Adaptive Architectures for Hybrid Fault Tolerance in Distributed Computing Systems

J. Xu, F. Di Giandomenico[1], A. Bondavalli[2]

Dept. of Computing Science, University of Newcastle upon Tyne, UK

[1] IEI/CNR, Pisa, Italy; [2] CNUCE/CNR, Pisa, Italy

Abstract

This paper discusses the issue of hardware and software fault tolerance in distributed computing environments as well as issues related to efficiency and flexibility. A set of new fault-tolerant architectures is presented, and a detailed dependability analysis of these architectures together with an efficiency evaluation is performed. The proposed architectural solutions are based on the assumption that the distributed supporting environments under consideration are highly varying and they would support multiple competing applications associated with different fault-tolerant architectures, thereby exhibiting dynamic service characteristics. Stress is placed on *adaptation* — the major goal of designs of these new architectures is to attempt the adaptive execution of redundant components so as to minimize hardware resource consumption and shorten the response time, as much as possible, for a required level of fault tolerance.

Key words — Adaptive architectures, dependability, distributed computing environments, efficiency, hardware and software fault tolerance, response time.

This work has been supported by funding from the ESPRIT Basic Research Action 6362 entitled "Predictably Dependable Computing Systems (PDCS2)".

I. Introduction

General purpose information-processing distributed systems are coming to life in which the motivation for distribution is primarily functional or administrative [1]. Such a system may contain a number of computing nodes with very different characteristics which may be connected by different kinds of communication networks. The system is intended to support many different applications and to execute concurrently many unrelated requests that could compete for both the hardware and the software resources. We will limit our attention to this class of distributed systems in this paper. However, in such systems distribution is not a direct solution to dependability and very high dependability cannot be achieved merely by backup or replication. In fact, the development of highly dependable computing systems may require the combined utilization of a wide range of techniques, including *fault tolerance* techniques intended to cope with the effects of *faults* and avert the occurrence of *failures* or at least to warn a user that *errors* have been introduced into the state of the system [2]. As we know, the provision of means for tolerating anticipated hardware faults has been a common practice for many years. A relatively new development concerns the techniques for tolerating unanticipated faults such as design faults in general and software faults in particular.

Software fault tolerance generally needs redundancy of software design or *design diversity*. Design diversity can be defined as the production of two or more systems (e.g. software modules) aimed at delivering the same service through independent designs and realizations [3]. The systems, produced through the design diversity approach from a common service specification, are called *variants*. Incorporating two or more variants of a system, tolerance to design faults necessitates an *adjudicator* [4], which is based on some previously defined decision strategy and is aimed at providing (what was assumed to be) an error-free result selected from the outcomes of the variants. The well-documented techniques for tolerating software design faults include recovery blocks [5], *N*-version programming [6], *N* self-checking programming [7].

In recovery blocks (RB) the variants are named alternates and the main part of the adjudicator is an acceptance test that is applied sequentially to the results of the variants: if the first variant (primary alternate) fails to pass the acceptance test, the state of the system is restored and the second variant is invoked on the same input data, and so on sequentially until either the result from a variant passes the acceptance test or all the variants are exhausted. In the *N*-version programming approach (NVP), each of *N* variants (or versions) is executed in parallel and the adjudicator performs an adjudication algorithm on the set of results provided by the variants to identify a correct result. *N* self-checking programming (NSCP) attains fault tolerance by the parallel execution of *N* self-checking software components. One of the components is regarded as the active component, and the others are considered as "hot" stand-by spares. Upon failure of the active component, service delivery is switched to a "hot" spare.

A newer scheme for software fault tolerance, called the Self-Configuring Optimal Programming scheme (SCOP), has been recently introduced by the authors [8, 9]. SCOP attempts to reduce the cost of fault-tolerant software, both in space and time, by providing designers with a flexible redundant architecture where dependability and efficiency could be combined dynamically at run time. Basically, it is organized in phases, each one involving the execution of an appropriate subset of variants. At the end of each phase, an adjudication is performed on a growing syndrome space which contains all the relevant information collected so far. Once conditions for the release of a result are satisfied any further phases and operations are ended, or otherwise a new phase is performed until variants are exhausted.

The SCOP scheme has high run-time efficiency, i.e. it makes a good use of the system resources available, mainly because of its working architecture that is designed following the observation — different amounts of redundancy are required for error detection and for fault tolerance. For given fault hypotheses, the amount of resources necessary for the detection of admitted fault situations is considerably smaller than those necessary for the tolerance to them. However, the gain of efficiency by SCOP would be limited if the supporting system (involving the hardware and the system software) were to be used only for a specific application where the hardware resources saved would be merely left idle. While considering general-purpose distributed systems, the significance of dynamic control and adaptive redundancy increases dramatically since dynamic combination of dependability and efficiency highly matches the characteristics of such systems — the saved resources by a specific application would be exploited in turn by other competing applications.

The rest of the paper is organised as follows. Section II addresses the problem of how fault tolerance could be incorporated into computing systems in a controlled manner. In the third section several hybrid-fault-tolerant architectures are defined at the top of a general-purpose distributed computing environment. In the fourth section we evaluate dependability of the architectures under consideration. Section V analyzes these architectures with respect to resource cost and response speed. We conclude this paper in Section VI.

II. System Design and Hybrid Fault Tolerance

Two major approaches to the incorporation of fault tolerance into systems could be identified which represent two extremes of a set of possible choices. The first, here called *structured* approach, starts from the system structuring principle widely applied in the design of general computing systems where a system is partitioned into different (and modular) abstraction layers, each performing its own tasks and providing services for the upper ones. Following this approach, the most profitable fault-tolerant techniques may be applied to different layers in such a way that the service by a specific layer is associated with the required features, usually with a restricted "failure semantics" [10]. (For example, a fail-stop component may deliver only correct services or no service at all [11, 12].) This has the merit in controlling extra complexities of fault tolerance. Since the set of faults needed to be treated is isolated to within a single layer plus a set of well-defined failures of the underlying layers, the provision of fault tolerance in that layer becomes a simple and controlled task. However, the approach may cause the loss of efficiency and performance — firstly, run-time costs introduced by fault tolerance in each layer are generally summed up resulting in a very high run-time overhead in a functioning system, especially in absence of faults, and secondly, the fault-tolerant techniques in different layers may overlap heavily leading to poor performance.

The other approach is here called *integrated* approach. Redundancy is still spread over layers (it could not be otherwise) but its management and the actions directed towards tolerance to faults are concentrated only in some (higher) layer. Faults in some lower layer are propagated upwards and are masked, further detected and treated, by a previously selected higher layer. This approach is obviously complementary to the structured one. The overlap of fault-tolerant mechanisms and techniques could be controlled and minimized so as to improve efficiency and performance, but the way of incorporating fault tolerance into systems would be quite complex. In particular, the set and granularity of faults would become much bigger, which gives rise to difficulties in performing exact fault diagnosis and other derivable consequences.

It is hard and contended to say whether one approach has to be generally preferred to the other, and we are not interested in disputing on this issue. In this paper we will describe our architectures primarily based on the integrated approach. Two layers, one software and one system/hardware, are distinguished. Fault-tolerant mechanisms and schemes applied in the software layer are in charge of assuring fault tolerance for a given architecture and for the corresponding application. To be more exact, the software layer may consist of many different, fault-tolerant or non-fault-tolerant, applications which use different architectures to achieve fault tolerance or other goals, whereas the system/hardware layer corresponds to a distributed supporting environment which contains a set of computing nodes connected by communication networks. Here, a "hardware" node (or component) is composed of both the hardware and the associated executive software that provides the necessary services for the execution of the application software and the node may have disks organized as stable storage.

Typical hardware failures caused by hardware faults include node (or workstation) crashes and inability of communication such as the repeated loss of messages. It is usual to assume that information and data recorded on stable storage are not destroyed by node crashes. We will assume that although the effects of hardware failures would be masked by fault-tolerant architectures in the software layer, it has to be the responsibility of the distributed supporting system to treat hardware failures, including fault diagnosis and the provision of continued service. Software and hardware faults could be further classified here as *independent* and *related* faults [13]. Related faults are either faults in the common specification, or come from dependencies in the separate designs and implementations. Independent faults usually lead to *separate* failures, whereas related faults manifest themselves in the form of *common mode* failures [14]. It is reasonable to assume that faults in the system/hardware layer can be independent only (rather than related) because of the factors of both professional system programmers and well-established hardware techniques. In general, such independent faults will not lead to common mode failures of software components or variants. Under this assumption common mode failures of a set of software variants in the software layer can be caused only by related faults among the software variants.

There is some similar work on the definition and analysis of hardware- and software-fault-tolerant architectures. In particular, Laprie, Arlat, Beounes and Kanoun [14] presented a set of hybrid-fault-tolerant architectures and analyzed and evaluated three of them. However, their architectures are based on a fixed set of hardware components and not related to both the dynamicity of hardware resources available and the efficiency issues. Such architectural solutions cannot well match the characteristics of general-purpose distributed systems where the resources must be competed by many unrelated but concurrent service requests. In fact, the architectures with the fixed requirement to hardware components are either inefficient or unavailable in a varying environment. Our architectural solutions provided in the next section are particularly aimed at the problem of how to make a good use of limited hardware (and common service software) resources in an adaptive manner to achieve a given degree of dependability.

III. Architectural Solutions

We shall define in this section a set of architectures for hybrid fault tolerance under the assumption of distributed supporting environments. Figure 1 shows a relationship between our architectures, abstraction layers and the distributed system.

First, each of our architectural solutions is single-application-oriented. For a given application, an architecture would contain a set of software variants designed independently, an adjudicator and a control program. These programs and the related data (e.g., inputs and outputs) may be stored on the disks of some nodes. The architecture, or more precisely its control program, must guarantee that the state information and output results are produced dependably with a required probability and recorded correctly on stable storage. The control program is thus responsible for helping error recovery in the software layer as well as for reporting faults to the supporting system.

Secondly, our architectures are not attached to a fixed set of hardware nodes. An architecture, thus a given application, must request hardware computing resources from the supporting system upon invocation and return them to the system when the required computation terminates. During the computation, the architecture may make requests for additional resources if necessary.

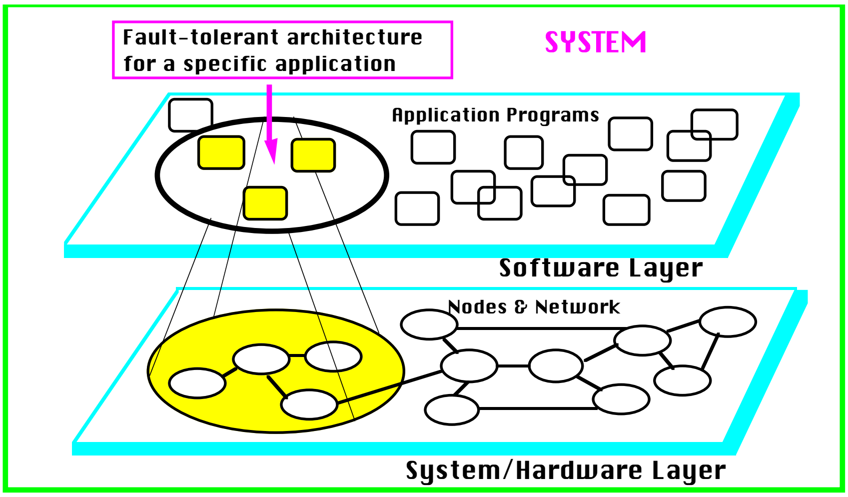

Figure 1. Architectures, abstraction layers and system.

Before introducing the architectures, we need further to list a set of assumptions that are common to all the architectures, which are used to simplify the definitions and evaluation of the proposed architectural solutions. (1) An adjudicator is replicated on all the hardware components supporting a specific architecture, but a selected hardware node has responsibility for taking a final decision from the decisions made by local replicated adjudicators and producing outputs of the architecture. As the final adjudication is short and simple, it is thus assumed to be highly dependable. (2) Control programs are organized in a similar manner. They are distributed locally and a final decision is made by a selected node based on local control commands.

We characterize an architecture with respect to three aspects: (1) levels of hybrid fault tolerance, (2) hardware resource consumption, and (3) impacts upon response speed. An architecture is denoted by a group of multi-elements $\boldsymbol{X}$ (F, N, H_b, H_{max}, ...) where

X indicates a specific architecture for hybrid fault tolerance, being equivalent to the name of the selected scheme for software fault tolerance;

F indicates the number of (hardware- and software-) faults to be tolerated and is further expressed by a detailed form: (f, i, j) in which f is the minimum number of hybrid faults to be tolerated, i is the number of hardware faults to be tolerated assuming perfect software, and j is the number of software faults to be tolerated assuming perfect hardware;

N is the number of application-specific software variants;

H_b is the basic (minimum) number of hardware nodes an architecture requests from its supporting system in order to achieve the given level of hybrid fault tolerance F;

H_{max} is the maximum (total) number of hardware nodes an architecture requests from its supporting system in order to achieve a given level of hybrid fault tolerance F when the worst fault situation occurs.

The response time of a static architecture attached to a fixed number of hardware nodes is mainly affected by factors of the supporting system such as scheduling algorithms for resource allocation and mechanisms for remote access. When an architectural solution is based on adaptive requests of the hardware components, the response time would be heavily increased in some rare, worst fault situations. However, if such rare events are acceptable by certain applications, the average response time of an adaptive architecture is even better than that of a corresponding static architecture because the number of hardware nodes requested by it is much smaller in the absence of faults. Due to the system-specific characteristics of response speeds, we will not incorporate obviously labels for response time into the ***X*** (F, N, H_b, H_{max}, ...) expression though we may do so whenever the need arises.

3.1. The SCOP architecture

Although various solutions of the SCOP architecture could be constructed, we here define two significant instances of SCOP, **SCOP**((2, 3, 2), 5, 3, 5) and **SCOP**((1, 2, 1), 3, 2, 4), as depicted in Figures 2 and 3. The **SCOP**((1, 2, 1), 3, 2, 4) instance will be further analyzed in detail and also used as a sample for dependability evaluation and efficiency estimation, compared with the corresponding NVP and RB examples.

SCOP((2, 3, 2), 5, 3, 5)

This SCOP instance normally requests three hardware nodes, and two further hardware nodes may be needed when a failure is detected in the software layer, as shown in Figure 2. Five application-specific software variants are distributed on, or accessible to, these basic and additional hardware components. This instance can tolerate any two hybrid (software or hardware) faults at least. If no any software fault manifests itself during computation, up to three hardware faults will be tolerated. In the absence of failures, only three of five variants execute in parallel on three hardware nodes, thereby benefiting efficiency. Response speed would also be affected positively since the average time of variant execution, of hardware resource requests and of possible remote access are relatively short in comparison with the architecture attached statically to five hardware components. The response time would be heavily increased only when a failure occurs, and it would be doubled in order to mask the effect of the faults.

Since realistic examples of implementing software fault tolerance are most based on three software variants, the next instance has a more practical implication.

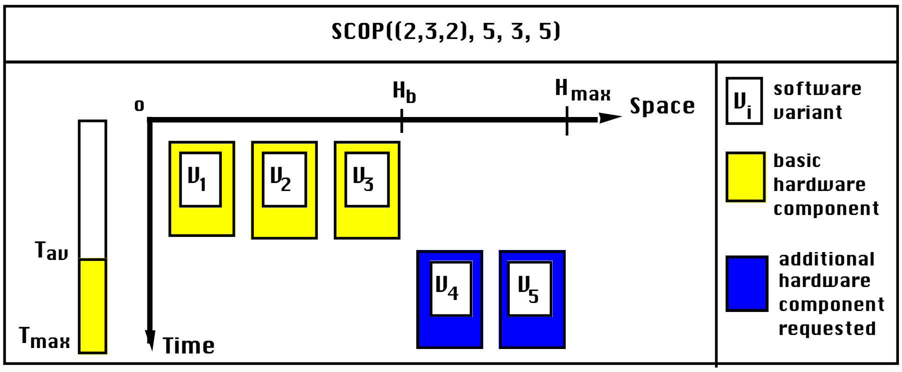

Figure 2. An instance of the SCOP architecture.

SCOP((1, 2, 1), 3, 2, 4)

This instance requests two basic hardware nodes and, when a failure occurs in the software layer, two further hardware nodes, as shown in Figure 3. Three software variants are distributed on, or accessible to, these basic and additional hardware components. This instance can tolerate one (hardware or software) fault at least. If no any software fault manifests itself during computation, two hardware faults will be tolerated. Normally, two variants execute in

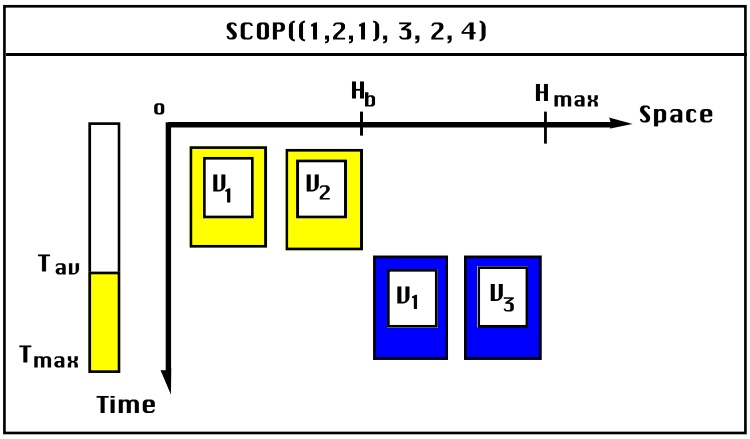

Figure 3. Another instance of the SCOP architecture.

parallel on two hardware nodes. Just in the presence of faults, the third variant will execute in parallel with one of the other variants on two additional nodes requested.

More precisely, **SCOP**((1, 2, 1), 3, 2, 4) organizes the execution of variants into two phases. In the first phase, variant 1 and variant 2 run on two hardware components and the adjudicator compares their results. Consistent results are immediately accepted. Otherwise, the second phase begins: variant 3 and variant 1 run on two additional hardware components. Then the adjudicator makes a decision based on all the four results, seeking a 2-out-of-4 majority. Figure 4 shows the details of how this instance works with respect to different fault situations, where the paths correspond to the different possible outcomes:

(1) at the end of the first phase there exists a majority representing a correct computation and the output is a correct result;

(2) at the end of the first phase the result is rejected, at the end of the second phase there exists a majority representing a correct computation and the output is a correct result;

(3) at the end of the first phase an erroneous result is accepted;

(4) at the end of the first phase the result is rejected, at the end of the second phase an erroneous result is accepted;

(5) at the end of the second phase the result is rejected;

(6) the duration of the redundant execution exceeds a specified limit (when a real-time constraint is present) and the execution is aborted.

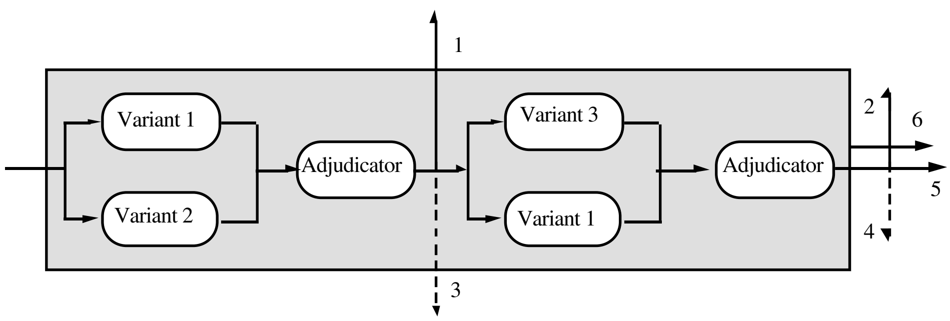

Figure 4. SCOP((1, 2, 1), 3, 2, 4) operation.

3.2. NVP and RB architectures

Two instances being considered here are **NVP**((1, 2, 1), 3, 4, 4) and **RB**((1, 2, 1), 2, 2, 3) for the sake of comparison and the subsequent evaluation.

NVP((1, 2, 1), 3, 4, 4)

This instance requests four hardware nodes. Three software variants are executed in parallel on these nodes, as shown in Figure 5. The NVP instance can tolerate one hybrid fault. If no any software fault manifests itself during computation, two hardware faults can be tolerated. The response time of **NVP**((1, 2, 1), 3, 4, 4) is guaranteed by executing the variants within a single phase, but it may be affected negatively by the time of requesting four hardware components from the system and that of possible remote access. In the extreme case where its request of hardware components cannot be met completely, this instance will not work.

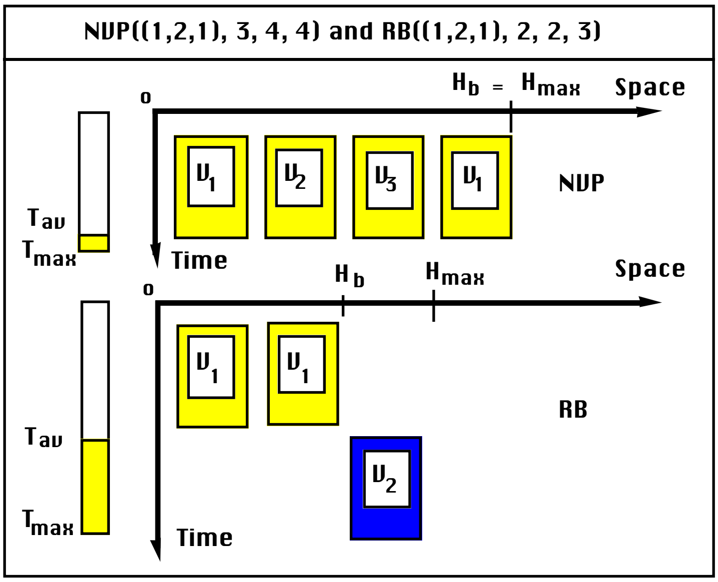

Figure 5. Instances of NVP and RB architectures.

RB((1, 2, 1), 2, 2, 3)

Normally, the primary variant V_1 is executed on two hardware components (see Figure 5). The results produced by the replicated variants are compared. (1) If they agree, acceptance tests are applied to them. The agreed result is released unless both acceptance tests reject it. In this case, an additional hardware node will be requested and the variant V_2 will be executed on it. (2) If the results produced by the replicated variants in the first phase disagree, a diagnostic routine is applied to them. When both hardware nodes are diagnosed as faulty, an additional node is needed to execute the variant V_2. This proposed solution can tolerate at least one hardware or software fault. It is also highly efficient when no fault manifests itself during computation — a most possible situation. Response speed would also be affected positively. However, the application that uses this architectural solution must be able to accept a rare, but still possible heavy degradation — the response time would be much longer while performing self-diagnosis and executing the variant V_2 in order to mask the effect of two hardware faults.

IV. Dependability Evaluation

In this section, the dependability analysis of **SCOP**((1, 2, 1), 3, 2, 4), **NVP**((1, 2, 1), 3, 4, 4) and **RB**((1, 2, 1), 2, 2, 3) defined in section III is performed adopting a Markov approach. In [14], the dependability analysis of hybrid architectures tolerating only one hardware or one software consecutive fault is conducted by first determining the two models for the hardware and for the software separately and then combining the obtained results into a single model. Our framework differs from [14] in (1) the faults our architectures tolerate and (2) the used model which is a combined one for both hardware and software faults. Our analysis starts from a set of special software failures which would lead to the failure of the whole architecture despite of the hardware conditions. Hardware failures are considered only when they affect the whole architecture alone or with some software failures together. In the analysis, we do not distinguish between detected and undetected failures. In addition, with the term adjudicator (and AT when referred to the RB architecture) we mean both the adjudication function and the control program.

Basic assumptions *special* for the evaluation are:

(1) hardware failures are independent and the probability that correct software variants running on the failed hardware nodes produce the same incorrect outputs is negligible;

(2) compensation among failures never happens, neither between software variants, nor between software variants and the adjudicator nor between hardware components and variants;

(3) for architectures with multiple phases, the adjudicator exercised in more than one phase will show the same erroneous or correct behaviour in all the phases (from the software behaviour point of view);

(4) hardware faults are independent from software faults (and vice versa): a failure in an hardware component will cause an incorrect output of the variant running on it, but will have no influence in activating a fault in the variant itself.

Table 1 shows the relevant types of failures of software and hardware components for the **SCOP**((1, 2, 1), 3, 2, 4) architecture.

The detailed dependability model of one execution of the SCOP architecture is illustrated in Figure 6. We briefly explain the meanings of the states and arcs in the figure. The model doesn't exactly describe the two phase execution of the architecture because the states representing the execution of the second phase are introduced only when necessary.

Failure Types	**Probabilities**
3 variants fail with consistent results	q_{3v}
2 variants fail with consistent results (the 3rd variant, can fail or not but produces a result different from the previous two)	q_{2v}
The adjudicator fails selecting an erroneous result (given that an erroneous result has been produced by a variant)	q_{vd}
A variant fails, conditioned on none of the above events happening	q_{iv}
Conditioned on the existence of a majority, the adjudicator fails by not recognising a majority (without releasing any result)	q_d
An hardware component fails during an execution, affecting the variant and/or the adjudicator running on it (independence is assumed between the failure of two or more hardware components and between hardware components and variants)	q_h

Table 1: Failure types and notation for SCOP((1, 2, 1), 3, 2, 4).

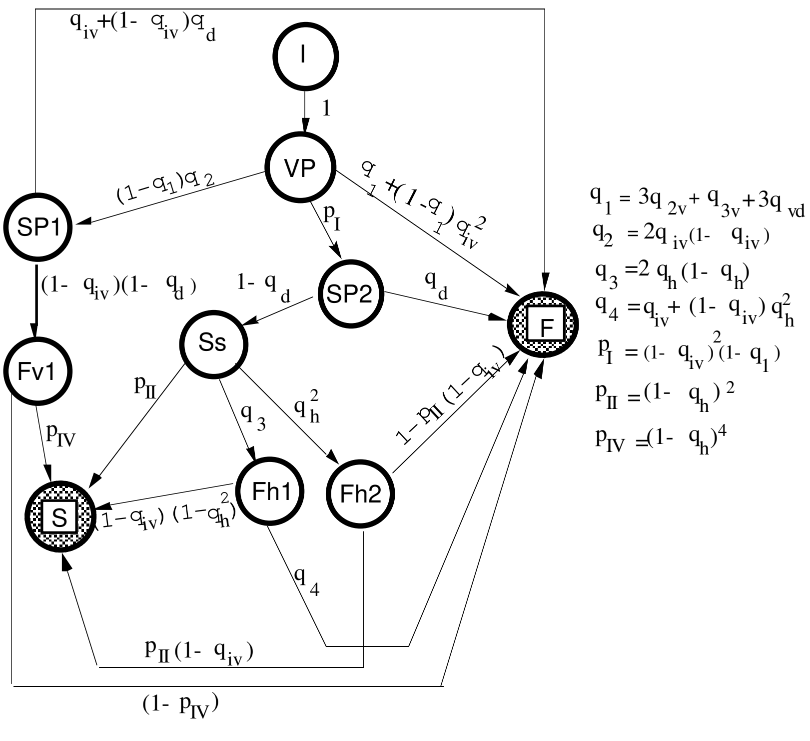

Figure 6. The dependability model for SCOP((1, 2, 1), 3, 2, 4).

I initial state of an execution;

F failure of the architecture (absorbing state);

S success of the architecture (absorbing state);

VP two variants are executed on two hardware components in the first phase; the arc from VP to F is labelled with the sum of the probabilities of the software failures causing the failure of the whole architecture without considering the hardware behaviour (i.e., common mode failures among the variants and among the variants and the adjudicator, and the independent failure of both the executed variants);

SP1 one of the two variants executes correctly while the other fails, and the second phase is performed;

SP2 the two variants execute correctly, but if the adjudicator fails in recognizing the agreeing result, the second phase will be performed unnecessarily; if the adjudicator works correctly, the state representing the correctness of the software components executed during the first phase is reached;

Fv1 just one variant fails after the first phase; success or failure of the whole execution depends on the behaviour of the hardware components through the two phases;

Ss software components including the adjudicator in the first phase are correct; the specific behaviour of the hardware components leads to the final state S or to the second phase (states Fh1 and Fh2);

Fh1 the second phase operates due to the failure of one hardware component during the first phase; success or failure of the whole execution depends on the behaviour of both hardware components and the software variants in the second phase;

Fh2 the second phase operates due to the failure of two hardware components; success or failure of the whole execution depends on the behaviour of both hardware components and the software variants in the second phase.

To simplify the expression of the solution, we define a set of intermediate parameters, as shown in the right side of Figure 5. From the state transition diagram, the probability of failure $Q_{SCOP((1,2,1),3,2,4)}$ of the **SCOP**((1, 2, 1), 3, 2, 4) architecture is:

$$Q_{SCOP((1,2,1),\ 3,2,4)} = q_1+(1-q_1)q_{iv}^2+p_I(q_d+(1-q_d)(q_h^2(1-p_{II}(1-q_{iv}))+q_3q_4))+$$
$$+ (1-q_1)q_2(q_{iv}+(1-q_{iv})q_d+(1-q_{iv})(1-q_d)(1-p_{IV}))$$

Similar models are derived for the other two architectures. Owing to the limitation of space, we omit details of these models and show the solutions directly. Note that Table 1 can be applied to **NVP**((1, 2, 1), 3, 4, 4). Table 2 introduces the relevant types of failures of software and hardware components for the **RB**((1, 2, 1), 2, 2, 3) architecture.

Failure Types	Probabilities
Primary and secondary alternates fail with consistent results and AT accepts their results	q_{psa}
The above event does not occurs, primary alternate fails and AT accepts its result	q_{pa}
The secondary alternate fails and AT accepts its result, conditional on the secondary being executed	q_{sa}
Primary and secondary alternates fail (with consistent results) and AT rejects their results	q_{ps}
Failure of the primary or secondary alternate (assumed independent)	q_p, q_s
Failure of the acceptance test causing it to reject a result, given the result is correct	q_a
An hardware component fails during an execution, affecting the variant and/or the adjudicator running on it (independence is assumed between the failure of two or more hardware components and between hardware components and variants)	q_h
probability that an hardware failure affects only the AT running on it, conditional on the hardware failed	c_{AT}

Table 2: Failure types and notation for RB((1, 2, 1), 2, 2, 3).

The expressions of the probability of failure, $Q_{NVP((1,2,1),3,4,4)}$ and $Q_{RB((1,2,1),2,2,3)}$, are as follows:

$$Q_{NVP((1,2,1),3,4,4)} = q_1+(1-q_1)q_2+3q_{iv}(1-q_1)(1-q_{iv})^2(q_d+(1-q_d)(1-p_{IV}))+ \\ +(1-q_1)(1-q_{iv})^3(q_d+(1-q_d)q_I)$$

$$Q_{RB((1,2,1),2,2,3)} = q_1+(1-q_1)q_5+(1-q_1)q_p(1-q_a)(1-q_s)(1-P_{III})+ \\ +(1-q_1)(1-q_p)(1-q_a)(q_h^2(1-c_{AT})^2+2q_h^2c_{AT}(1-c_{AT}))(q_s+(1-q_s)q_h+q_h^2c_{AT}^2q_s)$$

Table 3 lists a set of intermediate parameters used in the above expressions.

NVP intermediate parameters	**RB intermediate parameters**
$\mathbf{q_1}=3q_{2v}+q_{3v}+3q_{vd}$	$\mathbf{q_1}=q_{pa}+q_{psa}+q_{ps}+q_pq_{sa}$
$\mathbf{q_2}=3q_{iv}^2(1-q_{iv})+q_{iv}^3$	$\mathbf{q_5}=q_pq_a(1-q_s)+q_pq_s+(1-q_p)q_a$
$\mathbf{q_I}=q_h^4+4q_h^3(1-q_h)$	$\mathbf{p_{II}}=(1-q_h)^2$
$\mathbf{p_{IV}}=(1-q_h)^4$	$\mathbf{p_{III}}=(1-q_h)^3$

Table 3: Intermediate parameters for the NVP and RB architectures.

The derived expressions are too complex to allow a direct comparison among the three architectures with respect to dependability. As an example, Figure 7 gives a plot of the functions representing probabilities of failure of the three architectures under consideration. To produce the plot, precise values are chosen for the dependability parameters and listed in Table

4. (1) The values of parameters representing adjudicator failures and common-mode failure among three software variants are fixed for SCOP and NVP; in particular, it is assumed that AT has a higher probability of failure since it is usually more complex than the other adjudicators. (2) The probability of common-mode failure between two variants is assumed to be three orders of magnitude higher than that of independent failures; this assumption is also applied to the probability of common-mode failure between one alternate and the AT in RB. (3) The probability of independent failure of variants varies between 10^{-5} and 10^{-3}. (4) The probability of hardware failure is fixed.

Recovery Block	N-Version Programming	SCOP
$q_{ps} = q_{pa} = q_{sa} = Q\ 10^{-3}$	$q_{2v} = Q\ 10^{-3}$	$q_{2v} = Q\ 10^{-3}$
$q_{psa} = 10^{-10}$	$q_{3v} = q_{vd} = 10^{-10}$	$q_{3v} = q_{vd} = 10^{-10}$
$q_p = q_s = Q$	$q_d = 10^{-9}$	$q_d = 10^{-9}$
$q_a = 2 \times 10^{-7}$	$q_{iv} = Q$	$q_{iv} = Q$
Q: variable from 10^{-5} to 10^{-3}	Q: variable from 10^{-5} to 10^{-3}	Q: variable from 10^{-5} to 10^{-3}
$q_h = 10^{-3}$	$q_h = 10^{-3}$	$q_h = 10^{-3}$
$c_{AT} = 10^{-3}$		

Table 4: Values of the "dependability" parameters used in the example.

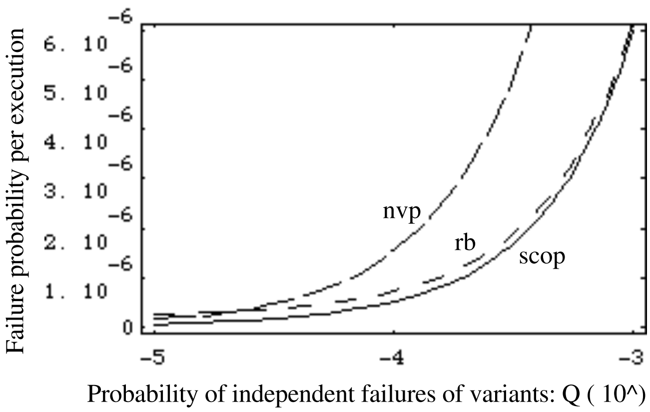

Figure 7. Plot of the probabilities of failure for the three architectures.

It must be mentioned that the values chosen for parameters simply represent a line in the space of all the possible combinations. Hence, it does not allow to derive any definitive conclusion about dependability of the three architectures in general. However, the example seems to be consistent with some intuitive conclusions. Curves related to the architectures based on SCOP and RB appear very close in the figure although we would remind the reader that the RB architecture needs perfect diagnostic routines. NVP seems to be slightly worse. This is because the architecture employs two more hardware components than SCOP and RB in most cases.

V. Resource Cost and Response Time

In this section we estimate the average resource consumption (i.e., the average number of hardware components required) and response time for each execution of a given architecture. For each execution of an architecture, when it is invoked, a timing constraint τ is set. The necessary computing nodes are first required to the system, then the proper software must be loaded on them before execution takes place. In particular, RB and SCOP architectures (which are organised in phases) request the processors in a strictly necessary manner — further hardware components will be requested only when the need arises. In the case of the NVP architecture, all the processors must be obtained at the start since the execution is purely parallel. We assume that the time necessary for obtaining the processors and loading the software is an independent and exponentially distributed random variable W_i with parameter λ_w and the execution times are independent and exponentially distributed random variables E_i with parameter λ_i, for the different combinations of variant/processor, and E_d with parameter λ_d, for the adjudicator.

5.1. Average resource consumption

NVP((1, 2, 1), 3, 4, 4) is not organised in phases — it executes its variants in parallel. Therefore it has a constant resource consumption which is equal to four, i.e.,

$$AV.RES_{NVP} = 4$$

The SCOP architecture may require two phases. The execution completes at the end of the first phase (paths 1 and 3 in Figure 4) with probability p_{1SCOP}, while it includes phase 2 (paths 2, 4 and 5) with probability 1-p_{1SCOP}. From the dependability evaluation of the architecture and using the same set of intermediate parameters we have

$$p_{1SCOP} = p_I(1-q_d)p_{II} + (q_{2v} + q_{3v})(1-q_d)p_{II} + 2q_{vd}.$$

The average number of resources required in one execution of **SCOP**((1, 2, 1), 3, 2, 4) thus becomes

$$AV.RES_{SCOP} = 2 + 2(1 - p_{1SCOP}).$$

The same argument applies to the RB architecture; its execution may consist of up to two phases. From the scheme of operation and the dependability analysis we obtain the probability p_{1RB} of stopping at the end of the first phase (it includes also the case where it is necessary to run diagnostic routines):

$$p_{1RB} = (1-q_a)((1-q_p-q_{ps}-q_{pa}-q_{psa})p_{II} + 2q_hq_p(1-q_h)(1-c_{AT})) + q_{pa} + q_{psa}.$$

The average number of resources required in one execution of **RB**((1, 2, 1), 2, 2, 3) thus becomes

$$_{AV.RES}{}_{RB} = 2 + (1 - p_{1RB}).$$

The average number of processors required by adaptive architectures based on SCOP and RB is a function of the probability that the architectures stop in the ith phase, which depends, in its turn, on the dependability parameters. So this must be numerically evaluated by assigning values to those parameters. Although the probability of common-mode failures is the main factor in success or failure of each individual execution, its influence on the number of required phases seem to be relatively weak. In Figure 8 we show the plot of the average number of processors required by the SCOP and RB architectures (continuous and dashed lines respectively) during each execution, which is regarded as a function of the probability of independent failures. The values assigned to the parameters are those already used in the example of dependability evaluation in the previous section.

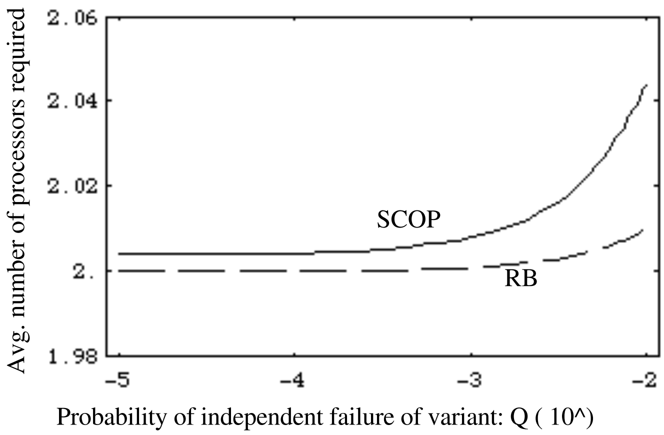

Figure 8. Average number of processors required vs. failure probability.

The realism of such kind of evaluation depends on realistic values and ranges which must be derived for each individual realisation. However, for most plausible values the probability that SCOP or RB stops at the end of the first phase is very high. This means that the average number of hardware components requested by these adaptive architectures is almost constant and very close to the number required to start with.

5.2. Response time

To analyse response time we follow the same approach used in [15, 16]. It is sufficient to designate Y_c the duration of an execution of the redundant component if the watchdog timer is absent. To compare the architectures, we derive the distribution of Y_c and its mean μ. The probability p_{bt} that an execution violates the timing constraint (that is, Y_c exceeds τ) can provide further information.

We first focus on the SCOP architecture and show how these quantities can be computed, then we will just define Y_c for the other architectures. We designate Y_{w1} the time necessary for obtaining two processors in the first phase, $Y_{w1} = \max\{W_1, W_2\}$, Y_{E1} the time for executing variants 1 and 2, $Y_{E1} = \max\{E_1, E_2\}$, similarly $Y_{w2} = \max\{W_3, W_4\}$ and $Y_{E2} = \max\{E_3, E_4\}$. Remembering that p_{1SCOP} represents the probability of completing execution at the end of the first phase, the execution time Y_c without the watchdog timer is:

$$Y_c = \begin{cases} Y_{c1} = Y_{W1} + Y_{E1} + Y_d = \max\{W_1, W_2\} + \max\{E_1, E_2\} + Y_d, & \text{with probability } p_{1SCOP} \\ Y_{c2} = Y_{W1} + Y_{E1} + Y_d + Y_{W2} + Y_{E2} + Y_d, & \text{with probability } (1 - p_{1SCOP}) \end{cases}$$

The probability density function of Y_c is a weighted sum of the probability density functions for the two expressions above. In these expressions, the random variables Y_{w1}, Y_{E1}, Y_{w2} and Y_{E2} are not exponentially distributed. Their cumulative distribution function can be easily obtained. For Y_{w1} it is:

$$G_{Y_{W1}}(y) = \begin{cases} (1 - e^{-y\lambda_w})\,(1 - e^{-y\lambda_w}), & \text{if } y \geq 0 \\ 0, & \text{otherwise} \end{cases}$$

Thus, we can first compute (through convolutions and summations) the probability density function of Y_c and then the probability p_{bt} that an execution violates the timing constraint τ for some values of τ, $p_{bt} = 1 - \int_0^{\tau} f_{Y_c}(y)dy$.

The NVP architecture requires four processors; when all of them are available, the variants are performed in parallel before the adjudication takes place. Based on the same setting as that for the SCOP architecture, we designate Y_w the time for obtaining the four necessary processors, $Y_{w1} = \max\{W_1, W_2, W_3, W_4\}$, Y_E the time for executing the variants, $Y_E = \max\{E_1, E_2, E_3, E_4\}$. The execution time Y_c without the watchdog timer is:

$$Y_c = Y_W + Y_E + Y_d = \max\{W_1, W_2, W_3, W_4\} + \max\{E_1, E_2, E_3, E_4\} + Y_d.$$

The RB architecture requires also diagnostic routines to check hardware failures; therefore additional parameters λ_{Di} are used to represent the mean of the exponentially distributed random variables Di, each of which accounts for the time necessary for running the diagnostic routine on a hardware processor. We designate Y_w the time for obtaining the two processors necessary in the first phase, $Y_w = \max\{W_1, W_2\}$, Y_E the time for executing the primary on them, $Y_E = \max\{E_1, E_2\}$, then $Y_2 = \{W_3 + E_3 + Y_d\}$ the time for acquiring a third processor and for running a variant and an acceptance test on it, and $Y_D = \max\{D_1, D_2\}$ the time spent in running diagnostic routines. According to the operation scheme of this architecture, four different situations may arise:

(1) the scheme stops after running the basic step: the primary and the acceptance test;

(2) after step (1) the diagnostic routine is run (with probability p_1);

(3) after step (2) a third processor is required to run a variant and AT (with probability p_3);

(4) after step (1) a third processor is required, and a variant and AT run on it (with probability p_2).

The probabilities of these events may be derived from the dependability analysis, and the execution time Y_c without the watchdog timer becomes:

$$Y_c = Y_W + Y_E + Y_d + p_1(Y_D + p_3(Y_2)) + p_2(Y_2)$$

In order to compare the response times offered by the three architectures we proceed to assign values to the timing parameters. As we did previously, we attempt to assign reasonable values, as shown in Table 5, to the mean execution time of the variants, of the diagnostic routine for RB and of the adjudicators while regarding the mean of the time necessary for obtaining the processors ready to be used (accounting also for loading the software) as variable and ranging from one tenth to ten times of that of the variant execution. Again the aim is to show how the evaluation is carried out. We fixed the probability that SCOP or RB stops at some phase which corresponds to the probability of independent failures of the software variants 10^{-4}.

Recovery Blocks	N-Version Programming	SCOP
$\lambda_p = 1/5$	$\lambda_1 = 1/5$	$\lambda_1 = 1/5$
$\lambda_s = 1/5$	$\lambda_2 = 1/5$	$\lambda_2 = 1/5$
$\lambda_d = 1/5$	$\lambda_3 = 1/5$	$\lambda_3 = 1/5$
$\lambda_D = 1/5$	$\lambda_4 = 1/5$	$\lambda_4 = 1/5$
$\lambda_W = 1/50$, 1/5 and 2	$\lambda_d = 2$	$\lambda_d = 2$
	$\lambda_W = 1/50$, 1/5 and 2	$\lambda_W = 1/50$, 1/5 and 2

Table 5: Values of the "timing" parameters used.

As results deriving from such setting we show in Figure 9 the plots of the pdf Y_c, in the case where the watchdog timer is absent, for three values of λ_W = 1/50, 1/5 and 2 respectively.

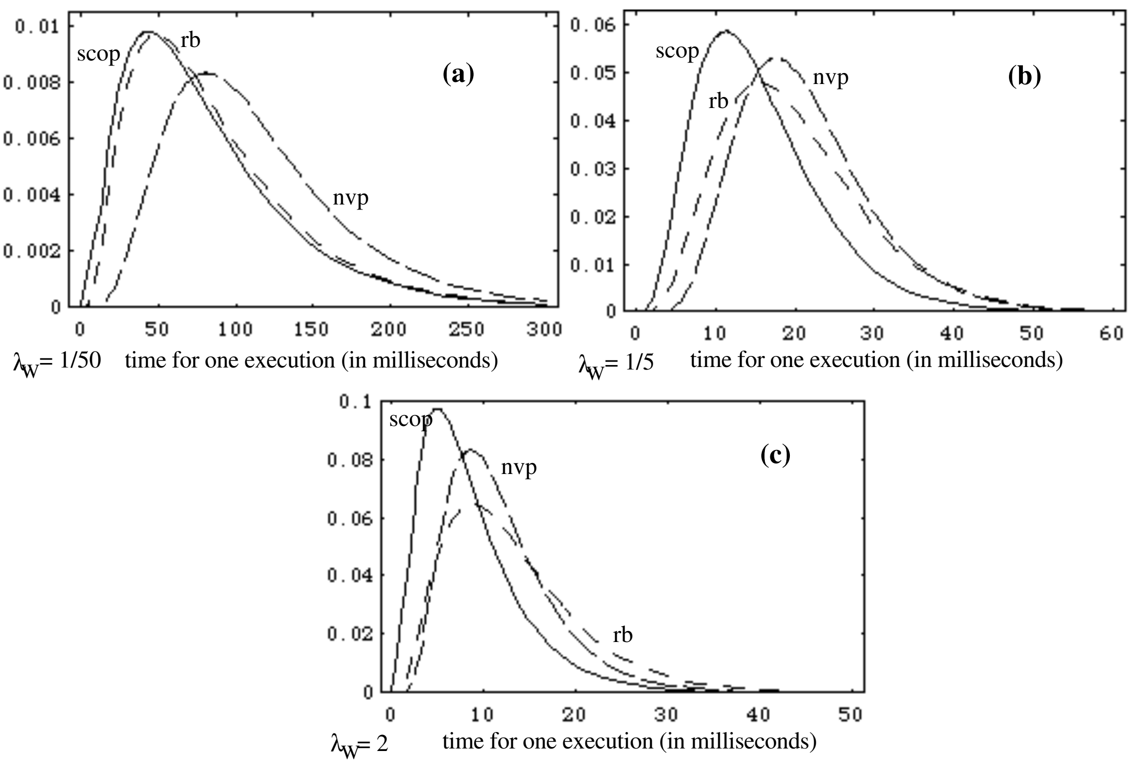

Figure 9. Distribution of Y_c (a): λ_W = 1/50, (b): λ_W = 1/5 and (c): λ_W = 2.

For three different values of λ_W, Table 6 shows the mean response time and the probabilities P_{bt} of exceeding the timing constraints when $\tau = 30$, 50 and 70 milliseconds, with respect to the three architectures under consideration.

		R B	NVP	SCOP
$\lambda_W = 1/50$	Mean of Yc	87.5211	115.083	83.1825
	P_{bt} ($\tau = 30$)	0.904756	0.986965	0.87031
	P_{bt} ($\tau = 50$)	0.721392	0.910135	0.678098
	P_{bt} ($\tau = 70$)	0.5355256	0.767781	0.497436
$\lambda_W = 1/5$	Mean of Yc	20.0165	21.3333	15.5341
	P_{bt} ($\tau = 30$)	0.1391803	0.144972	0.0549317
	P_{bt} ($\tau = 50$)	0.0076969	0.00583087	0.00192367
	P_{bt} ($\tau = 70$)	0.000290	0.000165896	0.000056007
$\lambda_W = 2$	Mean of Yc	13.26604	11.9583	8.76924
	P_{bt} ($\tau = 30$)	0.034107	0.0135991	0.0065822
	P_{bt} ($\tau = 50$)	0.001061	0.000250371	0.000123435
	P_{bt} ($\tau = 70$)	0.000027	0.00000458613	0.00000231025

Table 6: Some results of the timing evaluation.

VI. Conclusions

In this paper, the issue of hybrid fault tolerance in distributed computing environments has been addressed. A set of new architectures for tolerating both hardware and software faults have been defined and evaluated under different aspects including dependability, resource consumption and response time. Architectures based on the RB and SCOP schemes show the ability to adapt their execution to the manifestation of faults so as to minimize hardware resource consumption and to shorten the response time as much as possible. In almost all the cases their performance figures are much better than those of the architectures with static resource consumption. This obviously makes the first two kinds of architectures preferable to the other one while considering the competing applications in distributed environments. However, due to many dynamic factors of the environment under consideration, it is difficult to derive definitive conclusions about dependability and efficiency of the adaptive and non-adaptive architectures in the general case. For example, architectures based on the NVP approach could appear to be faster than those adaptive architectures as they always execute within one phase, especially in the manifestation of faults. But, depending on the underlying system, adaptive architectures, in particular the SCOP ones, would be better as to the mean response time, as shown in the examples of section V.

References

[1] L. Svobodova, “Attaining Resilience in Distributed Systems,” in *Dependability of Resilient Computers*, Oxford: BSP Professional Books, 1990, pp. 98-124.

[2] T. Anderson (ed.), *Resilient Computing Systems*, London, UK: Collins Professional and Technical Books, 1985.

[3] A. Avizienis and J.-C. Laprie, “Dependable Computing: From Concepts to Design Diversity,” *Proc. IEEE*, vol. 74, pp. 629-638, 1986.

[4] T. Anderson, “A Structured Decision Mechanism for Diverse Software,” in *Proc. 5th Symposium on Reliability in Distributed Software and Data Base Systems*, 1986, pp. 125-129.

[5] B. Randell, “System Structure for Software Fault Tolerance,” *IEEE TSE*, vol. SE-1, pp. 220-232, 1975.

[6] A. Avizienis and L. Chen, “On the Implementation of N-Version Programming for Software Fault Tolerance During Program Execution,” in Proc. *COMPSAC 77*, 1977, pp. 149-155.

[7] J.-C. Laprie, J. Arlat, C. Beounes, K. Kanoun and C. Hourtolle, "Hardware and Software Fault-Tolerance: Definition and Analysis of Architectural Solutions," in *Proc. l7th. IEEE International Symposium on Fault-Tolerant Computing*, 1987, pp. ll6-121.

[8] A. Bondavalli, F. Di Giandomenico and J. Xu, "A Cost Effective and Flexible Scheme for Software Fault Tolerance," Computing Laboratory, University of Newcastle upon Tyne, Technical Report 372, February 1992.

[9] A. Bondavalli, F. Di Giandomenico and J. Xu, "Cost-Effective and Flexible Scheme for Software Fault Tolerance," *Journal of Computer Systems Science and Engineering*, vol. 8, pp. 234-244, CRL Publishing Ltd. 1993.

[10] F. Cristian, "Understanding Fault-Tolerant Distributed Systems," *CACM*, vol. 34, pp. 56-78, 1991.

[11] S.K. Shrivastava, G.N. Dixon and G.D. Parrington, "An Overview of the Arjuna Distributed Programming System," *IEEE Software*, vol. 8, pp. 66-73, 1991.

[12] O. Babaoglu, L. Alvisi, A. Amoroso and R. Davoli, "Paralex: an Environment for Reliable Parallel Programming in Distributed Systems," ESPRIT PDCS Technical Report, 1991.

[13] A. Avizienis and J.P.J. Kelly, "Fault Tolerance by Design Diversity: Concepts and Experiments," *IEEE Computer*, vol. 17, pp. 67-80, August 1984.

[14] J.C. Laprie, J. Arlat, C. Beounes and K. Kanoun, "Definition and Analysis of Hardware-and-Software Fault-Tolerant Architectures," *IEEE Computer*, vol. 23, pp. 39-51, (Special Issue on Fault Tolerant Systems) 1990.

[15] A. T. Tai, A. Avizienis and J. F. Meyer, "Performability Enhancement of Fault-Tolerant Software," *IEEE Trans. on Reliability,* vol. 42, pp. 227-237, 1993.

[16] S. Chiaradonna, A. Bondavalli and L. Strigini, "On Performability Modeling and Evaluation of Software Fault Tolerance Structures," in *Proc. 1st European Dependable Computing Conference (E D C C - 1)*, 1994.