Managing Redundancy in CAN-based Networks Supporting N-Version Programming

Julián Proenza\textsuperscript{a,}\textsuperscript{*}, José Miro-Julia\textsuperscript{a}, Hans Hansson\textsuperscript{b}

\textsuperscript{a}Universitat de les Illes Balears, Departament de Matemàtiques i Informàtica, Campus Universitari, 07122 Palma de Mallorca, Spain

\textsuperscript{b}Mälardalen University, Mälardalen Real-Time Research Centre, S-721 23 Västerås, Sweden

Abstract

Software is a major source of reliability degradation in dependable systems. One of the classical remedies is to provide software fault-tolerance by using N-Version Programming (NVP). However, due to requirements on special hardware and the need for changes and additions at all levels of the system, NVP solutions are costly, and have only been used in special cases.

In a previous work, a low-cost architecture for NVP execution was developed. The key features of this architecture are the use of off-the-shelf components and that the fault-tolerance functionality, including voting, error detection, fault-masking, consistency management, and recovery, is moved into a separate redundancy management circuitry (one for each redundant computing node).

In this article we present an improved design of that architecture, specifically resolving some potential inconsistencies that were not treated in detail in the original design. In particular, we present novel techniques for enforcing replica determinism and a method for reintegration of the redundancy management circuitry after a transient failure.

Our improved architecture is based on using the Controller Area Network (CAN). This has several benefits, including low-cost, and that the CAN data consistency allows us to simplify the mechanisms for replica determinism and reintegration.

Although initially developed for NVP, our redundancy management circuitry also supports other software replication techniques, such as active replication.

Key words: Fault tolerance, software fault tolerance, system analysis and design, distributed computing, redundant systems, error compensation, forward error correction, replica determinism, CAN protocol, N-Version Programming.
1 Introduction

Software faults are widely accepted as one of the most important sources of unreliability in computer systems. Their effects can be so negative that there is a renewed interest in evaluating software risks [1]. Therefore design of systems which provide tolerance to software faults for critical applications is an important topic. A series of recent related projects [2–4] called DISCS, DISPO, DISPO-2 and DOTS have addressed the issue of evaluating the dependability provided by techniques for tolerance of software faults. Nevertheless the design of complete systems which are able to tolerate software faults is, perhaps, not receiving as much attention nowadays as the importance of software faults would suggest. One of the reasons is the high cost of that kind of systems.

Commercially available fault-tolerant systems often use application-specific multiprocessor architectures [5–7]. Their design and manufacturing costs have a strong impact on the final price and discourage potential buyers. When tolerance to software faults is also required, cost is further increased by the development of redundant —and diverse— application software.

According to [8], the reason fault-tolerant systems are not so widely used as the interest on the subject would suggest is that the fault-tolerant mechanisms are not orthogonal to the other functionalities of the system, in the sense that essentially all system components must be adapted to handle the fault-tolerance. This makes it difficult to use low-cost commercial components in the design of a fault-tolerant system. Therefore, the development in the fault tolerance aspects may even impede development in other aspects. As a final result, the cost of a fault-tolerant system is much higher than the cost of a non-fault-tolerant system with equivalent performance, even if the cost of redundancy is not taken into account.

In an attempt to solve this problem, Miro-Julia [9] proposed a low-cost architecture for the execution of applications which tolerate software faults following the N-Version Programming (NVP) paradigm [10]. In NVP, $N$ diverse versions of the same program are developed by independent teams. Each version is partitioned into a set of segments. Corresponding segments in different versions are intended to perform the same function. In execution, each time a version finishes a segment, it issues a vector of results of this segment, called $cc$-vector (see Figure 1). Then a decision algorithm is executed on to obtain a consensus $cc$-vector which is sent back to all versions to be used in the continued computation. This mechanism, called $cc$-point, provides both synchronization among versions and masking of faults in a minority of versions.

* Corresponding author.

Email addresses: julian.proenza@uib.es (Julián Proenza), joe.miro@uib.es (José Miro-Julia), hans.hansson@mdh.se (Hans Hansson).
NVP has received some criticisms in the past, particularly stating that the usual hypothesis in redundant systems that says that replicated components exhibit fault independence does not hold for the different versions in NVP [11,12]. Nevertheless it is accepted nowadays [13] that using NVP is “on average” more reliable than using a single software version. What is still an open issue is what is the actual gain in reliability that NVP provides [13]. In any case the final reliability gain depends on the way software versions are developed and this issue is beyond the scope of the present paper, which is focussed on proposing an architecture to execute the versions once they have been properly developed.

In the low-cost architecture proposed in [9], which is depicted in Figure 2, the $N$ versions are executed, each one on a different computer. Moreover, each time one of the versions finishes one of the segments, the corresponding cc-vector is immediately broadcast through a broadcast network equipped with a special hardware unit, called $N$-Version Executive Processor (NVXP). One NVXP is attached to each computer to manage the mentioned communication, as well as other key functions for fault-tolerance support. In particular, each NVXP, independently of the application and host processor, executes the decision algorithm on the cc-vectors of all versions and returns the consensus cc-vector to the local version. In this architecture the transmission of cc-vectors among NVXPs is triggered each time one of the versions finishes a segment, which corresponds to an event-triggered communication scheme. This is the basis of the Event Synchronous System (ESYS) approach, proposed in [14] as the most suitable way of executing applications which follow the NVP paradigm, given the diverse execution times which are to be expected from diverse software versions.
This architecture has several desirable properties. First, the single point of failure that a single voter would represent is eliminated. Second, the mechanisms for fault tolerance are concentrated in the added NVXP and are thereby orthogonal to the other functionalities of the system. And third, hardware Components Off The Self (COTS) are used as main computers and as building blocks for the implementation of the NVXP. However the specific design proposed in [9] for this distributed architecture is difficult to implement in practice and presents some scenarios of inconsistency among operations that are replicated in our architecture, e.g., the votings performed by different NVXPs may yield different results, which may cause even non-faulty versions to diverge, since they use different input in the subsequent executions.

In this paper we present a new architecture that we have devised in order to eliminate the scenarios of inconsistency mentioned above. Our architecture takes the one introduced in [9] as starting point. We have made the following three main changes in the hardware of this architecture. First, we have developed a completely new and improved design for the NVXPs, which we call Redundancy and Communication Management Boards (RCMBs). Second, in order to satisfy our low cost requirement, we have chosen PCs as platforms, i.e., the RCMBs are PC boards inserted in the bus of the host PCs (where the versions are executing). And third, we use the Controller Area Network (CAN) protocol [15] as the basic communication technology for the broadcast network, due to its well-known advantages related to cost, reliability and real-time performance, and due to the growing interest of using CAN for critical applications [16].

Note that for the rest of this paper we shall consider that a node of this architecture is any ensemble constituted by a PC and the RCMB which is directly attached to it.

Besides introducing changes in the hardware, we have designed a new software to be executed in the RCMBs. This software will be responsible for the consistent management of the redundancy in this architecture. Two issues related to the consistent management of the redundancy constitute the main focus of this research: replica determinism enforcement [17] of all replicated operations and consistent reintegration of RCMBs after transient faults. Replica determinism enforcement ensures, for instance, that all non-faulty replicas of the voting procedure executed by the different RCMBs produce the same consensus cc-vector. Reintegration allows an RCMB that has been disconnected in order to prevent error propagation caused by transient faults to again be integrated in the system. The purpose of reintegration is to make sure that the redundancy of the system does not permanently attrite (degrade) when RCMBs are being disconnected due to transient faults. It should be noted that mechanisms for reintegration of versions and PCs that have suffered a transient fault are provided by NVP itself. Indeed given that in the cc-points described above all
versions receive the resulting consensus cc-vector, not only fault masking is
achieved, but also versions which have issued a wrong cc-vector—e.g. because
of a transient fault in the corresponding computer—have an opportunity to
recover using the consensus cc-vector values to resume computation. In con-
trast, the reintegration of RCMBs affected by transient faults is a new problem
that has to be solved by our architecture. Due to space limitations, how we
have achieved this reintegration is not described in this paper. A thorough
description can be found in [18], where also the implementation of our entire
architecture is described.

Besides low cost, real implementability and consistency, our design has another
important requirement; the management of redundancy must be achieved
without introducing a significant computation or communication overhead in
the system. This is particularly important in the context of NVP, since the
performance of such applications strongly depends on the time they have to
be stopped waiting for the voting results.

In the next section a description of the basic features of our architecture is
provided. Section 3 is devoted to our strategy to enforce replica determin-
ism. Section 4 compares our approach with some related work, and Section 5
concludes the paper.

2 Basic features of our architecture

In this section we describe the basic features of our architectural infrastruc-
ture, on top of which we shall build our mechanisms for replica determinism
enforcement and consistent reintegration of faulty RCMBs. These features
are related to the definition of the error containment boundaries and to the
organization of the fault-tolerance operations.

2.1 Definition of the error containment boundaries

To ensure fault independence among replicated components, it is essential to
prevent errors of the different nodes from propagating [19]. This is the purpose
of the error containment boundaries.

We have decided to build these boundaries by restricting the failure semantics
of each node and by preparing the other nodes to deal with the errors that even
with a restricted failure semantics may occur. Restricting the failure semantics
of a node simplifies the operations that other nodes have to perform to deal
with potential errors. We restrict the failure semantics of nodes by letting the
RCMB perform self-checking, and policing of the attached PC.

When the self-checking circuitry detects an error, the RCMB is disconnected from the network; providing a crash failure semantics [17] for the RCMBs (i.e., the RCMBs can be considered to either function correctly or being silent).

As indicated above, RCMBs also police the PCs and the versions they execute. This is done by monitoring the operation of the PC to which it is attached. More specifically, each RCMB polices the messages that its PC wants to transmit to the rest of the nodes. By this it is possible to prevent the PC from broadcasting messages which may constitute either babbling idiot failures [20], two-faced behaviour or impersonations of other PCs [17]. The RCMBs only allow a single cc-vector per segment to be sent by their attached PC, thereby avoiding babbling idiot behaviour from the PC. Likewise, the RCMB sends all the cc-vector messages from its attached PC in a broadcast mode in order to eliminate any chance of a PC sending different messages to different nodes. Finally, the RCMB actually indicates in the cc-vector messages which is the identity of the corresponding PC-version, thereby eliminating the possibility of a faulty PC-version impersonating other nodes. This policing gives what is called an incorrect computation failure semantics [21], meaning that PCs and versions may only fail by delivering incorrect results either in the time domain or in the value domain.

In fact, the above described failure semantics of both RCMB and PC are achieved thanks not only to the local mechanisms which have been described, but also thanks to the properties of the communication protocol which has been chosen for the broadcast network. Indeed a wrongly chosen protocol may generate in the system the same kind of failures we are trying to prevent.

Therefore we need a communication protocol providing reliable broadcast [22]. Apart from providing reliable broadcast, we also need our protocol to be able to prevent a babbling idiot behaviour generated by the channel itself. Some channels are designed to detect communication errors and to tolerate them by retransmitting the affected frames. If the conditions producing the errors are permanent and nothing is done, the channel may be blocked by retransmissions as in a babbling idiot scenario.

The above indicated requirements have lead us to choose the Controller Area Network (CAN) protocol [15]. CAN provides reliable broadcast. In fact, according to the protocol specification [15], what CAN provides is data consistency, which roughly corresponds to the reliable broadcast’s list of properties plus the total order property [22]. Taking into account the definitions provided in [22], this means that CAN is supposed to provide atomic broadcast.\footnote{Even though CAN is often supposed to provide atomic broadcast, in fact it presents some error scenarios for which this supposition is not valid [23,24]. In}
CAN is able to prevent babbling idiot behaviour at the channel since each CAN controller (i.e. the circuit which implements the CAN protocol in each node of a CAN network) counts its errors and disconnects itself if the count surpasses a threshold.

The CAN protocol is designed for a bus topology. In fault-tolerant distributed systems it is customary to use other kinds of topologies in which some redundancy appears at the communication link level [25]. This redundancy is used to prevent the effects on the rest of the system of arbitrary failures of a single node, such as babbling idiots or impersonations. For instance, topologies in which not all nodes share the same transmission channel are used to prevent a babbling node from monopolizing the whole network. Similarly, interconnection topologies that provide the identity of the source of a message are used to prevent a faulty node from impersonating a non-faulty one. This kind of approach leads to topologies having a large number of redundant communication links, which causes high cost and also lack of extensibility, since we need to add a complete set of new links for each new node. In our architecture, we can use the bus topology, which is intrinsic to the CAN protocol, because of the restriction of the failure semantics we enforce in the nodes that prevents both babbling idiots and impersonations. The use of the bus topology represents a significant advantage of our architecture, since it reduces the cost of the communication hardware and facilitates the extensibility of the network. However the adoption of a bus topology for the broadcast network introduces an additional single point of failure in the system that has to be eliminated by including bus redundancy. We have adopted the approach to bus redundancy for CAN networks presented in [26], since it is completely compatible with the rest of our architecture. In the bus redundancy scheme proposed in [26], redundancy is introduced at the communication media level (i.e. transceivers and cables) and in each node a redundancy management circuit has been added. Any frame is transmitted simultaneously through all redundant transceivers and cables, and the redundancy manager in each node makes this media redundancy transparent to the rest of the system disconnecting the media that are diagnosed as faulty. Therefore all CAN nodes see a single logical channel as if there was no redundancy at all.

Our approach of providing consistent communication services at the lowest level of the system is fundamental to fulfill the requirement of reducing the communication and computation overhead. Furthermore it also offers the advantage of avoiding some potential amplification of communication.

In a previous work we have studied this problem and proposed a solution, which consists on a modification to the CAN protocol, called MajorCAN [24]. As MajorCAN is a simple modification to CAN which is fully compatible with the rest of our architecture, we shall hereafter talk about CAN as if it actually provided the atomic broadcast service.
failures [22]. To see why, consider the alternative of using a high-level im-
214plemented communication service, e.g. an Atomic Broadcast protocol, which
215is designed to be used on top of a non-reliable communication channel link-
216ing nodes which may exhibit arbitrary failures. This service requires several
217rounds of message transmissions, e.g. the broadcast of a message will require
218several transmissions and execution of several operations in all nodes, such as
219the transmission of ACK messages. In this kind of complex communication
220schemes, a failure at the low level (e.g. an omission to send a message) does
221not necessarily manifest at the high level as the same type of failure (e.g. an
222omission to broadcast a message to all receivers). In fact, this kind of broadcast
223algorithms are likely to amplify the importance of failures that occur at the
224low level [22] (e.g. messages delivered to different receivers in a non consistent
225order due to an omission to send a message).

Beyond avoiding amplification of communication failures, our approach also
228presents the significant advantage of reducing the number of required nodes. In
229general, the number of required nodes for a communication protocol to provide
230a specific service varies depending on the failure semantics of the nodes [17]. To
231reach agreement under byzantine (i.e. unrestricted) failure assumptions for the
232nodes it is necessary to have \( n \geq 3t + 1 \) nodes (where \( n \) is the number of nodes
233and \( t \) is the maximum number of faulty nodes permitted), whereas for simple
234majority voting, once all values have been consistently exchanged, \( n \geq 2t + 1 \)
235nodes are sufficient. Since NVP is based on majority voting, \( n = 2t + 1 \) should
236be sufficient, each node executing a different version (therefore \( N = n \)). But,
237under byzantine failure assumptions this bound is no longer valid. However,
238by designing the RCMBs to present a restricted failure semantics, which at
239the same time restricts the failure semantics of the PCs, and by guaranteeing
240consistent communications, we ensure that \( n = N = 2t + 1 \) nodes is enough.
241This reduction of nodes significantly reduces the cost of the system not only
242at the hardware level, but above all, at the software level, since it reduces the
243number of diverse software versions to be developed. Furthermore, reducing
244the number of nodes increases the dependability of the system, particularly
245in terms of the reliability. It is clear that if we reduce the number of nodes
246required to tolerate the failure of some of them, the complexity of the system
247decreases and the probability of a correct operation increases.

So far, we have just indicated how we restrict the failure semantics of the
249nodes. However, these restrictions are not enough to completely define the
250error containment boundaries. It is also necessary to prepare each node to
251properly deal with errors in the output (erroneous values or omissions) that
252still may be issued by other nodes. In fact the responsible for dealing with
253these errors is the RCMB of each node. So, for each message received, the
254RCMB has to determine if it originates from the RCMB or from the PC of
255the transmitting node. If the message originates from the RCMB, it has to
256be assumed to be correct, since RCMBs exhibit crash failure semantics. If the

8
message originates from the PC, some more elaborated processing has to be performed, since both PCs and the versions executed on them exhibit incorrect computation failure semantics (i.e., upon failure they may generate incorrect output, both in the value and in the time domains). Voting is the mechanism used by NVP to handle computation failures. Obviously this can only be done if results are available from enough number of nodes. In any case, such a voting should take into account that the messages may be incorrect also in the time domain [21] as will be seen in Section 3.

### 2.2 RCMB Hardware Design

Our architecture is based on the assumption that nodes exhibit the restricted failure semantics indicated above. Therefore, the effectiveness of the entire design and thus the level of reliability which is finally reached, dramatically depend on the coverage of our failure semantics assumptions, and thus on the RCMB design.

This assumption coverage [27] strongly depends, in our architecture, on the fault detection coverage of the RCMBs. The higher the latter is, the higher the former will be. In order to reach a high fault detection coverage we have chosen duplication with comparison [28] as a technique for error detection in the RCMBs. This technique is considered very effective, and we have applied it extensively within the RCMB structure; most of the circuits of each RCMB are duplicated and compared. Figure 3 shows the basic structure of a prototype for the RCMB which we have implemented. As can be seen in the figure all fundamental circuits have been duplicated and all frames are sent through a single logical channel (identifiable in signals Tx and Rx) which exhibits redundancy at the communication media as described in Section 2.1. A Main Comparator (MC, in the figure) is responsible for comparing the address and data signals which are issued by each circuit with those issued by its duplicate. A specific CAN Comparator has been developed in order to compare the bit streams issued by each replica of the CAN controller. If those streams match, the corresponding bits are sent simultaneously through the redundant communication media which, as indicated in Section 2.1, have their own redundancy management circuit to disconnect those media that are diagnosed as faulty. When any of these two comparators detect a discrepancy between duplicates they disconnect the RCMB from the CAN bus and interrupt the duplicated processor. More details on the hardware structure of this prototype can be found in [18].
2.3 Organization of the fault-tolerance operations

As has been already indicated, the fundamental operation that our architecture performs in order to achieve fault tolerance is the voting on the cc-vectors issued by versions at the end of each segment. This voting is executed at each RCMB and provides error compensation [29] (i.e. fault masking).

To improve the global dependability, the RCMBs perform the following three additional fault-tolerance operations; first, error detection of individual nodes, second, fault passivation [29], and third, recovery of components that have suffered transient faults.

Error detection of individual nodes is obtained by comparing the consensus cc-vector calculated by the voting with the cc-vectors sent by each node. Additional error detection is provided by checking the reception of specific messages. As the omission of a cc-vector may be diagnosed as caused by either a faulty version or a faulty RCMB, RCMBs must send an “I am alive” message at the beginning of each round of cc-vector exchange. If the “I am alive” message of an RCMB is missing, the fault is attributed to the RCMB, whereas if the “I am alive” message is received, the fault is attributed to the version.

Fault passivation is performed by disconnecting the components that are affected by faults. To prevent a quick attrition of redundancy, disconnection should not be permanent for components which are affected by transient faults. Therefore each RCMB maintains an error counter for each version and for each RCMB. The corresponding error counter is increased each time a new error is detected, and decreased when no error is detected. Only when an error counter reaches a prespecified threshold is the corresponding component considered permanently faulty and disconnected from the rest of the system. At any instant, the values of all these counters are considered to represent the status of the system.

Recovery is provided by specific mechanisms for recovering the RCMBs. Note
that, as indicated in Section 1, NVP already provides mechanisms, such as the
cc-points, for the recovery of transiently faulty versions which have issued a
cc-vector containing errors. In fact NVP also provides a mechanism which can
recover versions which have exhibit more severe errors, e.g., having executed the
wrong segment. This mechanism is called recovery point [10] and it is basically
a cc-point in which the complete state of the computation is exchanged and
voted. This allows the recovery of versions whose internal state (i.e. memory)
has been corrupted.

In contrast to faults in versions, faults in RCMBs always manifest themselves
as omissions. These omissions are caused by the disconnection from the net-
work in the event of an error detected by the duplicated and compared struc-
ture of the RCMB. The problem is that any of these omissions will become
a permanent omission (crash) unless specific recovery actions are performed
for the faulty RCMB. Therefore, in the absence of a recovery mechanism any
transient fault in an RCMB becomes permanent and the quick attrition of
redundancy that we purported to prevent by using error counters takes place
anyway.

During the disconnection from the network, the faulty RCMB may miss some
messages and thus may lose the consistency with the remaining RCMBs.
Therefore, the proper recovery of RCMBs affected by transient faults requires
the resynchronization with the rest of the RCMBs. We call this process rein-
tegration of faulty RCMBs. An important requirement of the reintegra-
tion process is that it has to lead the affected RCMB to a state which is consistent
with all non-faulty RCMBs. How we perform this is described in [18].

3 Replica determinism enforcement

As indicated in Section 1, the aspect of consistency that this paper focuses on
is the replica determinism enforcement [17] of all components that have been
replicated for fault-tolerance purposes.

Roughly speaking, we can say that a group of replicas of the same operation
exhibit replica determinism [17] when all non-faulty replicas show correspon-
dence of replica outputs and/or state changes. This definition must be com-
plemented by a correspondency requirement which indicates the meaning of
correspondence for each considered group of replicas.

Three different groups of replicated tasks can be identified: first, the group of
versions which are executed in the various PCs, second, the group of voting
operations performed by the RCMBs, and third, the group of status evalua-
tion operations also performed by the RCMBs. All three sets must be replica
determinate for proper operation of the system. Figure 4 presents one of the replicas of each replicated task and their inputs and outputs.

Fig. 4. The three sets of replicated tasks

Starting with the group of versions, it is clear that all non-faulty versions must generate corresponding outputs, i.e. cc-vectors, if voting on these values has to provide tolerance to faults in a minority of versions. The correspondence requirement in the value domain for the outputs of this group is different depending on the data types. Outputs of the boolean, character and integer types must be identical from all non-faulty replicas of the group, whereas outputs of floating-point—or fixed-point decimals—type are allowed a bounded deviation in their values, since different versions are allowed to use different floating-point algorithms. Versions generating outputs exceeding the permitted deviation are considered faulty and are therefore not included in the replica determinism requirement. The actually permitted deviation is application specific.

In contrast, in the time domain all data types are allowed to have a bounded deviation in their issue time. Again the diversity in the design of the versions allows different execution times for the same segment, and again this deviation must be bounded as the only means for differentiating non-faulty versions from faulty ones. The actually permitted deviation depends on the timing requirements of the specific application.

We make the following three assumptions in our approach to replica determinate the versions. First, the only input data used to generate the next cc-vector are the consensus cc-vectors of the previous votings\(^2\). Second, the code of the versions does not include any non-deterministic programming constructs. And

\(^2\) In control operations, values obtained from replicated sensors are also taken into account by the versions, however voting should be performed on those values before being used, and it has to be ensured that all versions receive the results of this voting consistently.
third, before taking a decision on the basis of information that is only available for the local version or that includes floating-point numbers, which are allowed to have slightly different values in the various nodes, said information is exchanged and voted using cc-vectors.

Under these assumptions, the only source of non-determinism (at least among those pointed out in [17]) for the versions are the potential differences in the results of the votings, i.e. the consensus cc-vector, which the versions receive as inputs from their local RCMBs (see Figure 4).

Therefore, in our architecture, the problem of replica determinism enforcement of the versions can be reduced to enforce replica determinism in the voting operations performed by the RCMBs, provided that the correspondency requirement for the group of replicas of the voting operation is established as: the consensus cc-vectors calculated by all non-faulty replicas of the voting operation for each segment have to exhibit identical values and be issued at almost the same time.

The status evaluation operations performed by the RCMBs have to provide all nodes with a consistent view of the available redundancy in the system. Moreover, in [18] we show that having a consistent status in all nodes is also useful in order to simplify the reintegration of RCMBs affected by transient faults. Therefore, the correspondency requirement for the status evaluation replicas is that the status evaluated by all non-faulty replicas for each segment must be identical and issued at approximately the same time.

3.1 Determinism enforcement of the voting replicas

For enforcing determinism of the voting replicas under the assumption that their code does not include any non-deterministic programming construct, it is enough to ensure that all RCMBs receive the same inputs. As illustrated in Figure 4, these inputs are the cc-vectors each RCMB broadcasts. The problem of providing a group of nodes with the same set of values in which each element is the value of a private information of one of those nodes was first formulated by Pease, Shostak and Lamport [30], and is thereafter called interactive consistency.

In our architecture, there are some features included in order to simplify the achievement of interactive consistency. The way in which we have defined the error containment boundaries in Section 2.1 eliminates the possibility of arbitrary failures of the nodes and guarantees atomic broadcast at the application level. Therefore, any message containing a part of a cc-vector is consistently received by all receivers, and furthermore, is received in the same order by all of them. Note that total order was not required for the communication
protocol to prevent the arbitrary failure scenarios pointed out in Section 2.1. Nevertheless, as CAN exhibits this property, we are going to take advantage of it to achieve consistency in the inputs of the voting processes. Thus, if we ensure that even the transmitter of a message also receives it by using the mode called self-reception, which is available in modern CAN controllers, and if all messages are sent to all RCMBs, then all non-faulty RCMBs will receive the same ordered list of messages. Thus, the only issue which is missed is which portion of this ordered list of messages is taken into account for the voting of each segment.

In order to identify this portion of the ordered list in a consistent manner, we use a message—the start message—to consistently indicate the first message to be taken into account for the voting and another message—the stop message—to do the same with the last message to be taken into account.

Identifying a start message is very simple. It will simply be the first cc-vector message broadcast at the end of the current segment. In contrast, for the stop message things are complicated by the design diversity of the versions and the failure semantics of the RCMBs. Design diversity gives variations in execution times among versions and the failure semantics of the RCMBs may cause omissions, i.e., individual cc-vectors may be transmitted late or not at all.

The possibility of a faulty version never sending its cc-vector was already taken into account in previous developments of architectures for NVP execution. The solution proposed in [9] uses a timeout in each node which is started when a majority of cc-vectors have been received. If the timeout expires without having received all cc-vectors, only those received before the expiration are included in the voting process. However, local timeouts are a typical cause of replica non-determinism [17] when they are used without global coordination.

Our approach is that each RCMB starts the timeout independently when it has received a majority of cc-vectors. We call this timeout TIMEOUT1 to differentiate from other timeouts which are to be described later on. For any RCMB, if either its TIMEOUT1 expires before that RCMB has received the cc-vectors of all nodes, or the cc-vectors of all nodes are received before the TIMEOUT1 of that RCMB expires, the RCMB sends a message identified as a candidate to stop message. The first received among the candidates of all RCMBs is considered as the stop message, and the transmission of the candidates that are pending is aborted.

The ordered list of cc-vector messages received between the start and the stop messages is consistent for all non-faulty RCMBs and is taken by all these circuits as the input values of the voting. With this simple mechanism we fulfill our goal of achieving consistency in the voting process. Moreover, this
consistent voting prevents the propagation of errors from the versions and RCMBs to the rest of the nodes.

3.2 Determinism enforcement of the status evaluation replicas

As illustrated in Figure 4, the status evaluation uses as inputs the status evaluated after the previous segment, the cc-vectors received from the different versions, the result of the voting provided by the local replica of the voting operation, and the “I am alive” messages of the different RCMBs.

Remembering that both status evaluation and voting are performed by the RCMBs and that these circuits exhibit crash failure semantics, we can ensure that all non-faulty replicas of the status evaluation receive the same status from the previous evaluation, as long as we assume this evaluation to have been consistently performed. Likewise, we can ensure that they receive the same result of the voting provided by the local replica of the voting, since replica determinism of this voting has been ensured, as described in the previous section. Moreover, all status evaluation replicas receive the same cc-vector messages since we use the mechanism of start and stop messages described above.

Remains to ensure that all non-faulty replicas of the status evaluation receive a consistent list of “I am alive” messages. This is easy to achieve. We simply design the RCMBs to send their “I am alive” messages immediately after receiving the start message. In this manner each non-faulty RCMB has time to send its “I am alive” before the stop message. Moreover, we can use the same start and stop messages to determine which “I am alive” messages have to be taken into account in the next status evaluation.

Figure 5 summarizes the different mechanisms we have introduced in order to enforce replica determinism of the voting and status evaluation. The figure presents the operations performed by each RCMB in order to agree on the start and stop messages and in order to ensure the consistent reception of “I am alive” messages.

4 Related work

In this section we identify some similarities and differences of our system with other architectures. We shall focus on Delta-4 [25] and GUARDS [31] since all the concepts and techniques related to consistent management of redundancy were already mature at the time these architectures were developed. However,
Fig. 5. The state machine which enforces replica determinism for voting and status evaluation

neither Delta-4 nor GUARDS present specific mechanisms for the execution
of NVP applications. Though both pay special attention to the replica non-
determinism problem. Moreover, GUARDS is one of the most recent complete
designs of its kind, providing insights on the state-of-the-art in this area.

In order to make the comparison fair, it is important to say that both Delta-4
and GUARDS were more focussed on achieving general architectures which
could be used with different standards of data communication. In contrast, our
approach sacrifices this generality in order to take advantage of the specific
features of the CAN protocol.

The main difference between GUARDS and our system is that we concentrate
the redundancy management to a separate circuit (the RCMB). In contrast,
GUARDS introduces relevant features at all levels, including the application.
For instance, to enforce internal replica determinism [17], which is fundamen-
tal to reduce the number of communication rounds needed to exchange the
results for the voting, GUARDS uses a mechanism based on timestamps at
the application level.

Another difference is that we restrict more than GUARDS the failure seman-
tics of the nodes. Therefore, GUARDS has to use a high-level protocol (with
high overhead) to execute the consistent communications (i.e. interactive con-
sistency). Another penalty for restricting less than us the failure semantics of
the nodes in GUARDS is the need for a more complex and expensive network
topology, whereas we can use a simple and low-cost one.
Just as us, Delta-4 is interested in reducing the complexity of the communications, particularly in simplifying the network topology. At this end they introduce the Network Attachment Controller (NAC), which is a circuit attached to each host to take care of the operations related to communication. These devices show that a restricted failure semantics is an effective way of preventing NACs or hosts (which may show unrestricted failure semantics) from blocking the communication channel by sending useless messages. It also makes it impossible for a host to impersonate another host. This justifies the use of a simple network topology similar to ours in Delta-4. However, in contrast with us, Delta-4 does not lean on the consistency of the low-level network technology, since it is designed to be used on standard LAN technology, particularly, token-ring, token-bus and FDDI. For this reason, it purports to provide a rather general high-level communication protocol that can be used on top of any of those technologies.

5 Conclusions

We have described a new architecture for an embedded distributed system that is tolerant to software faults through the execution of NVP applications. Our architecture is based on a previous design which is aimed at providing a low-cost hardware infrastructure by keeping the orthogonality between, on the one hand, the mechanisms which are related to fault tolerance and, on the other hand, the rest of the functionality of the system. Taking this design as our starting point we have studied the issue of consistent management of the redundancy, and we have proposed a new design in order to solve two specific problems; the replica determinism enforcement of all the groups of replicated programs that can be identified in the system, and the reintegration of components that have suffered a transient fault.

Solving the first of these problems must be taken as a requirement for correctness of any distributed fault-tolerant system, whereas the solution of the second one is important to prevent a quick and unnecessary redundancy attrition from happening. Both problems have been solved without introducing a significant computation or communication overhead (in terms of number of messages) in the system.

Our new design is centered in one specific component of the architecture, the Redundancy and Communication Management Board (RCMB). To each computer of the distributed system an RCMBs is attached, providing communication services and playing a central role in all the solutions we propose. In particular, RCMBs are designed to exhibit a restricted failure semantics and they use the CAN protocol for exchanging messages among them. Also, as the RCMBs are the only interface for the computers to communicate with the rest
of the system, the fail-restricted RCMBs are able to ensure that neither the computers nor the versions they execute will reach the network with messages either blocking the channel or impersonating other computers. This means that the restricted failure semantics of the RCMBs serves to restrict the appearance of the failure semantics of the computers from the viewpoint of the other computers. We take advantage of the restricted failure semantics of the nodes and also of the CAN properties in order to simplify communications. More specifically we avoid using complex consistency protocols based on the exchange of multiple messages which are common in distributed fault-tolerant systems.

Our approach provides other significant advantages. Among others, it reduces the number of required nodes and versions, restricts the potential communication failures and simplifies the network topology. Finally, although initially developed for NVP, our architecture also supports other software replication techniques, such as active replication [17]. Active replication is a classical replication technique in which a group of identical copies of the same program are executed in parallel, servicing identical requests. Since all mechanisms included in our architecture are devised to be used in the more general case of executing diverse versions, they can also be used when this diversity is absent. This is a significant advantage of our architecture since active replication is more widely used than NVP because of its lower cost, even though it does not provide the same level of tolerance to software faults.

References


