Designing and Verifying Media Management in ReCANcentrate

Manuel Barranco, Julián Proenza
Dpt. Matemàtiques i Informàtica
Universitat de les Illes Balears, Spain
manuel.barranco@uib.es, julian.proenza@uib.es

Luís Almeida
DETI/IEETA
Universidade de Aveiro, Portugal
lda@ua.pt

Abstract

To overcome some dependability limitations of CAN that arise from its non-redundant bus topology, we have proposed a CAN-compliant replicated star topology, ReCANcentrate, whose hubs incorporate the necessary fault-treatment and fault-tolerance mechanisms. This paper presents ongoing work regarding the design and formal verification of the strategy each node of ReCANcentrate uses to manage the transmissions and the receptions on the replicated star, as well as to tolerate faults.

1 Introduction

Although there has been an interest in using CAN [1] and CAN-based protocols [2] as highly reliable communication infrastructures, one of the major limitations of CAN is that it relies on a non-redundant bus topology with scarce error-containment and fault tolerance mechanisms. To overcome this limitation, we developed a replicated star topology, ReCANcentrate [3], that includes two hubs (Figure 1). Each node is connected to each hub by a dedicated link containing an uplink and a downlink. Additionally, both hubs are interconnected by 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 or hubs, and to provide tolerance to hub and link faults [3]. ReCANcentrate is fully compatible with off-the-shelf (COTS) CAN components and any CAN-based application or protocol.

In ReCANcentrate the same data are transmitted in parallel throughout each of the media replicas to provide fault tolerance, i.e. each star is a channel that conveys a replica of the same data. However, it is necessary to provide mechanisms to allow each node to use the replicated media as a single CAN channel that enforces the same properties of CAN concerning *atomic broadcast* [4].

For this purpose, firstly, it is necessary that each node identifies duplicates and omissions. Duplicates are copies of the same frame received at different instants of time, each one through a different channel; whereas an omission occurs when a copy of a frame received from one or more channels is omitted from any of the others. The second step to achieve atomic broadcast is to properly manage duplicates and omissions. The node must discard duplicates, since they only carry redundant data. Besides, an omission could be a sign that the network is partitioned, so that nodes which are each one in a different partition can no longer communicate with each other. Thus, when it detects an omission, the node must initiate the appropriate treatment actions to ensure that all nodes agree on whether the frame for which the omission is detected has been consistently exchanged.

To solve these problems in time-triggered communications is relatively easy. Since the traffic is synchronized among the different media replicas, removal of duplicates and detection of omissions is done on the fly. However, since CAN is an event-triggered protocol, it does not provide any mechanism for synchronizing the frames in the different channels. In fact, due to the error-signaling and arbitration mechanisms of CAN, a bit error in one channel is enough for its traffic to evolve in a different way than in the other replicas.

Some approaches were proposed to achieve atomic broadcast when using replicated CAN busses [2] [5]. Any of them could be adopted for ReCANcentrate. But they are complex and expensive in terms of hardware and software, or they would limit the accuracy of the fault diagnosis each hub performs [3]. Fortunately, the hubs of ReCANcentrate exchange their traffic through the interlinks and couple with each other [3], so that both of them transmit the same value bit by bit in their downlinks. Thus, the traffic cannot evolve in a different way in both stars, i.e. the hubs can be seen as a single one that provides a single communication domain.

This paper focuses on how each node of ReCANcentrate manages the media in order to communicate as if there were a single CAN channel that ensures atomic broadcast. The paper briefly explains how nodes can take advantage of the hub coupling to manage the media and, then, outlines the characteristics of an on-going software implementation of the proposed management executed on hardware COTS components. This paper reflects part of the results we have obtained so far formally verifying the media management we propose.

2 Fault model

The physical layer of CAN implements a wired-AND function of every node contribution, thereby providing a dominant/recessive transmission [1]. Besides, its bit synchronization mechanism guarantees that nodes quasi-simultaneously observe every single bit on the channel.

Our fault model [3] differentiates between faults occurring at the media, i.e. at any hub or any link/interlink (transceivers, connectors and cables), and at CAN con-

 doi:10.1109/ETFA.2008.4638442

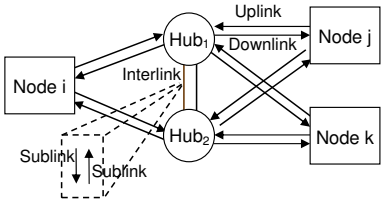

Figure 1. ReCANcentrate architecture

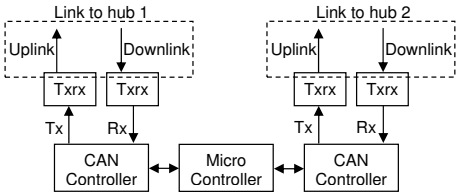

Figure 2. Node architecture

trollers. Faults at a link/interlink can only generate stuck-at or bit-flipping bits. Since a hub cannot build or buffer frames, it can basically fail by generating or propagating stuck-at or bit-flipping bits. However, a hub may also fail by erroneously deciding not to couple with the other hub.

Notice that the single communication domain is no longer enforced if all interlinks are faulty, or if a hub erroneously decides not to couple with the other hub. Fortunately, the probability of such situations is almost negligible since there are several interlinks and the hub could include internal redundancy to reduce the likelihood of an incorrect decoupling. Therefore, these situations are beyond the scope of this paper.

Regarding faults at CAN controllers, we distinguish between crash and byzantine failures. When a CAN controller crashes, it stops performing any action, so that it only delivers recessive bits to the network and it notifies its node about nothing, i.e. about no transmission, no reception, etc. A byzantine fault manifests arbitrarily in the value and time domains. At the side of the network, a byzantine CAN controller may generate stuck-at or bit-flipping bits, as well as syntactically correct but semantically incorrect frames. Additionally, it may forge or delay notifications to its node.

The hub detects and isolates stuck-at and bit-flipping faults at the port of origin [3]; so that it confines these faults happening at controllers, links, interlinks and at the other hub. Since the detection of semantically incorrect frames requires knowledge about the application executed at nodes, we postponed their treatment for future work.

Finally, each node is responsible for managing the notifications delivered by its CAN controllers. However, byzantine incorrect notifications are an old problem present in any communication subsystem, independently of whether or not it is based on a replicated media. Thus, they are somehow out of the scope of this paper and we only address them to some extent, as explained later.

3 Media management in ReCANcentrate

Due to the single communication domain hubs enforce, the management of the replicated media can be basically reduced to trigger each transmission towards one of the hubs only, while receiving from both hubs at the same time. 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 two COTS transceivers [3]. 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 (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. 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 notification can occur in two different manners. First, if the node successfully transmits a frame, the *transmission controller* and the *reception controller* notify of the transmission and reception of this frame respectively. Second, if the node receives a frame sent from another node, it is notified of this reception by its two CAN controllers.

To diagnose that it cannot communicate through a given hub, the node uses the *Transmission Error Counter* (TEC) and the *Reception Error Counter* (REC) [1] of the controller connected to that hub. Every CAN controller includes the TEC/REC and, usually, also a threshold for them, called *error warning limit*. When any of the error counters of a controller reaches this limit, the node rules it out and continues transmitting and receiving through the controller that has no problems for communicating.

The node identifies duplicates and omissions by observing the way in which its controllers notify of each *delivery event*. The node always expects duplicates in the form of a simultaneous notification from its two controllers, as explained before; whereas it detects an omission when a discrepancy between its controllers occurs so that only one of them notifies of a *delivery event*.

Concerning the way in which the node treats an omission, recall that both hubs are coupled and that each node is connected to each hub by a dedicated link. Thus, the network can only be partitioned if hubs become decoupled, a situation that is out of our fault model. Therefore the node can just accept the notification received from one of its controller as valid.

To better understand these considerations about duplicates and omissions, next we explain what discrepancies a node may observe between its two controllers, and how it treats them.

3.1 Faults and discrepancies

To analyze the discrepancies between the two controllers of a node, let us differentiate again between faults occurring at the media and at controllers. Media faults manifest as the generation of stuck-at or bit-flipping bits that usually generate errors that corrupt data.

In general, these errors block the channel. Consequently, as long as the hub/s do not contain them by dis-

abling the adequate hub ports, no controller notifies about a transmission or a reception and no discrepancy between controllers can take place. When the media fault is isolated, the channel becomes unblocked, but only the controllers that have not been isolated so far will communicate again. Thus, thereafter, each node that has an isolated controller will observe what we call a *notification omission discrepancy*, i.e. that only one of its controllers notifies of a *delivery event*.

Notice that there are some situations in which a media fault does not prevent all controllers from communicating, but that does lead them to inconsistently exchange frames. First, a frame is inconsistently exchanged in any of the error scenarios affecting the last-but-one bit of a frame that have been identified for CAN [6]. Second, a stuck-at-recessive fault may provoke an inconsistency if it prevents a controller from monitoring the traffic, or if it impedes that its contribution reaches its corresponding hub. For instance, if a downlink is stuck-at-recessive during the broadcast of a whole frame, the controller connected to that downlink will not observe it. The media management we propose takes into account these situations, but due to space limitations, this paper does not reflect how nodes treat them. Anyway, the probabilities of the scenarios due to errors in the last-but-one bit are controversial [7], whereas the probability of an inconsistency due to a stuck-at-recessive fault should be also very low.

Regarding what discrepancies can be provoked by controller faults, first notice that when a controller is crashed, its node will observe a *notification omission discrepancy* each time a new frame is exchanged. A controller that suffers from a byzantine failure also provokes a *notification omission discrepancy*, if it issues syntactically incorrect data to the media, or if it arbitrarily omits a notification. Additionally, it can also provoke a *notification non-omission discrepancy*, which occurs when both controllers of a node notify of a *delivery event*, but they do not agree on the frame the event is related to. This may occur if, for example, a controller forges notifications.

3.2 Treatment of discrepancies

When a node observes a discrepancy, it must decide which controller has problems. Then, the node rules out that controller and will not pay attention to its notifications anymore. When no controller has been ruled out so far, the node treats the discrepancies between their visions of a *delivery event* as follows.

A *notification omission discrepancy* can be provoked by a media fault, a crashed controller, or a byzantine controller. Hence, it is not possible to elucidate if the controller that has problems is the one that omits the notification or the other. To treat this, 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 *delivery event* actually occurred, it means that the controller that did not notify of it is faulty or has problems for communicating due to a media fault. Thus, to accept the frame allows tolerating the fault. If there were no *delivery event*, the notification was actually performed by a byzantine controller and to accept it is wrong. But, as said in Section 2, to fully treat byzantine faults is out of the scope of this paper. Moreover, since we do not diagnose the correct controller (which omitted the notification) as faulty, at least we do not unfairly penalize it.

Regarding *notification non-omission discrepancies*, the node can use them to diagnose byzantine controller faults to some extent. In particular, a byzantine controller can be detected when its notification coincides in time with a notification of the correct controller, and both notifications refer to a different frame. Although the node cannot know a priori which controller is actually faulty when this happens, it can 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 [5], since they cannot detect controller faults by means of a simple comparison.

4 Replicated media management driver

We are currently implementing the presented media management as a driver that abstracts away the details of the node structure and the media replication. This driver basically includes a transmission and a reception 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). A controller is also marked as *non-active* when it is ruled out. The driver is devised to use CAN controllers that at least include three interrupts: a *transmission interrupt*, which is originated when it successfully transmits; a *reception interrupt*, who is triggered when it receives a new frame; and an *error interrupt*, which is fired when it reaches the *error warning limit*. The driver assumes interrupts with the same priority, so that they are not nested.

The *transmission routine* and the *reception routine* respectively handle the *transmission interrupt* and the *reception interrupt* of a controller that is not ruled out. When a *delivery event* occurs, it is expected that each controller notifies of it by triggering one of these routines, which will be executed in the node's micro-controller. One of the controllers will be served first, while the execution of the interrupt triggered by the other controller will be pending. However, the two routines must cooperate to 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*, if both controllers are not marked as *non-active*, it is necessary to check that the notifications of both of them refer to the same frame. *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,

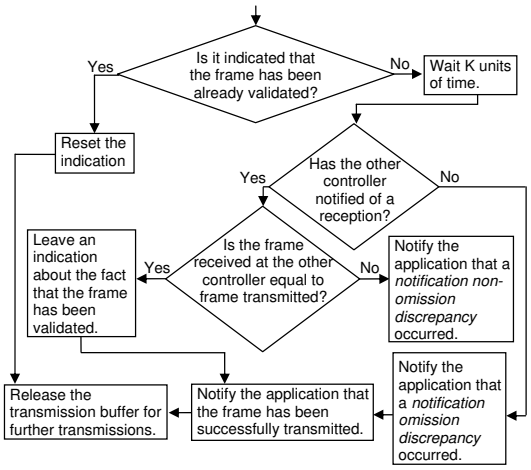

Figure 3. Transmission routine

routine A indicates to the application that a *notification non-omission discrepancy* occurred. Note that since *routine A* is executed first, it has to give enough time to allow the other controller to fire the interrupt that will launch *routine B*. If when this time expires, the other controller has not fired that interrupt yet, *routine A* assumes that a *notification omission discrepancy* occurred, and goes ahead to perform alone the actions needed to handle the *delivery event*. Concerning *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 is only possible if the micro executes the routines 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*.

Due to space limitations, we only outline the general structure of the *transmission routine* (Figure 3). Since a frame transmitted by the *transmission controller* should be received by the *reception controller*, the routine checks if a former *reception routine* has validated the correspondence between the frame transmitted and received. If negative, the routine 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, but that frame does not coincide with the transmitted one, the routine detects a *notification non-omission discrepancy*. Finally, if the *reception controller* does not notify the expected reception, the routine indicates that a *notification omission discrepancy* occurred.

Besides these two routines, the driver includes a *quarantine routine*, which is triggered when a *notification non-omission discrepancy*, or an *error interrupt* occur. In the first case, the routine first executes a test to determine which controller is faulty, whereas in the second case, it is not necessary since the faulty controller is the one who triggered it. The routine performs the operations needed to mark the faulty controller as *non-active*, and to use the surviving controller as the *transmission controller* if the controller that has been diagnosed as faulty is the current *transmission controller*.

5 Verification of the media management

To formally verify the correctness of the proposed replicated media management, we are modelling it using the model checker UPPAAL [8]. We want to verify that a ReCANcentrate network, in which nodes manage the media using the proposed driver, keeps the atomic broadcast properties [4] that CAN presents when the inconsistency scenarios pointed out in Section 3.1 do not occur: validity, agreement, at-most-once delivery, non-triviality and total order. Results we have obtained so far applying model checking indicate that such properties are fulfilled.

6 Conclusions

In ReCANcentrate data are transmitted in parallel through both its stars to provide fault tolerance. In this paper we outline how its nodes manage the replicated traffic by means of a driver, whose implementation and formal verification we are currently carrying out. A special coupling between both hubs creates a single logical channel. Thus, conversely to other CAN replicated media architectures, identifying duplicated and omitted frames is straightforward, since each frame is quasi-simultaneously broadcasted by both hubs even in the presence of faults.

References

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

[2] 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.

[3] 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.

[4] J. Rufino, P. Veríssimo, G. Arroz, C. Almeida, and L. Rodrigues, "Fault-tolerant broadcasts in CAN", *FTCS-28, The 28th International Symposium on Fault-Tolerant Computing, Munich, Germany*, 1998.

[5] 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.

[6] 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.

[7] 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.

[8] K. G. Larsen, P. Pettersson, and W. Yi, "UPPAAL in a Nutshell", *Int. Journal on Software Tools for Technology Transfer, 1(12):134152*, 1997.