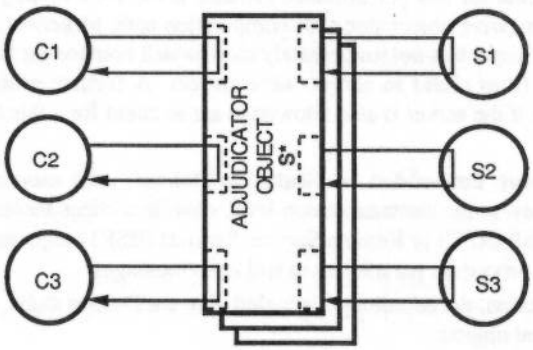


Fig. 2 - Combined Client-Server Adjudicator: Principle

The adjudicator implemented as an object has the advantage of allowing quite sophisticated fault-tolerance techniques.

An example is depicted in figure 3. Each client replica is associated to a specific server replica. Each service request is immediately forwarded to the associated server replica, without waiting for the other requests, therefore without going through the adjudication process. Adjudication is executed on the reply messages; the adjudged values are multicast back to clients.

On the request side, when all the replicated requests are available, *S** may compare them for error detection only.

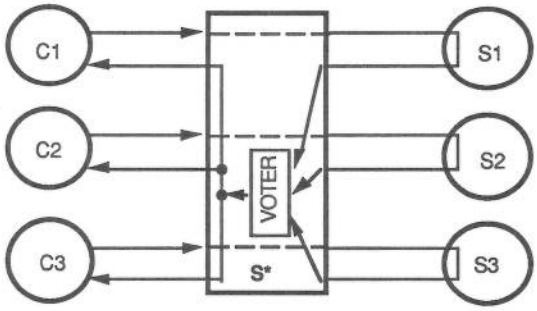


Fig. 3 - Combined Client-Server Adjudicator: Example

The advantages of the structure in this example are:

- It allows a high degree of diversity between the implementation of the clients (since their requests are not required to be compared). It is even conceivable that the formats of the RSR primitives are not the same for all the S_i-C_i pairs.
- The time to get through a chain of multiple clients and servers is approximately equal to the sum of the average execution and transmission times on the "request" path, instead of the sum of the worst times. Besides, the voting operation on requests can be executed concurrently with the main computation.

The price to be paid for this performance increase is the vulnerability to multiple software errors, namely in successive object along the computation path. Moreover, the recovery process is more complicated, since it is not immediately clear which component failed, and whether the interactions passed from client to server were correct. A further problem arises from the propagation of errors if the server is also allowed to act as client for a third object.

14.3.2.5. Adjudicator embedded in Stubs. In Deltase, *stub* modules are used to map the language-level view to the communication-level view. In a client-server interaction using the Remote Procedure Call (RPC) or Remote Service Request (RSR) language primitives, the stubs pack and unpack the procedure parameters to and from messages.

As an added function, the adjudicator is included in the Deltase stubs, both in the client and in the server replicated objects.

The adjudication procedures supplied by the user, according to a specific format, should be linked at component generation time to a modified Deltase stub module.

In figure 4, a subsystem composed of a 3-replicated client and a 3-replicated server is depicted. As all components are standard Deltase objects, communication among them takes place by means of RPCs or RSRs. The basic stub functions sketched out in [Powell 1988] should be complemented by: a) multicast of the message carrying the service request to all replicated servers; b) at the server input port, execution of the adjudication procedure on the data coming from the client variants. The same considerations apply for the handling of reply messages from the servers to the clients.

This configuration effectively distributes the adjudication service among the message-receiving modules. The consistency of servers, with regard to the sequence of input messages from different, replicated clients, is ensured by using the atomic multicast protocol. All the clients (even non-replicated ones) sending messages to a replicated server should use this facility.

Another source of inconsistency of servers can be the time-out mechanism, used to avoid hanging up while waiting for a message from a faulty client. In fact, whatever length of waiting

time is chosen, it is possible that some replicas time-out whilst others do not. Since this problem is bound to the independence of decisions in the replicas, it can be avoided only by some form of agreement between replicas on the decision itself.

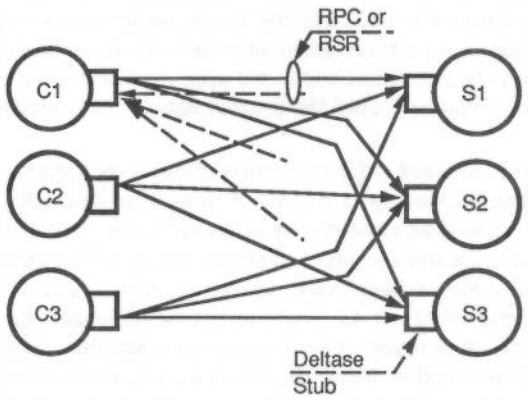


Fig. 4 - Adjudicator embedded in Stubs

If there are no specific safeguards, a slow replica can increasingly lag behind the other peers. It is conceivable to let faster replicas proceed, setting up a list of missing messages from the slower one(s) paired with the correct (adjudged) values. This allows safe discarding of late messages, as well as checking their correctness, for error detection purposes. This solution allows tolerance of temporary slowdown of a replica. After a given threshold in time or in the number of listed messages from an object, this one should be considered faulty, and the standard recovery procedure should be started (see also §6.5.2).

To avoid the possibility of erroneously declaring faulty a slow replica, some synchronization technique should be used. The cost is that the speed of progress of a replicated object over any given time interval is that of the replica that is slowest during that interval.

14.3.2.6. Communication-Level Adjudicator. If the application can be given specifications that reliably guarantee identical results, software-fault tolerance can be easily achieved using the N-Variant Programming model, as an extension of the N-modular redundancy already specified for hardware-fault tolerance. The MCS session-level, signature-based, IRp adjudicator can be used (cf. sections 6.5.2, 6.5.3 and 8.1.2.1). For example, a 1-fault-tolerant system would require 3 variants running on 3 hosts; to tolerate two simultaneous faults several configurations are available: i) 3 variants running on 9 nodes can mask 2 hardware faults, or 1 hardware plus 1 software fault; ii) the same result is obtained by using 4 variants on 7 nodes: more design effort traded against less run-time resources; iii) 5 variants on 5 hosts can also mask 2 software faults. The latter result stems obviously from the fact that each host equipped with a diverse variant is a fault-containment region (cf. chapter 4).

To allow wider use of software-fault tolerance, divergent variants are supported. Since application-dependent adjudication procedures, normally required in this case, cannot be included in the low-level communication software, the adopted solution, discussed in [Strigini 1988], is to use self-checking software components. The majority, signature-based, voting is of no use here; the adjudicator only has to choose among several correct results

The use of the *<ABSTENTION>* notification generated by failing self-checking components allows better performance, by i) reducing the use of time-outs in case of error, and ii) improving the adjudication process [Ciompi and Grandoni 1990]. As an example of the usefulness of this mechanism, consider the case where an application-level acceptance test shows that an individual output is erroneous. The programmer knows that this output is computed from the values of a data structure that is continuously updated with data from outside the software component. It is reasonable to send an abstention message instead of the output that was found to be erroneous, and to the same destination, but not to alter the normal execution flow of the component, since the internal state of the component may soon correct itself.

A simple example is depicted in figure 5: two self-checking variants, *A* and *B*, run on three hosts, with *A* replicated in two copies *A1*, *A2*. A straightforward adjudication policy is the following: i) if *A1* or *A2* send an abstention or differing results, then choose the result from *B*; ii) if *A1* and *A2* generate the same result, then choose this one. This configuration can tolerate one hardware or one software fault. Now, let variant *A* be faulty. Without the use of the abstention message, neither *A1* nor *A2* would send a message, and the value produced by *B* would trigger an adjudication round. After a proper time-out, this value would be considered valid and forwarded to destination. However, if the host where *B* is running fails by outputting an undue or "impromptu" message, this message would be considered valid and then erroneously forwarded. If abstention-messages are used instead, the adjudicator should expect to get, after the message from *B*, at least one message from *A1* and/or *A2*. No more than one host can fail, and if the variant *A* has a bug, it sends an abstention notification message. This allows the discrimination of the above erroneous condition.

In summary, at the MCS level non-divergent variants with exact comparison, and self-checking variants, are supported. In both cases there is a need for new adjudication procedures (other than the present majority rule); in the second case the possibility of notifying an abstention is also required.

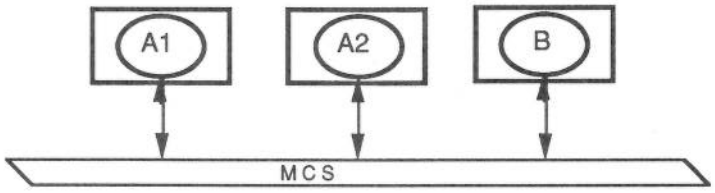


Fig. 5 - Communication-Level Adjudicator

The above illustrative examples of adjudication functions are based on fault hypotheses expressed in terms of number of faults. It is of course unrealistic to exclude the possibility of higher number of faults, and to treat all possible failures as though they had the same importance. Adjudicators that take into account the expected probabilities of all fault patterns can be designed as described in [Di Giandomenico and Strigini 1990].

14.3.3. The Problem of State Recovery of Variants

Once an error (or even a disagreement between correct variants) has been detected in some result, the first problem is how to mask the error and give the user (of the result itself) a correct result; once that is solved, a second problem is what to do with the failed (or disagreing)

variant(s). The internal state of the variant is presumably corrupted. We may therefore expect subsequent results to be erroneous as well, and thus not only useless but dangerous.

A first choice is to kill the erroneous variant altogether. For short-lived components this seems advantageous. The component will be automatically "repaired" when its execution terminates and, at the next execution, a correct redundant module will be instantiated, from the same code, taking its internal state from an uncorrupted global state (data base, sensor readings, etc.).

For components with long execution life-spans, this is not appropriate: too many variants would be needed (the analogous hardware configuration is self-purging N-modular redundancy without repair). Then, there is a need to recover a variant that erred, so that it can catch up with the correct variants and carry on its work. Recovery brings with it an inherent problem that we discuss in the following.

Recovery may be:

1) programmed into the same variant that errs (estimation of approximate correct values, for instance, or reset to safe values). This is the simpler choice, but is not widely applicable; in particular, as noted earlier, perfectly healthy variants might drift apart from each other, with time, if no built-in way exists to recover them to consistent states.

or,

2) the recovery may use values produced by the other variants. The problems with this solution, i.e., correcting the internal state of a variant or create a state based on the states of the others, is that it requires either identical internal representations of those states (same internal variables laid out in the same way) or a way to translate between diverse representations.

For Delta-4, the latter solution is proposed. Specifically, a unique representation of internal state *for inter-variant exchange,* not necessarily for internal use, must be given in the object specification. Each variant must be equipped with special *Input_State, Output_State* routines; the latter should be able to be started, in pre-defined computation points, upon receiving a specific signal from System Administration (SA).

Assuming that the redundant configuration is made up by three variants *C1*, *C2*, *C3*, and that *C2* has to be re-instantiated, the state cloning operation can be outlined as follows:

a) *SA* loads new *C2* code;

b) *SA* activates *C2* on the Input_State routine; *C2* starts, waiting for the state information from *SA*;

c) *SA* sends *C1* and *C3* signals to enter the Output_State routine, which should be bound to *SA*;

d) At proper times, *C1* and *C3* output their state to *SA*, after executing some synchronization protocol, then they wait for a *Continue* signal from *SA*;

e) *SA* sends *C2* the state received (options: send the first received; send all copies, with the number thereof as an additional parameter);

f) *C2* sets internal state (option: checking validity if multiple copies received), then waits for the *Continue* signal (option: an acknowledgement signal to *SA* can be needed before beginning to wait);

g) *SA* sends all variants the Continue signal.

The state cloning operation describer above should be considered as a complement to the present cloning process.

14.4. Specifying Software Components for Software-fault Tolerance — Tradeoffs in an Object Environment

Any software-fault tolerance paradigm using redundant components needs to tradeoff between the conflicting needs: i) establish the minimum number of design constraints to allow diversity; the design manager can then force diversity among programmers, issuing individual constraints (e.g., development environment, language, etc.); ii) issue specifications that are tight enough to insure some pre-defined "homogeneity" among the diverse results.

As a simple example of the problem, consider the component *A* implemented as three variants *A.1*, *A.2*, *A.3*. If no specific restrictions are given to the variant designer, it can happen that, say, *A.1* and *A.2* are both programmed to import a service interface from the *same* component *B*, while *A.3* uses services from another component *C* or, possibly, no external resources at all. The following problems arise:

a) A design fault in *B* can cause a failure in both *A.1* and *A.2*; in this case, it is likely that the correct result computed by *A.3* will be outvoted, leading to a catastrophic failure.

b) The variants exhibit different communication patterns. This makes it difficult to determine where and how to place adjudication mechanisms; for example, are the service calls to *B* from *A.1* and *A.2* to be adjudicated?

c) There may be consistency problems in shared components, like *B* in the example, which are quite unusual: in fact, *A.1* and *A.2* are semantically the same component, therefore for correct behaviour their operations on *B* should be made idempotent, in some way.

d) A problem related to the point c) above is that of recovery: if, say, variant *A.1* fails after interacting with *B*, the recovery of *A.1* mandates recovering *B* too, and then, in the classical domino effect, *A.2* must be brought back to a consistent state. This operation, possibly triggered by a single fault, may require undoing preceding successful adjudications: a much more complex computational model would be required.

In the object model of computation, the conflict depicted above appears harder, because of the emphasis put on hiding the internal structure of the object, as well as in the methods used in implementing the offered services (in particular, which services from other objects are used).

On the other side of the coin, one can wonder if adjudicating results surfacing at the object boundary is enough: relatively complex, "taciturn" objects may suggest performing cross-checks among intermediate results, to prevent non-recoverable divergence among variants. However, this again contrasts the internal information hiding philosophy.

For the sake of simplicity, and as a first step in devising software-fault tolerance structures in object-oriented systems, like those based on the Delta-4 architecture, it is assumed here that adjudication is only performed at the level of object interfaces (as accomplished in the validate-before-propagate paradigm to mask hardware-caused errors). In other words, the scope of the attainable diversity is limited to the internal implementation of the object specification, since diverse use of external services is forbidden. Of course, the issue of how to organize diverse implementations across object boundaries is of great interest. In fact, some simple extensions can be easily given. For example, concurrent use of services exported by a memory-less object can be safely allowed to variants, from the point of view of information consistency. Such an object, is of course a reliability bottleneck. More general methods for wide-scope software-fault tolerance structures are under investigation.

14.5. Concluding Remarks

Techniques and mechanisms to implement and support software-fault tolerance in systems built on Delta-4 machines have been described. According to the Delta-4 philosophy, a range of solutions is envisaged, differing in complexity and cost as well as in the attainable dependability.

Some tradeoffs presented here are likely to be superseded by better solutions in the near future. For example, the restriction that variant objects must exhibit identical interfaces may be relaxed, by introducing higher level structuring concepts that allow wider scope diversity, while ensuring data consistency and recoverability.

The proposed solutions are based on the assumption of fail-uncontrolled hosts. More restrictive assumptions may lead to simple solutions for software-fault tolerance.

Recovery blocks, for example, can be easily implemented in conjunction with the passive replication model for fail-silent hosts. In this model, the backups do not execute recovery blocks, just like any application code. However, the ordinary checkpoints are not sufficient to enable recovery from a software error occurring after a hardware fault. In this case, the next alternate in the recovery block must be executed *in the backup host*, after restoring the initial information, which was set at the *recovery point* in the primary host. The recovery point needs to be sent to the backup independently of the checkpointing mechanism: successive checkpoints along the execution of the recovery block supersede the previous one, while recovery point information must be held until the recovery block is exited.

A possible solution based on the leader-follower model could be the following. The follower does not maintain recovery points, and does not execute multiple alternates. On arrival at a recovery block, the followers suspend waiting for instructions from the leader. The leader runs through the recovery block and, on completion, informs the followers on which alternate to execute. The followers execute that alternate and the acceptance test. The replica determinism requirements of the leader/follower model would ensure that the alternate succeeds in the follower as it did in the leader in most cases. However, if a Heisenbug occurs in a follower, the acceptance test should fail; in that case, the classic recovery block paradigm would trigger the execution of the next alternate, starting a state divergence with the leader. The acceptance test failure should instead cause the abort and successive cloning of the follower. Therefore, a mechanism to trigger the abort of a follower from the component itself should be added to the standard leader/follower support.