Management of Media Replication in ReCANcentrate

Manuel Barranco[1], Julián Proenza[1], and Luis Almeida[2]

[1]DMI - Universitat de les Illes Baleares, Spain, [2]DET/IEETA – Universidade de Aveiro, Portugal

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.

Introduction

One of the most important requirements of distributed embedded control systems in safety-critical applications is a highly reliable communication infrastructure, such as TTP [1] and FlexRay [2]. This requirement can be achieved, in part, by means of replicated communication media schemes, which provide the necessary fault tolerance.

Although there has been a growing interest in using CAN [3] or CAN-based protocols in this context [4], one of the major limitations of CAN is that it relies on a non-redundant bus topology that 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 [5] (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 [5], 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 or protocol.

In ReCANcentrate the same data is transmitted in parallel throughout each of the media replicas in order to provide fault tolerance, i.e. each star can be considered as a channel that conveys a replica of the same data. However, two major problems arise when managing active replicated channels in parallel. First, to transmit in parallel does not guarantee the traffic to be equal in all channels. Thus, 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). Notice that due to the error-signaling and arbitration mechanisms of CAN, a single bit error in one channel is enough for its traffic to evolve in a different way than in the other replicas. The other main problem of using replicated channels is that each node must be able 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.

Mechanisms have been proposed to cope with these problems when using replicated

CAN channels [4] [6]. Although any of them could be adopted for dealing with replicated stars in ReCANcentrate, either they are complex and expensive in terms of hardware and software, or they would limit the accuracy of the fault diagnosis performed by each hub [5].

This paper explains how each node of ReCANcentrate manages the replicated star in a simple way that does not present the disadvantages of existing solutions. We firstly address the basic characteristics of 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 to be executed on hardware COTS components, and, finally, we conclude the paper.

ReCANcentrate basics

Due to its scarce error-containment and fault tolerance mechanisms, a CAN bus includes multiple single points of failure, i.e. multiple components whose failure cause the failure of the overall system [7]. The faults that may cause a generalized failure in CAN are: stuck-at-dominant, stuck-at-recessive, medium partition, bit-flipping and babbling idiot faults [8].

In order to eliminate all single points of failure from a CAN network we have developed a new replicated star topology called ReCANcentrate [5] (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 medium partition cannot provoke a network partition, but only manifests as either a stuck-at or a bit-flipping fault. A further improvement of ReCANcentrate concerns the treatment of babbling-idiot faults, which could be achieved in a relatively simple way [8].

In order to prevent that an error in one star leads the traffic in both stars to evolve in a different way, making data replication more complex to manage, both hubs exchange their traffic through the interlinks and perform a special AND coupling [5] to create a single communication 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.

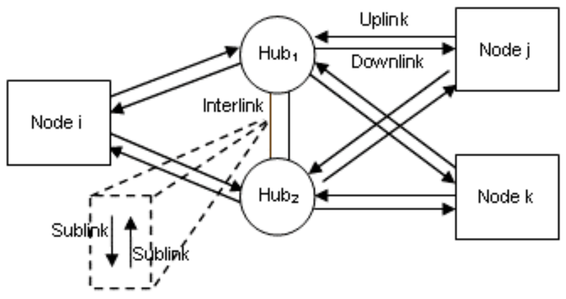

Figure 1: Architecture of ReCANcentrate

When a hub fails, it can exhibit three different types of failures. First, it may unfairly isolate non-faulty ports, what will not generate errors that propagate to other ports. Second, it may fail independently transmitting stuck-at or bit-flipping bits through any of its ports. This second type of failure embraces all faults that lead the hub to generate erroneous bits, as well as faults that lead the hub to stop performing fault-treatment actions. Note that a hub has not the capacity of building CAN frames [7].

A hub that transmits erroneous bits is confined at two different levels. Firstly, each hub is able to detect errors in each 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

The third type of failure a hub may exhibit occurs when it erroneously decides to stop coupling the contribution of the other hub with its own contribution, so that the single communication domain is not enforced. The same faulty situation happens if all the incoming sublinks of a hub fail.

To enforce data consistency when this kind of failure occurs, we are devising mechanisms that prevent any node from communicating as long as the fault is not detected and treated. Moreover, these mechanisms will ensure that all non-faulty nodes are consistently informed about whether the hubs have been able to reestablish the single communication domain, or it is only possible to use both stars independently. This paper focuses on how nodes manage the replicated star when the single communication domain is enforced. How to treat a permanent or transient hub decoupling is out of scope.

1. Media management in ReCANcentrate

1.1. Simplification of the media management

Since hubs are coupled, the set of hubs of ReCANcentrate can be seen as a single hub that provides a single communication domain. 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.

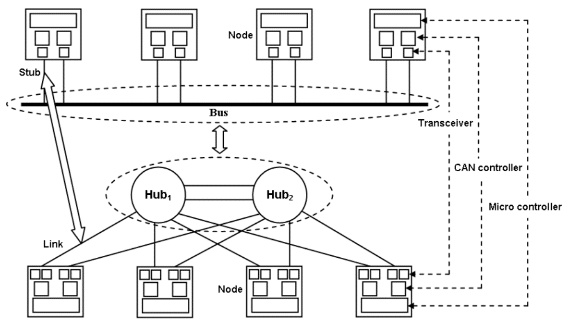

Figure 2: Bus & replicated star analogy

The management of the replicated media can be reduced to 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 scheme needed to support it in [5]. The node is constituted by COTS components only: two CAN controllers and a micro-controller (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 [5].

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 the *transmission controller* and the node will no longer receive its own messages, which, nevertheless, has no negative consequence.

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 this reception by its two CAN controllers.

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 fulfillment 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.

In what concerns how a node detects a fault in a given star, e.g. a failure in the link that attaches it to a given hub, and stops communicating through it, recall that a CAN controller includes a *Transmission Error Counter* (TEC), a *Reception Error Counter* (REC) [3] and, sometimes, a threshold for them, called *error warning limit*. Whenever any of the error counters of a CAN controller reaches the referred limit, the node stops using it.

Since both stars form a unique communication channel even in presence of faults in the media and nodes, this eliminates the need for each node to deal with discrepancies between channels, which is difficult and typical in other replicated media schemes. In contrast, a fault can only lead a node to observe that its two controllers differ in the vision of the same channel. To better understand this, next we analyze the faults that may occur in the communication subsystem and the discrepancies they provoke between the controllers of a node.

1.2. Faults and discrepancies

We differentiate between faults occurring at the media, i.e. at any hub or any link (transceivers, connectors and cables), and at controllers.

Media faults only manifest as the generation of syntactically incorrect bit values, which block the channel until they are isolated (stuck-at-recessive bits can be considered as instantaneously isolated). Thus, as long as a media fault is not confined, it is impossible that any controller notifies about a transmission or a reception and hence, no discrepancy between controllers 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 [9]. In these scenarios the atomic broadcast is violated, even when there is a single communication domain. However, it is not compulsory that a node of ReCANcentrate deals with any of these *inconsistent scenarios*. First, because their probability of occurrence have been controversial [10]. Second, because they are not a new problem introduced by the use of media replication, but an old problem of CAN that can be solved using any of the modifications or additions to CAN that have been already proposed [9]. 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 the *inconsistency scenarios* of CAN from our replicated media management.

Once a media fault is isolated, the following types of situations are possible. First, if the erroneous bits were generated by a hub, then it was isolated by the CAN controllers these bits were transmitted to, as well as by the corresponding interlink ports of the other hub, as already explained. A controller that isolates a hub will not notify its node of any further *delivery event*. Second, if the bits were generated by a cable, connector or transceiver used to attach a controller to a given hub, the corresponding hub port was disabled, so that the controller does not notify its node of any further transmission. Moreover, in this case, the controller also will eventually reach its *error warning limit* and the node will stop using it. However, notice that once the controller is isolated, it may continue receiving frames sent by other controllers, as long as it does not reach the *error warning limit* and there is at least one controller that acknowledges the frames in their ACK slot. This has no negative consequences and it is simply due to the fact that the hub continues broadcasting through the downlink of a port it has isolated.

Taking into account all these considerations, we can assure that a media fault can only provoke what we call a *notification omission discrepancy*, i.e. it can only lead a node to observe that only one of its CAN controllers informs about the occurrence of a *delivery event*.

Regarding faults happening at controllers, we analyze their effects following the well-known categorization of failures proposed in [11]. We distinguish between *crash* and *byzantine* controller failures. When a controller exhibits a *crash failure*, it stops performing any action, so that it will not generate errors on the network and will notify its node about nothing. Thus, the node will observe a *notification omission discrepancy* thereafter. In the case of presenting a *byzantine failure*, the controller fails arbitrarily with no restrictions either in the value domain or in the time domain. A *byzantine* controller failure manifests at the side of the network by generating stuck-at or bit-flipping bits, as well as semantically incorrect frames. If it generates erroneous bits, it will cause the same effects on the communication subsystem as a fault in any of the components that attach it to a given hub port and, hence, its node will only observe *notification omission discrepancies*. Semantically incorrect frames do not provoke discrepancies, but lead nodes to receive incorrect data as long as the hubs do not confine the failure. Once isolated, the controller acts as if it was isolated due to any of the failures already explained. Finally, when a *byzantine* failure leads a controller to deliver notifications that are arbitrarily incorrect, then not only *notification omission discrepancies* can occur, but also what we call a *notification non-omission discrepancy*. This 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.

1.3. Treatment of discrepancies

Each node treats discrepancies between the visions its two controllers have of a delivery event as follows. In what concerns a *notification omission discrepancy*, it can be provoked by both a fault in the media and any kind of failure of a controller, as explained before. Thus, when such a *discrepancy* occurs, it is not possible to elucidate which controller is faulty (or has problems for communicating). To overcome this problem, 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 correct, it means that the controller that omitted it is faulty or was isolated by its hub due to a media fault. Thus, to accept the frame is correct because it allows the node to tolerate the fault. If the notification was actually incorrect, to accept it is wrong, but this situation can exclusively be provoked by a byzantine controller fault and we are not obliged to deal with it. This is because controller faults are an old problem of communication subsystems that we have not introduced, e.g. in a typical non-redundant CAN bus, the controller of a node might forge notifications of transmissions and of receptions. Moreover, since we do not diagnose the correct controller (which omitted the notification) as faulty, at least we do not unfairly penalize it.

In other replicated media schemes the decision of what to do when observing omissions cannot rely on a best-effort approach. This is because an omission would not happen between controllers but between channels and, thus, nodes should run an algorithm to reach a consensus about which channel is faulty.

Regarding *non-omission notification discrepancies*, the node can use them to diagnose byzantine controller faults to some extent. In particular, a byzantine controller fault can be diagnosed when the notification from a faulty controller coincides in time with a notification of the correct controller, and both notifications refer to a different frame. Since when this happens, the node cannot know a priori which controller is actually faulty, it must stop communicating and run an internal test in order to make a decision. This capacity of fault diagnosis is an advantage of our approach compared with other solutions. Especially with respect to those that use only one CAN controller [6], since they cannot detect controller faults by means of a simple comparison.

2. Replicated media management routines

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

Each controller is marked as playing one of the following roles: *transmission controller* or *reception controller* (Section 3.1). A controller can also be marked as *non-active*, which indicates that it is not being used because it is faulty.

The driver 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. The driver assumes that all interrupts have the same priority, so that they cannot be nested.

2.4. Transmission and reception routines

The *transmission routine* and the *reception routine* respectively handle the *transmit interrupt* and the *reception interrupt*. 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 node's micro-controller.

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 slightly different internal delays. Thus, when a *delivery event* occurs, one controller will be the first to interrupt the micro 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 a *delivery event*, the routines must check whether or not the notifications of both controllers refer to the same frame, in order to detect a possible *notification non-omission discrepancy*. *Routine A* performs this checking. If affirmative, the routine leaves an indication to inform *routine B* that it has validated the correspondence between the notifications, so that *routine B* does not have to check it again. Otherwise, *routine A* indicates to the application that a *notification non-omission discrepancy* occurred.

Note 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 a *notification 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 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 for executing them must not exceed the time required for transmitting the shortest CAN *remote frame* [3].

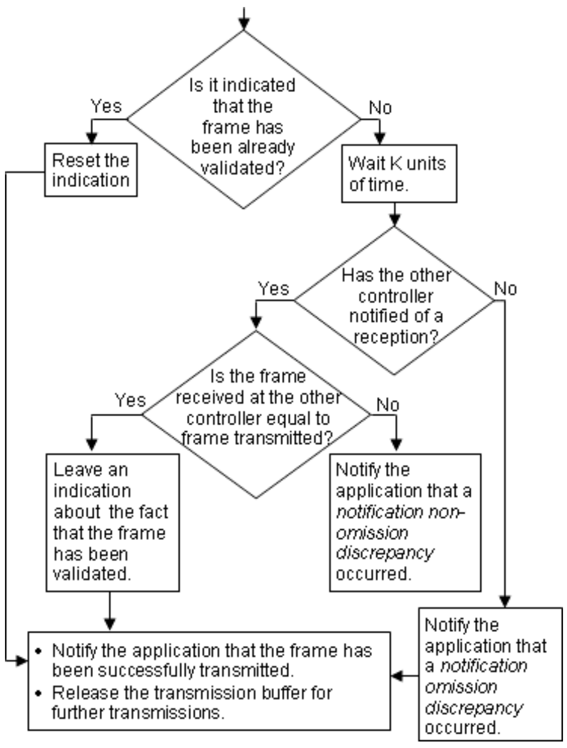
Is it indicated that the frame has been already validated?
Yes
No
Reset the indication
Wait K units of time.
Has the other controller notified of a reception?
Yes
No
Is the frame received at the other controller equal to frame transmitted?
Yes
No
Leave an indication about the fact that the frame has been validated.
Notify the application that a notification non-omission discrepancy occurred.
Notify the application that a notification omission discrepancy occurred.
• Notify the application that the frame has been successfully transmitted.
• Release the transmission buffer for further transmissions.

Figure 3: Transmission routine

Figure 3 depicts the general logic structure of the *transmission routine*. Recall that a frame transmitted by the *transmission controller* should be received by the *reception controller*. Thus, it is possible that a *reception routine* has been triggered and executed before. Therefore, the *transmission routine* checks if a former *reception routine* has indicated that it has validated the correspondence between the frame transmitted and received. If the result of this checking is affirmative, the routine plays the role of *routine B* and 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 *notification non-omission discrepancy* occurred. Finally, if the *reception controller* does not notify the expected reception, a *notification omission discrepancy* occurred; the routine indicates this condition and goes ahead.

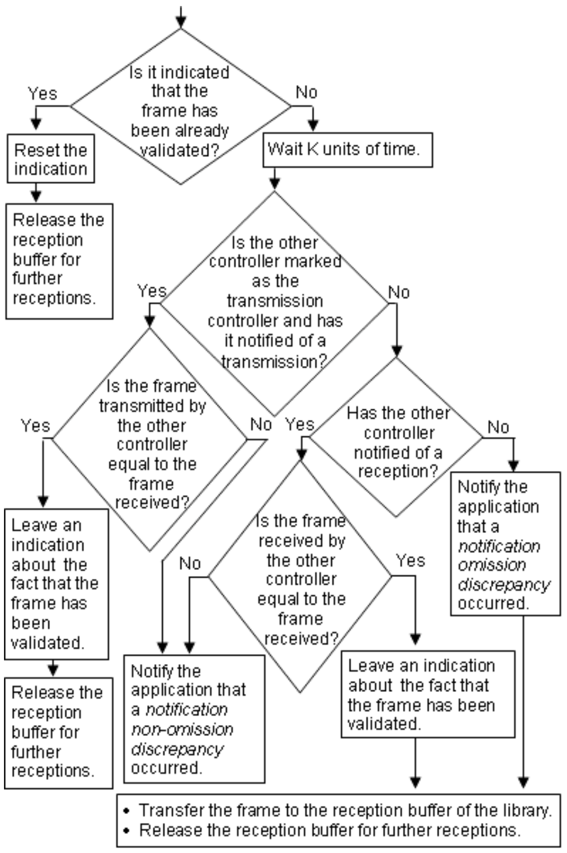
Is it indicated that the frame has been already validated?
Yes
No
Reset the indication
Wait K units of time.
Release the reception buffer for further receptions.
Is the other controller marked as the transmission controller and has it notified of a transmission?
Yes
No
Is the frame transmitted by the other controller equal to the frame received?
Yes
No
Yes
Has the other controller notified of a reception?
No
Leave an indication about the fact that the frame has been validated.
Is the frame received by the other controller equal to the frame received?
No
Yes
Notify the application that a *notification omission discrepancy* occurred.
Release the reception buffer for further receptions.
Notify the application that a *notification non-omission discrepancy* occurred.
Leave an indication about the fact that the frame has been validated.
• Transfer the frame to the reception buffer of the library.
• Release the reception buffer for further receptions.

Figure 4: Reception 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 when *reception routine* acts as *routine A*, it 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 other controller is marked as the *transmission controller* and it has notified a successful transmission. If affirmative, the routine is the responsible for testing the correspondence between the frame it handles and the frame transmitted. If negative, it definitively abandons the possibility of handling a reception of a frame of its own node.

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 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 driver, and releases the reception buffer of its corresponding controller.

2.5. Quarantine routine

The *quarantine routine* performs the actions needed when a controller is diagnosed as faulty. On the one hand, this routine is executed one time for each controller when a *notification non-omission discrepancy* occurs. On the other hand, it is triggered when a controller reaches its *error warning limit* (Section 3.1). When the routine executes due to a *notification non-omission discrepancy*, it resets the controllers and performs a test to determine which of them is faulty.

If the routine is triggered because a given controller reaches the *error warning limit*, it marks that controller as *non-active*, and further performs the following actions. If the other controller is also marked as *non-active*, the routine notifies the application that it is not possible to communicate with the other nodes. If the other controller is not marked as *non-active*, and the controller the routine is executing for was the *transmission controller*, the routine marks the other controller as the *transmission controller*. Additionally, if the application is waiting for the result of a transmission request, the routine notifies that such request was not granted, so that the application can request the transmission using the surviving controller.

3. Conclusions

One of the major objections against using CAN as the communication infrastructure of safety-critical distributed control systems is that its non-redundant bus topology lacks the appropriate error-containment and fault tolerance mechanisms. To provide these mechanisms, we developed a replicated star topology called ReCANcentrate, in which data is transmitted in parallel through both stars. A special coupling between both hubs creates a single communication domain or logical channel, so that each node quasi-simultaneously receives each bit from both hubs.

The enforcement of this single communication domain simplifies the management of the replicated traffic each node has to perform. In this paper we describe this management and propose a particular implementation using COTS components.

Specifically, each node includes two CAN controllers, each one connected to a different star. Differentiating duplicated from omitted frames is straightforward, since each frame is quasi-simultaneously broadcasted by both hubs, and faults in the media or at controllers cannot provoke that each hub broadcasts a given frame at a different instant of time. A media fault may only lead one or more nodes to observe that one of its controllers cannot access the channel or does not notify about the frames that are exchanged through the network. This is tolerated just by communicating through the controller that can correctly access the channel.

Byzantine controller faults may further lead its node to observe that both controllers do not agree on which frames are being exchanged in the network. We are not obliged to deal with these situations, since they are not introduced by the use of a replicated media scheme. Thus, the node manages these situations following a best-effort approach. Moreover, beyond the capacity of other replicated media architectures, a node of ReCANcentrate can diagnose, to some extent, controller malicious faults.

M. Barranco, manuel.barranco@uib.es
J. Proenza, julian.proenza@uib.es
Systems, Robotics and Vision Group
Dep. Ciències Matemàtiques i Informàtica,
Universitat de les Illes Balears (SPAIN)

L. Almeida, lda@det.ua.pt
Dep. de Electrónica e Telecomunicações,
Universidade de Aveiro (PORTUGAL)

4. References

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

[2] FlexRayTM. “FlexRay Communications System-Protocol Specification, Version 2.0”. 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.

[5] M. Barranco, L.Almeida and J. Proenza. “ReCANcentrate: A replicated star topology for CAN networks”. ETFA 2005, 10th IEEE International Conference on Emerging Technologies and Factory Automation. Catania, Italy, September 2005.

[6] 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.

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

[8] I. Broster and A. Burns. “An Analyzable Bus-Guardian for Event-Triggered communication”. Proceedings of the 24th Real-Time Systems Symposium (RTSS). Cancun, Mexico, December 2003.

[9] 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.

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

[11] 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.