Redundancy Management in a Low-Cost Distributed Hardware and Firmware Support for Software-Fault Tolerance

Julián Proenza, José Miro-Julia, and Hans Hansson

October 3, 2007

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.

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, 3, 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, 6, 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.

![Figure 1: NVP execution](image)

NVP has received some criticisms in the past, particularly stating that the usual hypothesis in redundant systems that says that replicated components exhibit fault in-
dependence 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 focused 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 per-
formance, 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 man-
agement of the redundancy in this architecture. Two issues related to the consistent
management of the redundancy constitute the main focus of this paper: replica deter-
minism 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 re-
dundancy of the system does not permanently attrite (degrade) when RCMBs are being
disconnected due to transient faults. It should be noted that mechanisms for reinte-
gation 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 result-
ing 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 re-
sume computation. In contrast, the reintegration of RCMBs affected by transient faults
is a new problem that has to be solved by our architecture.

Besides low cost, real implementability and consistency, our design has another
important requirement; the management of redundancy must be achieved without in-
troducing a significant computation or communication overhead in the system. This
is particularly important in the context of NVP, since the performance of such appli-
cations 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 pro-
vided. Section 3 is devoted to our strategy to enforce replica determinism and Section 4
presents our approach to consistent reintegration of RCMBs. Section 5 compares our
approach with some related work. Section 6 discusses some additional features of our
architecture, and in Section 7 we conclude the paper.

2 Basic features of our architecture

In this section we describe the basic features of our architectural infrastructure, on top
of which we shall build our mechanisms for replica determinism enforcement and con-
sistent 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 [18]. This is the purpose of the error containment boundaries.

We have decided to build these boundaries by restricting the failure semantics of the nodes and by preparing the 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 [19], 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 [20], 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 [21]. 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 no special measures are taken, the channel may be blocked 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 [21]. Taking into account the definitions provided in [21], this means that CAN is supposed to provide atomic broadcast\(^1\). Also, CAN is able to prevent babbling idiot behaviour\(^1\).

\(^1\)Even though CAN is often supposed to provide atomic broadcast, in fact it presents some error scenarios
at the channel given that each CAN controller—which is the circuit which implements
the CAN protocol in each node of a CAN network—counts its number of errors and
disconnects itself if the count surpasses a pre-specified threshold.

The CAN protocol is designed for a bus topology. In fault-tolerant distributed sys-
tems it is customary to use other kind of topologies where some redundancy appears
at the communication link level [24]. 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,
impersonations or two-faced behaviours. In our architecture, we can use the bus topol-
yogy, which is intrinsic to the CAN protocol, because of the restriction of the failure
semantics we enforce in the nodes. 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 topol-
yogy 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 [25], since it is completely
compatible with the rest of our architecture. In the bus redundancy scheme proposed
in [25], 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 which are diagnosed as faulty. Therefore all
CAN nodes see a single logical channel as if there was no redundancy at all.

So far, we have just indicated how we restrict the failure semantics of the nodes.
However, these restrictions are not enough to completely define the error containment
boundaries. It is also necessary to prepare each node to properly deal with errors in
the output (erroneous values or omissions) that still may be issued by other nodes. In
fact the responsible for dealing with these errors is the RCMB of each node. So, for
each message received, the RCMB has to determine if it originates from the RCMB
or from the PC of the transmitting node. If the message originates from the RCMB, it
has to be assumed to be correct, since RCMBs exhibit crash failure semantics. If the
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. Of course 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 [20].

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

---

for which this supposition is not valid [22, 23]. In a previous work we have studied this problem and proposed
a solution, which consists on a modification to the CAN protocol, called MajorCAN [23]. 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.
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 [26] 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 level of reliability we have chosen duplication with comparison [27] 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. The actions to be performed by the duplicated processor after this interrupt will be described in Section 4. More details on the hardware structure of this prototype can be found in [28].

![Diagram of duplicated and compared structure of the RCMB](image)

**Figure 3: Duplicated and compared structure of the RCMB**

### 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 the 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 above, 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 network in the event of an error detected by the duplicated and compared structure 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 loose 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 reintegration of faulty RCMBs. An important requirement of the reintegration 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 Section 4.
3 Replica determinism enforcement

As indicated in Section 1, the first aspect of consistency that we shall consider is the replica determinism enforcement \[17\] of all components which have been replicated for fault-tolerance purposes.

Roughly speaking, we can say that a group of replicas of the same operation exhibits replica determinism \[17\] when all non-faulty replicas show correspondence of replica outputs and/or state changes. This definition must be complemented by a correspondence 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 evaluation 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.

Figure 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 per-
mitted 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. Second, the code of the versions do
not include any non-deterministic programming constructs. And 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 the replica determinism in the voting operations
performed by the RCMBs, provided that the correspondence 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 Section 4 we shall 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 correspondence requirement for the status replicas is that the status information of
all non-faulty replicas must for each segment 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 same assumptions which
were indicated above for the versions, 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 achieve-
ment of interactive consistency. The way in which we have defined the error con-
tainment 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

---

2 In control operations, values obtained from replicated sensors are also taken into account by the versions,
however voting is performed on those values before being used, during what is called an input cc-vector in
NVP.
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 mes-
sage broadcasted 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 seman-
tics of the RCMBs. Design diversity gives variations in execution times among ver-
sions and the failure semantics of the RCMBs may cause omissions, i.e., individual
cvectors 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. 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 which are
pending is aborted.

The ordered list of cc-vector messages received between the start and the stop mes-
sages 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 achiev-
ing 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.

![Figure 5: The state machine which enforces replica determinism for voting and status evaluation](image-url)
4 Consistent reintegration of RCMBs

The second aspect of the consistent management of the redundancy in our architecture that we are going to analyze is the reintegration of RCMBs affected by transient faults. As indicated in Section 1, this reintegration has to be performed as a basic means for preventing a quick and unnecessary attrition of the redundancy in our system.

The main requirement of this reintegration is that it has to lead the affected RCMB to a state which is consistent with all non-faulty RCMBs. There are also other properties exhibited by the reintegration mechanism. The complete set of properties is formalized as the following list:

- **Termination**: Any RCMB affected by a transient fault is eventually either reintegrated or disconnected from the network.
- **Consistency**: Any reintegrated RCMB reaches the same state of all other non-faulty RCMBs which includes the account of its own transient fault.
- **Non-triviality**: Any reintegrated RCMB reaches a state which is the same as the state all non-faulty RCMBs would reach at that moment in the absence of any fault or any invocation of the reintegration procedure, excepting the accounting of the RCMB fault in the status held by all non-faulty RCMBs.
- **Coherency**: Any RCMB affected by a transient fault and which is considered as permanently faulty by the status held by all non-faulty RCMBs is not reintegrated but disconnected from the network.
- **Fault Tolerance**: The RCMBs which were non-faulty when the RCMB requiring reintegration was affected by the transient fault are able to play their role in the reintegration as long as at least one of them does not fail during the reintegration process.

When an RCMB suffers a fault, the internal duplicated and compared circuitry detects the corresponding error. Then, the RCMB is preventively disconnected from the network in order to substantiate its crash failure semantics. At this moment the reintegration process starts. The RCMB’s duplicated processor is interrupted and it begins to execute a special procedure, called **Last Opportunity Procedure (LOP)**. If a new interrupt occurs before the RCMB is completely reintegrated the RCMB is shutdown.

Figure 6 presents the execution of the LOP in the faulty RCMB. The LOP starts in state 2 by locally determining whether the fault makes it impossible to reintegrate the RCMB (e.g. because the fault is permanent). This is done by checking some positions of the RCMB’s memory which are critical for recovering the circuit.

While the faulty RCMB is disconnected from the network, it may miss messages which have been sent through the bus. Therefore, even if the LOP decides that the RCMB is recoverable, the RCMB may have lost synchronization with the other RCMBs. Our approach to reintegration is that the affected RCMB asks the other RCMBs for the necessary information to become synchronized again. Instead of performing a complex protocol to determine which messages have been missed by the requesting RCMB, non-faulty RCMBs take the simpler approach of broadcasting to all requesters —there may
be several in a round— the complete status and the results of the last voting, which all together constitute what we call the **reintegration information**.

Thus, after the initial check, the LOP goes to state 3 in order to enable the transceiver that connects it with the network, broadcasts a **reintegration request** message, starts TIMEOUT2 and waits for the reception of the reintegration information. If this timeout expires before the RCMB has received the reintegration information, the LOP performs a shutdown of the RCMB (going to state 5).

However, the simple reception of the **reintegration information** is not enough to achieve complete reintegration with all non-faulty RCMBs. Additionally, when this **reintegration information** is received, it is necessary to ensure that the requesting RCMB has not missed other messages which are also necessary for the next voting and status evaluation. These messages include the reintegration requests transmitted by other RCMBs, since these request messages are used for updating the error counters of the status. Given that these requests may be issued at any time during the operation of the system, when a new status is evaluated after a stop message, all requests received between this stop message and the previous one are taken into account.

Therefore, non-faulty RCMBs can ensure that the requesting RCMB has not missed any single message for the next status if they do not send the last reintegration information immediately. Instead, they wait to calculate and send the reintegration information until after the next stop message. As long as the requesting RCMB starts collecting all messages when it receives the first stop message after sending its request (see state 4), this simple mechanism ensures a consistent reintegration.

The remaining problem is that, immediately after enabling its connection to the
network to transmit the reintegration request, the requesting RCMB does not know in which phase of the message exchange the system is. In particular if it receives a candidate to stop message, it cannot know whether this candidate is actually the stop message (i.e. the first candidate) or one of the candidates which were sent later because the RCMBs transmitting them were not capable to timely abort their transmissions. As several RCMBs may have requested reintegration in different moments, a requesting RCMB must know if it has actually received the stop message before using a reintegration information, which may have been broadcasted because of some other RCMB’s request. In order to solve this kind of scenarios, non-faulty RCMBs include in the reintegration information a unique identification of the stop message in this round. If the requesting RCMB in state 4 receives a reintegration information that indicates that the stop message is the same as the first candidate to stop message it received, the RCMB will use the reintegration information and move to state 1. In contrast, if the candidate is different from the stop message, the RCMB will continue in state 4 waiting for the next stop message.

Another potential scenario that is properly solved by our scheme occurs when an RCMB which has requested reintegration receives first a candidate to stop message and then another candidate, but the second one corresponds to the next segment. Given that no reintegration information was received between them, the RCMB understands that the first candidate was not the stop message for that segment, that there were no other RCMBs requesting reintegration before the stop message of that segment, and that the second candidate must be the stop message for the next segment. Therefore, the RCMB rejects all messages stored since it received the first candidate, starts storing all messages received after the second candidate, and gets ready to receive the reintegration information (see state 4).

It is important to note that receiving the reintegration information does not guarantee that the RCMB is going to be eventually reintegrated to normal operation. The reintegration request of the RCMB that has received the reintegration information has been used to increase the error counter for the corresponding RCMB, which may cause the RCMB to be declared as permanently faulty. If the reintegration information received indicates that the requesting RCMB is in a permanent failure, the RCMB will perform a shutdown going to state 5, thereby achieving the Coherency property indicated at the beginning of this section.

A final remark on the role of non-faulty RCMBs in the reintegration of faulty RCMBs is that our architecture guarantees that all non-faulty RCMBs have consistent values of the consensus cc-vector and status. Therefore all RCMBs are equally suitable for sending the reintegration information. In fact, our proposal is that all of them actually send this information. In the event of a failure of one of them, the information transmitted by any of the other is enough to successfully finish the operation. This behaviour corresponds to the Fault Tolerance property of the reintegration procedure.

We have used the modeling and verification tool UPPAAL [31] to verify the properties of the reintegration procedure. UPPAAL provides means for modeling in a timed-graph based notation, which essentially is a communicating state machine type notation extended with real-valued clocks. Properties are specified in a simple modal logic, and verified using the model-checking algorithm of UPPAAL. In our work with UPPAAL some of the reintegration properties listed at the beginning of this section (e.g. Co-
herency) have not been explicitly verified, since they appear as self-evident by analyzing the structure of our procedure. In contrast, the level of abstraction of our model has permitted the verification of other, less evident, properties such as Termination. A detailed description of the verification of the reintegration procedure will be provided in a technical report.

5 Related work

In this section we identify some similarities and differences of our system with other architectures. We shall focus on Delta-4 [24] and GUARDS [32] since all the concepts and techniques related to consistent management of redundancy were already mature at the time these architectures were developed. However, 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.

5.1 Similarities and differences in the replica determinism enforcement

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 fundamental 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 semantics of the nodes. Therefore, GUARDS has to use a high-level protocol (with high overhead) to execute the consistent communications (i.e. interactive consistency). 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.

5.2 Similarities and differences in the reintegration of faulty nodes

To prevent unnecessary attrition of redundancy, GUARDS proposes a specific mechanism for the reintegration of faulty computation nodes (called channels in GUARDS). Like in our approach, GUARDS’ faulty channels are also given the possibility of recovering in a first attempt by simply overwriting their erroneous variables with the values obtained after the voting, which is essentially the same as NVP proposes for the versions after voting on the cc-vectors. Another similarity is that when more drastic reintegration operations need to be performed in a node, the status is explicitly copied from the correct nodes, in analogy with NVP’s recovery points and with our mechanism for RCMB reintegration.

There are however also significant differences between the way we perform node reintegration in our system and the way GUARDS does it. In our approach the status information necessary for reintegration of a faulty RCMB is consistently maintained by all RCMBs, thanks to the fact that all the required data are consistently received by all RCMBs and that these circuits exhibit crash failure semantics. None of these features are present in GUARDS. Instead the status information may be perceived differently by different nodes. This makes it necessary to consolidate the status information periodically through an interactive consistency protocol, which represents a significant overhead.

Also, the reintegration of faulty RCMBs in our architecture is as simple as copying the value of the consensus cc-vector and of the status from any correct RCMB. In GUARDS this part of the reintegration is called state restoration (SR) and is a complex procedure in which all non-faulty nodes participate. This SR includes an error detection mechanism for the reintegration process but it does not guarantee the final success of this process.

As a last remark it can be said that our strategy (restricting the failure semantics of the nodes and using low-level communication services with consistency properties) was initially proposed with a view to simplify the enforcement of replica determinism, but in the end it turned out to also be fundamental for simplifying the reintegration of faulty RCMBs.

6 Discussion

After describing our architecture, some relevant issues arise which are discussed in this section.
6.1 Reduced number of nodes

Besides reducing the communication and computation overhead, our approach of providing consistent communication services at the lowest level of the system offers other advantages. One of these is a reduction in the number of required nodes.

In general, the number of required nodes for a communication protocol to provide a specific service varies depending on the failure semantics of the nodes [17]. To reach agreement under byzantine (i.e. unrestricted) failure assumptions for the nodes it is necessary to have \( n \geq 3t + 1 \) nodes (where \( n \) is the number of nodes and \( t \) is the maximum number of faulty nodes permitted), whereas for simple majority voting, once all values have been consistently exchanged, \( n \geq 2t + 1 \) nodes are sufficient. Since NVP is based on majority voting, \( n = 2t + 1 \) should be sufficient, each node executing a different version (therefore \( N = n \)). But, under byzantine failure assumptions this bound is no longer valid. However, by designing the RCMBs to present a restricted failure semantics, which at the same time restricts the failure semantics of the PCs, and by guaranteeing consistent communications, we ensure that \( n = N = 2t + 1 \) nodes is enough.

This reduction of nodes significantly reduces the cost of the system not only at the hardware level, but above all, at the software level, since it reduces the number of diverse software versions to be developed. Furthermore, reducing the number of nodes increases the dependability of the system, particularly in terms of the reliability. It is clear that if we reduce the number of nodes required to tolerate the failure of some of them, the complexity of the system decreases and the probability of a correct operation increases.

6.2 Restriction of communication failure

Another advantage of our approach is that it makes it possible to avoid some potential amplification of communication failures [21]. To see why, consider the alternative of using a high-level implemented communication service, e.g. an Atomic Broadcast protocol, which is designed to be used on top of a non-reliable communication channel linking nodes which may exhibit arbitrary failures. This service requires several rounds of message transmissions, e.g. the broadcast of a message will require several transmissions and execution of several operations in all nodes, such as the transmission of ACK messages. In this kind of complex communication schemes, a failure at the low level (e.g. an omission to send a message) does not necessarily manifest at the high level as the same type of failure (e.g. an omission to broadcast a message to all receivers). In fact, this kind of broadcast algorithms are likely to amplify the importance of failures which occur at the low level [21] (e.g. messages delivered to different receivers in a non consistent order due to an omission to send a message).

6.3 Simple network topology

One of the key techniques we use to reduce the complexity of the communication protocols is to restrict the failure semantics of the nodes. However this restriction
per se provides additional advantages to our architecture. Among them, is the simple topology of the broadcast network [24].

Without restricted failure semantics for the nodes the topology of the network must be carefully chosen. To prevent a faulty node from monopolizing the network it is necessary to use network topologies in which not all nodes share the same transmission channel [24]. Furthermore, to prevent a faulty node from impersonating a non-fault node it is necessary that the interconnection topology provides the identity of the source of a message. This can only be achieved with topologies having a large number of redundant communication links, which leads to high cost and also lack of extensibility, since we need to add a complete set of new links for each new node.

In contrast, when the failure semantics of the nodes is restricted enough so as to prevent both babbling idiots and impersonations (e.g. when nodes exhibit crash failure semantics), it becomes possible to deal with faults in the communication network independently from faults of the nodes. As a consequence, simpler network topologies based on shared multipoint transmission channels can be used. In fact, a single physical channel (communication medium) is enough to provide the necessary communication services. Nevertheless, we propose the use of replicated media following the approach of [25] in order to tolerate the permanent failure of one of the media.

6.4 Advantages of the mechanism for RCMB reintegration

Our particular approach to RCMB reintegration presents some advantages that are worth mentioning. First, it allows the reintegration of RCMBs affected by transient faults independently of the seriousness of their errors (i.e. the importance of their loss of consistency). And second, the reintegration is achieved with a low computation and communication overhead. Since we do not use an agreement protocol for reintegration, the amount of messages which have to be exchanged is substantially reduced. Furthermore, the transmission of the complete consensus cc-vector plus the status information requires a smaller communication overhead, compared to sending all the messages that a faulty RCMB could have missed in the worst case. Finally, with our approach the RCMB receives the information already processed and does not need to perform a voting or any other calculations on it.

6.5 Version synchronization

Voting on cc-vectors after executing each segment is the basic mechanism for version synchronization in NVP, since all versions wait for the consensus cc-vector before starting the next segment. In our architecture, the fact that all RCMBs receive the stop message simultaneously makes all of them to start voting at the same time and, therefore, to obtain the consensus cc-vector almost simultaneously. As a consequence, synchronization at the beginning of each segment is even tighter.

Also in our architecture, as the exchange of cc-vectors and the subsequent voting are performed by the RCMBs, an RCMB may be voting on a set of cc-vectors while its local version has not yet finished the current segment. Although not described in this paper due to space limitations, our architecture provides means for helping late versions to resynchronize with the rest. In particular each RCMB can send the calculated
consensus cc-vector to its late local version as long as the version finishes the current segment before RCMBs start voting again after the stop message of the next segment. If the late version has not finished by that time, it is simply considered as a permanently faulty version and the RCMB continues its operation with the cc-vectors provided by all the operating versions.

6.6 Limited bandwidth of CAN

Even though the consistency properties of CAN allowed the reduction of the communication overhead, the bitrate of CAN is limited to 1Mbit/s [15] in its typical implementation. This has to be taken into account when designing the NVP application, since it has an important effect on the time that versions have to wait until receiving the consensus cc-vector. Both the frequency of the votings and the size of the cc-vectors have to be properly chosen in order to fulfil the timing requirements of the application, while at the same time, achieving the required level of reliability. In a previous analysis [9] on the performance of NVP applications, it is estimated that the overhead introduced by the need of periodic votings is between 25 percent and 90 percent. The 90 percent overhead corresponds to very short segments (4ms. average segment execution times) issuing very large cc-vectors (200 floating-point numbers per cc-vector on average) which is not typical in most applications. The problem of the overhead introduced by the periodic voting is intrinsic to NVP and even to other software replication techniques, such as active replication [17]. In any case the actual suitability of NVP and of this architecture depends on the specific application and on its specific timing requirements. An analysis on the suitability of active replication on CAN for automotive applications can be found in [17].

6.7 Generality of our architecture

Although initially developed for NVP, our redundancy management circuitry 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.

6.8 Instantiation of our architecture

As pointed out in several parts of this article, there is a number of parameters which have to be fixed for instantiating our architecture in order to build a specific system. First of all, it is necessary to decide the value of $N$, since it determines the actual fault tolerance provided by the system. Given that the general cost of the system critically depends on $N$, $N = 3$ is the most likely choice since it provides tolerance to the failure of one node-version at the lowest cost. Other parameters that have to be established are
the thresholds of the counters of the RCMB and version errors. These two parameters are not so critical since they only determine how many opportunities we give to a component which is generating errors before declaring it as permanently faulty.

It is also necessary to choose values for the permitted deviation for floating point outputs. Likewise the permitted deviation in the issue time of all outputs has to be specified to permit both detecting faulty versions and to satisfy the specific timing requirements of the application. Also related to the timing requirements is the choice of the frequency of the votings and the size of the cc-vectors. The higher the frequency and the size are, the more efficient the error detection will be. In contrast, the high number of votings and the time necessary to process those large cc-vectors will negatively affect the chances of the system to meet its deadlines. This tradeoff is a typical problem both in NVP and in active replication and has been studied by other authors in previous work as indicated above.

6.9 System start-up

Another issue which is relevant in the practical use of this architecture is the start-up of the system. A simple strategy for this start-up is the following. After connecting all components to the power supply, both PCs and RCMBs perform their own self-test procedures independently. When these procedures are satisfactorily completed, each PC receives the notification corresponding to its local RCMB. Each PC shows the notification of the success of its own test and of the test of the local RCMB to a human operator. Then the operator launches the NVP application from any of the PCs. This PC, instead of starting the execution of the first segment of the application, broadcasts a message to all PCs through its local RCMB. Thanks to the properties of the RCMBs and of the CAN protocol the message will be consistently received by all RCMBs at almost the same instant. Then it will be sent to each of the PCs (including the sender) and they will start executing the first segment of the application almost simultaneously.

7 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.

References


