Department of Mathematical Sciences and Informatics. UIB.
Technical report A-02-2007

On the Management of Media Replication in ReCANcentrate

Manuel Barranco, Julián Proenza
Dpt. Matemàtiques i Informàtica
Universitat de les Illes Balears, Spain

Luís Almeida
DET/IEETA
Universidade de Aveiro, Portugal

September 13, 2007

Abstract

Distributed embedded control systems for safety-critical applications require a high level of dependability. Despite the existence of communication protocols such as TTP or FlexRay specifically developed to provide that level of dependability, there has also been an increasing interest in CAN, given its low-cost, electrical robustness, good real-time properties and widespread use. However, the use of CAN in these applications has been controversial due to dependability limitations. To overcome some of those limitations, namely those arising from its non-redundant bus topology, we have proposed a replicated star topology, ReCANcentrate, which is transparent for any CAN-based application and protocol, and whose hubs incorporate the necessary fault-treatment and fault tolerance mechanisms. In this document we focus on how each node of ReCANcentrate manages the transmissions and the receptions on the replicated star, as well as how it tolerates faults.

1 Introduction

Distributed embedded control systems for safety-critical applications, e.g., X-by-Wire systems, are widespread in several domains, such as avionics and the automotive industry. One of the most important requirements of these distributed systems is to rely on a high-dependable communication infrastructure. In this sense, a big effort is being made in developing high-reliable communication infrastructures, such as TTP [1] and FlexRay [2]. These infrastructures fulfill high reliability, in part, by means of replicated communication media architectures, which provide the necessary fault tolerance.

However, there has also been a growing interest in using CAN [3], or CAN-based protocols, e.g. FlexCAN [4], given its high electrical robustness, low price, and bounded access delay. In addition, CAN has been extensively used in practice for over 13 years with low failure rates [5]. Nevertheless, the use of CAN in critical applications has

been controversial due to dependability limitations. Some of these limitations arise from its non-redundant bus topology, which lacks the necessary error-containment and fault tolerance mechanisms. In order to overcome these limitations, we have developed a new replicated star topology, called ReCANcentrate that includes two hubs [6] (see Figure 1). In ReCANcentrate each node is connected to each hub by a dedicated link that contains an uplink and a downlink. Additionally, both hubs are interconnected by means of at least two *interlinks* each of which contains two independent sublinks, one for each direction. Each hub includes fault-treatment capabilities to contain errors originated at nodes [7], and to provide tolerance to hub and link faults. ReCANcentrate is fully compatible with CAN and commercial off-the-shelf (COTS) CAN components, being transparent for any CAN-based application. In this way, ReCANcentrate can make CAN appropriate for the most demanding safety-critical systems, providing for CAN many of the features concerning fault tolerance that are typical of protocols such as TTP and FlexRay.

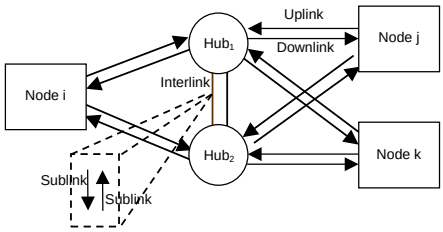

Figure 1: Architecture of ReCANcentrate

This document focuses on the management of the replicated star performed in each node. Notice that ReCANcentrate uses active replicated media in order to provide fault tolerance. For this purpose, the same data is transmitted in parallel throughout each of the media replicas; so that, in principle, each communication medium, i.e. each star, can be considered as a channel that conveys a replica of the same data.

One of the major problems when managing active replicated channels in parallel is that each node must be able to deal with redundant frames. Notice that to transmit in parallel does not guarantee the traffic to be equal in all channels. Therefore, each node must determine whether or not two frames received at different instants of time, each one through a different channel, are in fact copies of the same frame (duplicates). Moreover, the node must also be able to diagnose when a frame received from one channel is omitted from the others (omissions).

Synchronizing frame transmissions and receptions across the network is a possible solution. This synchronization is easily achieved in time-triggered protocols, such as TTP and FlexRay, since they rely on a TDMA transmission schema. With TDMA each frame is expected to be transmitted quasi-simultaneously in all channels at predefined time slots. Hence, to remove duplicates and to detect omissions is straightforward.

Unfortunately, CAN provides no means for synchronizing frames in different replicas. Therefore, due to the error-signaling and arbitration mechanisms of CAN, a single bit error in one channel is enough to lead its traffic to evolve in a different way than in

the other replicas. Thus, additional mechanisms have been proposed in the literature to cope with this problem. Some solutions, such as FlexCAN, are intended to provide some sort of synchronization, coordinating the transmissions and receptions on the channels by means of timers. Another interesting solution, proposed in [8], avoids the necessity of dealing with duplicates and simplifies the detection of omissions, by coupling the streams received from all replicas, at the bit level, in each node.

It would be possible to adopt any of these existing solutions for dealing with redundant frames in ReCANcentrate. However, either they are complex and expensive in terms of hardware and software, or they limit the accuracy of the fault diagnosis performed by each hub [6]. Fortunately, ReCANcentrate allows each node to remove duplicates and to detect omissions in a very simple way that does not present these disadvantages. This is the first topic covered in this document.

The other main problem that must be solved by each node when managing replicated media is to detect when a fault in the media prevents it from communicating through a given medium; so that the node can continue communicating using only non-faulty medium replicas. The complexity of this problem depends on the architecture of the communication subsystem. The second contribution of ReCANcentrate presented here is that it allows each node to easily perform such required fault diagnosis and passivation.

In this document we firstly address the basic characteristics of CAN and ReCANcentrate. We then focus on how ReCANcentrate allows nodes to easily manage the replicated media and explain the proposed management itself. Afterwards, we describe the basics of a possible software implementation of such management using hardware COTS components, and, finally, we conclude the document.

2 CAN and ReCANcentrate basics

The Controller Area Network (CAN) protocol is a field bus which fulfills the communication requirements of many distributed embedded systems. Probably the most important characteristic of CAN is that its physical layer implements a wired-AND function of every node contribution, thereby providing a dominant/recessive transmission. This property guarantees that whenever one of the nodes transmits a dominant bit value, i.e. a logical '0', this value is received by all the nodes in the network. In contrast, a recessive bit value, i.e. a logical '1', is only received as long as every node issues a recessive.

Moreover, CAN communication relies on a complex bit synchronization mechanism which guarantees that nodes have a quasi-simultaneous view of every single bit on the medium. This bit synchronization allows the definition of a number of additional mechanisms [3], which significantly improve the dependability properties and real-time response of CAN [9]. One of the properties that CAN is commonly assumed to have is the referred to as *atomic broadcast*, which guarantees that a frame is either simultaneously accepted by all nodes or by none. This property is of capital importance in fault tolerance and real-time distributed systems [10].

However, one of the main impediments for using CAN in safety-critical control systems is that it relies on a non-redundant bus topology that provides scarce error-

containment and fault tolerance mechanisms. For this reason, a CAN bus includes multiple single points of failure, i.e. components whose failure cause the failure of the overall system [9]. Particularly, the faults that may cause a generalized failure in CAN are: stuck-at-dominant and stuck-at-recessive faults, which can occur within nodes or in the medium; medium partition faults, which occur whenever the network is physically broken into several subnetworks; bit-flipping faults, which occur whenever a network component, either node or medium, sends random erroneous bits with no restrictions in value or time domains; and babbling idiot faults that occur whenever a node sends incorrect frames that are erroneous in the time domain, causing undesired interference [11].

In order to eliminate all single points of failure from a CAN network we have developed a new replicated star topology called ReCANcentrate [6] (see Figure 1). Each hub receives each node contribution through the corresponding uplink, couples all the non-faulty contributions with a logical AND function, and broadcasts the resultant coupled signal through the downlinks. The use of an uplink and a downlink allows each hub to monitor each node contribution separately and detect faulty transmissions. Permanently stuck-at or bit-flipping contributions are disabled, and so not propagated to the coupled signal, thus being confined to the port of origin. A further improvement of ReCANcentrate concerns the detection and passivation of babbling-idiot faults, which could be achieved in a relatively simple way [7].

The replication strategy is such that nodes transmit the same data through both stars in parallel. However, an error in one star could cause inconsistencies in the traffic of the stars, making data replication more complex to manage. Hence, both hubs exchange their traffic through the interlinks and perform a special AND coupling [6] to create a single logical broadcast domain. Thereby the same value is transmitted bit by bit through their downlinks, so that the quasi-simultaneous view of each bit is enforced in the whole replicated domain.

In what concerns a hub fault, notice that it can only manifest as the transmission of stuck-at or bit-flipping bits through one or more hub ports. This is because a hub has not the capacity of building CAN frames [9]. A hub fault is confined at two different levels. Firstly, each hub is able to detect errors in any sublink coming from the other hub and to isolate it when faulty. Secondly, as will be explained later, each node confines the fault mainly using the CAN standard fault diagnosis mechanisms, and continues communicating through the non-faulty hub. This second level of fault confinement also applies to link faults, so that each node can tolerate the failure of one of its links.

3 Replicated media management in ReCANcentrate

3.1 Simplification of the media management

As already explained, since hubs are coupled, each node receives each bit from all hubs quasi-simultaneously . In such a way, the set of hubs of ReCANcentrate can be seen as a single hub that provides a single communication domain. This feature allows to build a simple and reliable solution for replicated media management.

To better understand this, we can make an analogy between ReCANcentrate and a

CAN bus in which each node includes two controllers to access the bus. As depicted in Figure 2, the two coupled hubs are logically equivalent to a unique CAN bus, and each link corresponds to a given stub. Thereby, each node does not have to deal with a set of replicated channels, but with different views of the same channel, which is easier.

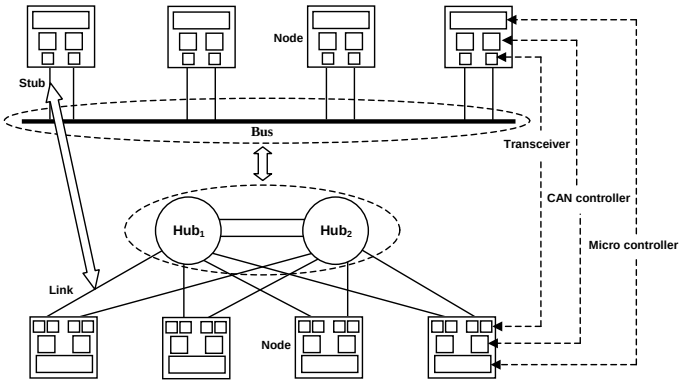

Figure 2: Analogy between a bus and the coupled hubs.

In particular, the management of the replicated media can be reduced to basically trigger each transmission towards one of the hubs only, while receiving from both hubs at the same time. We proposed a sketch of this idea and the node hardware architecture needed to support it in [6]. The node is constituted by COTS components only: two CAN controllers and a micro-controller, as depicted in Figure 2. A given CAN controller is connected only to one hub by means of a dedicated uplink and downlink, using for this purpose two COTS transceivers [9].

One of the controllers acts as the *transmission controller*, so that it is used to both transmit the frames of its node and receive frames sent by other nodes. Note that the *transmission controller* does not receive its own frames. The other controller is used as the *reception controller*. It receives frames transmitted by its own node, as well as by other nodes. If one controller fails, the non-faulty one is used as *transmission controller*.

When a frame is successfully exchanged through the network, i.e. when a *delivery event* occurs, each node expects that its two controllers quasi-simultaneously notify of that event. This quasi-simultaneous notification can occur in two different manners. On the one hand, if the node successfully transmits a frame, the *transmission controller* and the *reception controller* notify of the transmission and reception of this frame respectively. On the other hand, if the node receives a frame sent from another node, it expects to be simultaneously notified of its reception by its two CAN controllers.

Notice that the node must be fast enough to handle the pair of notifications corresponding to a given *delivery event* before a new *delivery event* occurs. As will be explained, the fulfilment of this requirement is necessary to correctly associate each controller notification with its corresponding *delivery event*, and further enhances the capabilities of detecting controller faults.

It is worth noting that all these simplifications are possible under the hypothesis that there is a single communication domain. Nevertheless, there is a situation in which such hypothesis does not hold: when each hub continues coupling the contributions of its own nodes and, meanwhile, both hubs are not coupled with each other. This can only happen if all interlinks are faulty and, thus, isolated at their corresponding hub ports. Since ReCANcentrate uses several interlinks to tolerate permanent interlink failures, the probability of such a situation should be very low. How nodes manage two independent stars when hubs become decoupled is beyond the scope of this document.

Leaving out the scenario in which all interlinks are faulty, the fact that both stars form a unique communication channel is valid even in presence of faults. This eliminates the necessity for each node to deal with discrepancies between channels, which is difficult and typical in other replicated media architectures. In contrast, a fault can only lead a node to observe that its two controllers differ in the vision of the same channel. As will be explained, to manage such local discrepancies between controllers is simple.

Next, we analyze the faults that can occur in the communication subsystem, all discrepancies they can provoke, and describe our replicated media management.

3.2 Discrepancies between the two visions of the single communication domain

We differentiate between faults occurring at the media: a hub, transceivers, connectors, cables, etc., and at controllers. In what concerns media faults, recall that a hub has not capacity of building CAN frames [9]. Thus, a hub fault can only manifest itself as the transmission, through any of its ports, of syntactically incorrect bits. Because of the same reason, faults at other parts of the media can also only manifest as the generation of syntactically incorrect bit values.

Once a fault in the media is confined at the corresponding hub port, the controller attached to that port will not notify its corresponding micro-controller of any further transmission or reception. Thus, thereafter its node will constantly detect what we call an *omission discrepancy*, which occurs when the node observes that only one of its CAN controllers informs about the occurrence of a *delivery event*. In contrast, since in ReCANcentrate there is a single communication domain, a non-confined media fault will be signaled by all controllers by means of CAN *error-flags* [3]. This implies that, in principle, it is impossible that any controller notifies about a transmission or a reception until the fault is confined and, hence, no discrepancy can take place meanwhile. Nevertheless, there is an exception to this statement: the occurrence of any of the *inconsistency scenarios* that have been identified for standard CAN, which may occur in the presence of errors in the last-but-one bit of a frame [10]. In these scenarios the atomic broadcast is violated, even when there is a single communication domain. From the point of view of a node of ReCANcentrate an *inconsistency scenario* may manifest as an *omission discrepancy*.

Regarding faults happening at controllers, we analyze their effects following the well-known categorization of failures proposed in [12]. We distinguish between *crash* and *byzantine* controller failures. When a controller exhibits a *crash failure*, it stops performing any action, so that the node will observe an *omission discrepancy* thereafter.

In the case of presenting a *byzantine failure*, the controller fails arbitrarily with no restrictions neither in the value domain nor in the time domain. Thus, a *byzantine failure* in a controller can provoke not only an *omission discrepancy*, but also what we call a *non-omission discrepancy*. A *non-omission discrepancy* occurs whenever a node observes that its two controllers notify of a *delivery event*, but they do not coincide in the frame the event is related to. For example, a controller that exhibits a byzantine failure in the time domain may notify with a very big delay the occurrence of a *delivery event*. If a delayed notification coincides in time with a notification related to a later *delivery event*, the frames related to those events might not coincide provoking a *non-omission discrepancy*.

3.3 Media management functionality of a node

In absence of faults, the management strategy to handle transmissions and receptions in ReCANcentrate is simple. When a successful transmission and reception are respectively notified from the *transmission* and the *reception* controllers, the node assumes the transmission as correct and releases the reception buffer of the *reception controller*. Otherwise, when a successful reception is notified from both controllers, the node reads the frame from one of them (no matter which), an releases their reception buffers.

Only minor changes in such strategy are required in order to deal with faults. Notice that it is not mandatory that a node handles all possible faults, which have been identified in Section 3.2. On the one hand, it is not compulsory that a node detects any of the *inconsistent scenarios* of CAN. First, because the probability of occurrence of these scenarios has been controversial [5]. Second, because they are not a new problem introduced by the use of media replication, but an old problem of CAN, which can be avoided using any of the modifications or additions to CAN that have been already proposed [10]. Furthermore, since ReCANcentrate is transparent for any CAN-based protocol, it can be used as their communication infrastructure anyway. Therefore, we exclude the treatment of *inconsistent scenarios* from our replicated media management.

Likewise, controller faults are also an old problem of communication subsystems, e.g. in a typical non-redundant CAN bus, the controller of a node may forge notifications of transmissions and of receptions. Thus, to treat controller faults is not mandatory for a proper management of the replicated media. Nevertheless, since our node hardware architecture includes two controllers, we propose that each node takes advantage of the discrepancies between them to detect controller faults to some extent.

Taking into account all these considerations, each node treats faults as follows. First, a fault in the media can only provoke *omission discrepancies* in which the controller that has problems for communicating is the one that omits the notifications (*inconsistency scenarios* are excluded). However, as explained in Section 3.2, an *omission discrepancy* can also be provoked by a *byzantine* controller fault, so that the controller that omits a notification may be the non-faulty controller. Thus, since an *omission discrepancy* does not indicate which controller has problems for communicating, it cannot be used to diagnose media faults. Fortunately, a typical CAN controller includes some useful fault diagnosis mechanisms: a *Transmission Error Counter* (TEC), a *Reception Error Counter* (REC) [3], and a programmable threshold for them, called *error warning limit*. We propose to use these mechanisms to treat media faults. Whenever any of

the error counters of a CAN controller reaches the referred limit, the node stops using that controller for communicating, thereby isolating the faulty media.

Although an *omission discrepancy* cannot be used to diagnose a media fault, it is still necessary to decide whether or not the notification of a *delivery event* is valid. We propose to use a best-effort strategy that consists in assuming the notified event and its corresponding controller as correct, but without diagnosing the controller that omits it as faulty. If the notification was actually incorrect, to accept it is wrong, but this situation can exclusively be provoked by a controller fault and we are not obliged to deal with it. Moreover, at least the non-faulty controller is not penalized. If the notification was correct, then the controller that omitted it is faulty or was isolated due to a media fault. In both cases the decision is correct because it allows to tolerate the fault. It is worth noting that in other replicated media architectures the decision of what to do when observing omissions is not so simple. Since an *omission discrepancy* does not happen between controllers but between channels, nodes must diagnose which fault provoked it; otherwise they would not treat an important number of media faults.

In what concerns controller faults, it is possible to detect a *byzantine* controller fault when the notification from a faulty controller coincides in time with a notification of the other controller, and both notifications refer to a different frame, i.e. when a *non-omission discrepancy* occurs. When this happens the node cannot know a priori which controller is actually faulty. Hence, it has to stop communicating and run an internal test in order to take a decision. This simple fault diagnosis feature is an advantage of our approach compared with other solutions. Specially with respect to those that use only one CAN controller [8], since they cannot detect controller faults by means of a simple comparison.

4 Replicated media management routines

Next we propose a possible implementation of the presented replicated media management. It consists in building a library that includes all the functionality needed to abstract away the details of both, the node architecture and the management strategy. This library basically includes a reception and a transmission buffer, as well as a set of interrupt service routines to handle different communication events.

In order to correctly manage the role that each controller must perform during communication, the library marks each controller as being in one of the following states: *transmission controller*, *reception controller*, and additionally *non-active controller*, which means that the controller is not being used because it has been diagnosed as faulty, or because it has just been initialized.

Notice that to base the structure of the library on a set of interrupt service routines allows to reduce the overhead of the application which, otherwise, would need to periodically read the status registers of the CAN controller to check the state of the communications. In particular, the library is devised to use CAN controllers that at least include three interrupts: a *transmit interrupt*, which originates whenever a frame has been successfully transmitted; a *reception interrupt*, which triggers whenever a new frame has been received; and an *error interrupt*, which is launched when the *error warning limit* is reached, as well as when the controller changes from being involved

in communication activities to not being communicating, i.e. when it passes from the *active state* [3] to the *bus-off state*, and viceversa. Additionally, the library assumes that these interrupts have the same priority, so that they are served following a FIFO policy.

This section explains the basics of this library by briefly describing the general logic structure of the service routines that are triggered by the referred interrupts.

4.1 Transmission and reception routines

The *transmission routine* and the *reception routine* respectively handle the *transmit interrupt* and the *reception interrupt*. Then, when a *delivery event* occurs, it is expected that each controller of the node notifies of it by triggering one of these routines, which will be executed in the micro-controller of the node.

It is worth noting that a node does not observe each bit in both stars exactly at the same instant of time, and that its controllers (and transceivers) have different internal delays. Thus, when a *delivery event* occurs, one controller will be the first to interrupt the micro-controller and will be served. Meanwhile the execution of the interrupt triggered by the other controller will be pending. However, as explained next, these two routines cooperatively handle the event. For the sake of simplicity, the routine executed first will be referred to as *routine A*, whereas the other as *routine B*.

Besides performing the operations needed to handle the *delivery event*, routine A has to check that the notifications performed by both controllers refer to the same frame. If the result of this checking is affirmative, the routine leaves an indication of the correspondence between both notifications to routine B, so that routine B does not have to check it again. If the notifications do not refer to the same frame, then a *non-omission discrepancy* occurs; routine A notifies of it to the application, and executes the *quarantine routine* for each controller to mark them as *non-active*, as will be explained in Section 4.2.

Notice that since routine A is the one that is executed first, it has to give enough time to allow the trigger of routine B. If when this time expires, routine B has not been launched yet, routine A assumes that an *omission discrepancy* occurred, and goes ahead to perform alone the actions needed to handle the *delivery event*.

In what concerns routine B, it must reset the indication (left by routine A) that informs about the correspondence between notifications. Otherwise, the execution of a routine corresponding to a future *delivery event* would accept an obsolete indication. Besides, routine B performs the actions needed to handle the *delivery event*, but without carrying out the operations already performed by routine A.

This cooperation between the two routines is only possible if the micro-controller executes them fast enough to prevent that a new *delivery event* occurs before they finish. Otherwise, a given routine could cooperate with a routine related to a later *delivery event*. The time used to execute them must not exceed the time required for transmitting the shorter CAN *remote frame* [3]. Particularly, in a CAN network operating at 1Mbit/s and with a utilization of 100%, this time is around 49us. Thus, the temporal requirements of routines that handle *delivery events* have to be taken into account when dimensioning the whole distributed system.

Figure 3 depicts the general logic structure of the *transmission routine*. First, the routine checks if a former reception routine (which could have been triggered first by the *reception controller*) left an indication that confirms that the frame transmitted has been already received. If the result of this checking is affirmative, the routine plays the role of routine B. It knows that the frame was not only successfully transmitted by the *transmission controller*, but also correctly received at the *reception controller*. Then, it only needs to reset the indication, to notify the application of the successful transmission, and to release the transmission buffer of the controller.

If the result of the checking is negative, the *transmission routine* acts as a routine A. It waits K units of time to give enough time to the *reception controller* to notify the reception of the transmitted frame. If the *reception controller* notifies the reception of a frame, and that frame coincides with the transmitted one, the *transmission routine* leaves an indication of this correspondence. Then, it notifies the application of the successful transmission, and releases the transmission buffer. Otherwise, if frames do not coincide, the routine notifies the application that a *non-omission discrepancy* occurred, and executes the *quarantine routine* for each controller. Finally, if the *reception controller* does not notify the expected reception, an *omission discrepancy* occurred. In this case, the routine does not indicate the correspondence, but goes ahead.

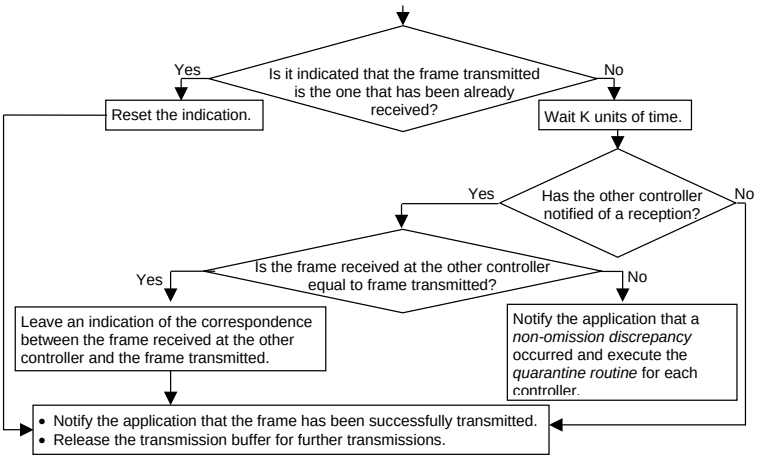

Figure 3: Transmission routine

Figure 4 depicts the general logic structure of the *reception routine*. It is analogous to the *transmission routine*. The main difference between them is that the *reception routine* must check whether the received frame is in fact a copy of the frame transmitted through the other controller, or in contrast, it is a frame transmitted by another node.

To know if the former possibility has occurred, the routine inspects if the controller whose notification is handling is marked as the *reception controller*. If the result of the inspection is affirmative, it knows that the other controller is the *transmission controller*, but it cannot still be sure of being handling the reception of a frame sent by that controller. So it further inspects whether or not a former *transmission routine* has

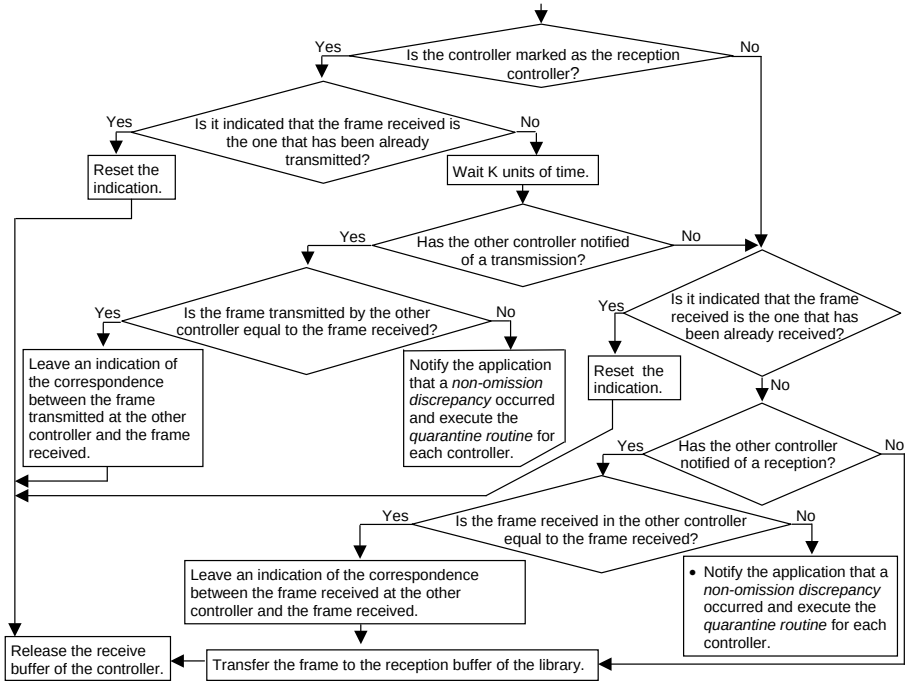

Figure 4: Reception routine

indicated such correspondence. Moreover, if this correspondence is also not indicated, the *reception routine* still does not discard being handling the reception of a transmission, since maybe the *transmission routine* is going to trigger later on. Hence, it waits a possible notification of transmission from the other controller. If it occurs, the routine is the responsible for testing the correspondence between the frame it handles and the frame transmitted. But if the notification of transmission does not happen, it definitively abandons the possibility of handling a reception of a frame of its own node. Nevertheless, it does not assume an *omission discrepancy*, but continues as explained next.

When the routine knows that the unique possibility left is to be handling the reception of a frame sent by another node, it must check if a former *reception routine* has indicated the correspondence between the two received frames. If this correspondence was indicated, the routine only needs to reset the indication and to release the reception buffer of its corresponding controller. Otherwise, it acts as the routine A and must check the correspondence with the frame that is expected to be received at the other controller. If this correspondence is successfully confirmed, the routine leaves an indication of it, transfers the received frame to the reception buffer of the library, and releases the reception buffer of its corresponding controller.

4.2 Error state routines

The *error state routines* handle two events: the diagnosis of a controller as being faulty, and the decision of reintegrating it. These two events are respectively handled by two routines: the *quarantine routine* and the *reintegration routine*.

The *quarantine routine* is executed when a controller triggers an *error interrupt* due to the reaching of its *error warning limit*. In addition, whenever the node observes a *non-omission discrepancy*, this routine is executed one time for each controller. Please, refer to Section 3.3 for an explanation of when a controller is diagnosed as faulty.

Firstly, this routine resets the corresponding controller. Notice that each hub performs a reintegration policy [9] that demands the controller to be inactive for a specific period of time. Once is reset, a CAN controller enters the *bus-off state*, in which it does not communicate. This opens the possibility of its reintegration at its hub port, if it was isolated by the hub as a consequence of a transient fault. Moreover, this reset can be the first action of a test performed by the node to determine which controller is faulty after detecting a *non-omission discrepancy*.

Additionally, this routine has to mark the corresponding controller as *non-active*, and to further perform three verifications. Firstly, it has to check if a transmission request was pending on the controller. If the result of the checking is affirmative, the routine must notify the application that such request was aborted. Secondly, if the controller was the *transmission controller*, the routine has to assign this role to the other controller. Finally, if the other controller is already marked as *non-active*, the routine must notify the application that it is not possible to communicate with the other nodes.

Regarding the *reintegration routine*, it performs the actions needed to use again a controller that was previously quarantined. On the one hand, it is executed when the controller triggers the *error interrupt* as a consequence of passing from the *bus-off state* to the *active state*. This happens when, after a reset, the controller monitors certain conditions [3] that lead it to consider that, maybe, it is able to communicate again. On the other hand, the *reintegration routine* is also executed when after a *non-omission discrepancy* the node runs a test that diagnoses said controller as non-faulty.

Basically, the routine has to mark the controller as the *reception* or the *transmission* controller depending on whether or not the other controller is marked as the *transmission controller*. In case that the other controller is marked as *non-active*, it also has to notify the application that it is possible to communicate again.

Finally, notice that the execution of a *reintegration routine* is not enough to ensure that the corresponding controller is not isolated at its hub port. However, to re-use a controller that actually cannot communicate with the other nodes will not cause any problem. The controller only will generate omissions, and will eventually reach the *error warning limit*, thereby being quarantined again.

5 Conclusions

Communication infrastructures for safety-critical distributed control systems, such as TTP and FlexRay, are evolving towards the use of replicated media architectures to

fulfill the required high reliability by means of fault tolerance. Besides, there is an increasing interest of using CAN in safety-critical applications, due to its robustness and widespread use. However, the non-redundant bus topology it relies on lacks the appropriate error-containment and fault tolerance mechanisms. To provide these mechanisms, we developed a replicated star topology called ReCANcentrate, which additionally yields interesting advantages compared with other CAN replicated media architectures [6].

In this document we have proposed the management of the replicated media performed in each node of the network. Such management takes advantage of the fact that the hubs of ReCANcentrate are coupled with each other, thereby forcing both stars to behave as a unique communication channel. Each node can easily remove duplicates and detect omissions, since its two controllers are attached to the same channel and, thus, it is expected that they quasi-simultaneously notify of each frame exchanged on the network. Furthermore, a node can easily diagnose and passivate a fault in one of the stars. A fault in the medium of a star prevents from communicating one of the controllers of one or more nodes. When this happens, an affected node observes that its impaired controller either omits notifications of transmissions and of receptions, or accumulates too many errors. In the first case, the fault is tolerated since the node continues communicating through the other controller. In the second case, the node uses the fault diagnosis mechanisms already provided by CAN, to diagnose that the impaired controller cannot communicate.

Moreover, controller faults can also be tolerated to some extent beyond the capabilities of CAN and other CAN redundant architectures. In particular, a node becomes aware of the failure of one of its controllers when it observes that its two controllers quasi-simultaneously notify of the exchange of different frames.

Finally, we have presented a possible software implementation of the proposed replicated media management, by briefly describing the general logic structure of its main functions.

FTUs: nodos fail-silent (como ttp) la estrella podra ser innecesaria en terminos estadisticos. Lo unico que gana: proximity, common-mode, etc.

Estos parametros son adems muy difciles de cuantificar.

En cambio, si comparamos implementacin nodos fail-silent en bus con implementacin sin nodes fail-silent en la estrella; aqu podemos ver ventaja de la estrella: puede que la fiabilidad sea mayor en la estrella (nodos complejos en el bus), ms barato

more fail-silent... no queda clara la aportacion estrella.

- FTUs complexity, only to bus (in duplicate nodes).
- FTUs more
- Numbers of

Teniendo nodos duplicados la semantica de averias de los nodos esta ms restringida, cosa que el hub no puede: mensajes semanticamente incorrectos.

References

[1] H. Kopetz and G. Grunsteidl, “TTP - A Protocol for Fault-Tolerant and Real-Time Systems,” *IEEE COMPUTER*, January 1994.

[2] FlexRay^{TM}, "FlexRay Communications System - Protocol Specification, Version 2.0," FlexRay^{TM}, 2003.

[3] ISO, "ISO11898. Road vehicles - Interchange of digital information - Controller Area Network (CAN) for high-speed communication," 1993.

[4] J. R. Pimentel and J. A. Fonseca, "FlexCAN: A Flexible Architecture for Highly Dependable Embedded Applications," *The 3rd International Workshop on Real-Time Networks, Catania, Italy*, July 2004.

[5] J. Ferreira, A. Oliveira, P. Fonseca, and J. Fonseca, "An experiment to Assess Bit Error Rate in CAN," *Proceedings of 3rd International Workshop on Real-Time Networks, Catania, Italy*, 2004.

[6] M. Barranco, L.Almeida, and J. Proenza, "ReCANcentrate: A replicated star topology for CAN networks," *ETFA 2005. 10^{th} IEEE International Conference on Emerging Technologies and Factory Automation, Catania, Italy*, 2005.

[7] M. Barranco, L. Almeida, and J. Proenza, "Experimental assessment of ReCANcentrate, a replicated star topology for CAN," in *Safety-Critical Automotive Systems*. Society of Automotive Engineers, USA, 2006.

[8] J. Rufino, P. Veríssimo, and G. Arroz, "A Columbus' Egg Idea for CAN Media Redundancy," *FTCS-29. The 29th International Symposium on Fault-Tolerant Computing, Winconsin, USA*, June 1999.

[9] M. Barranco, J. Proenza, G. Rodríguez-Navas, and L. Almeida, "An Active Star Topology for Improving Fault Confinement in CAN Networks," *IEEE Transactions on Industrial Informatics, vol. 2, issue 2*, pp. 78–85, May 2006.

[10] J. Proenza and J. Miro-Julia, "MajorCAN: A modification to the Controller Area Network to achieve Atomic Broadcast," *IEEE Int. Workshop on Group Communication and Computations, Taipei, Taiwan*, 2000.

[11] I. Broster and A. Burns, "An Analyzable Bus-Guardian for Event-Triggered Communication," in *Proceedings of the 24th Real-time Systems Symposium (RTSS)*. Cancun, Mexico: IEEE, Dec 2003, pp. 410–419.

[12] S. Poledna, "System model and terminology. In Fault-Tolerant Real-Time Systems: The Problem of Replica Determinism, Real-Time Systems, chapter 3," in *Engineering and Computer Science*. Kluwer Academic Publishers, Boston, Dordrecht, London, 1996, pp. 21–30.