Demonstrating the feasibility of media management in ReCANcentrate

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

Luís Almeida
IEETA / DEEC-FEUP
Universidade do Porto, Portugal
lda@fe.up.pt

Abstract

Star topologies are rising the interest of newer field-bus communication technologies like TTP/C and FlexRay, given the dependability advantages stars can offer. However, it is also possible to take advantage of a mature technology such as Controller Area Network [1] (CAN), while benefiting from stars. For that, we developed a CAN-compliant replicated star called ReCANcentrate. It includes two hubs that are coupled with each other, thereby forcing a single broadcast domain that allowed us to define, in a previous work, a strategy for each node to easily manage the replicated star. To demonstrate the feasibility of this management, this paper presents its on-going implementation as a driver to be executed at each node.

1 Introduction

In the field-bus communication domain there is a growing interest in using star topologies instead of buses as the communication infrastructure for highly-reliable distributed control systems, given the stars potential dependability benefits. For instance, we can find examples of this transition in in-vehicle systems, such as with TTP/C [2] and FlexRay [3]. Furthermore, it is also possible to take advantage of stars to make a mature technology such as Controller Area Network [1] (CAN) appropriate for the most demanding dependable systems. This is specially interesting, given the low cost of components and the know-how gained by the engineers during the last decades.

In particular, in order to overcome the limitations the CAN bus topology presents in terms of error containment and fault tolerance, we proposed a CAN-compliant replicated star topology called ReCANcentrate [4]. As depicted in Figure 1, ReCANcentrate includes two hubs and each node is connected to each of them by a dedicated link containing an uplink and a downlink. Both hubs are also interconnected by at least two interlinks, each of which contains two independent sublinks, one for each direction.

Each hub includes mechanisms to contain errors originated at nodes, links, interlinks or hubs [4]. Moreover, in ReCANcentrate each star is a CAN channel that conveys a replica of the same data in parallel to provide tolerance to hub and link faults. However, to use parallel CAN channels has an important drawback: due to the error-signaling and arbitration mechanisms of CAN, a bit error in one channel is enough for its traffic to evolve different than in the other replicas. Thus, it is not easy for each node to detect when frames received at different instants of time, each one through a different channel, are copies of the same frame (duplicates); as well as when a frame received from one channel is omitted from the others (omissions). Moreover, since channels are independent, the network can become partitioned if faults prevent nodes from communicating through different replicas [4].

To overcome these limitations, the hubs of ReCANcentrate exchange their traffic through the interlinks and couple with each other [4]. In this way, both hubs transmit the same value bit by bit in their downlinks, guaranteeing the traffic to be the same in both stars. Moreover, regardless the hub or hubs a node is able to communicate through, all nodes will be able to communicate with each other, thus preventing partitions [4]. This allowed us to define a strategy for each node of ReCANcentrate to easily manage transmissions, receptions and treat faults in the stars [5]. To demonstrate the feasibility of this management, we are implementing it as a software driver to be executed on each node of a real ReCANcentrate prototype. The current paper explains the driver's basics and the main technical problems it has to face. Special emphasis is put on the characteristics that hide the replicated star architecture, thereby allowing CAN-based applications and protocols to transparently use ReCANcentrate while at the same time benefiting from its higher dependability.

2 Media management basics

2.1 Management in the absence of faults

The physical layer of CAN implements a wired-AND function of every node contribution, thereby providing the *dominant/recessive transmission* property [1], which ensures that a dominant bit, '0', prevails over a recessive bit, '1'. Additionally, the CAN bit synchronization guarantees the *in-bit response* property, thanks to which nodes quasi-simultaneously observe every single bit on the channel.

The hubs of ReCANcentrate perform a special AND-coupling within a fraction of the bit time, thereby creating a single logical broadcast domain that keeps the two referred CAN properties. Thus, both hubs behave like one, transmitting the same value bit by bit in their downlinks.

This coupling allowed us to define a strategy for each node to easily manage the replicated star [5]. The node ar-

 doi:10.1109/ETFA.2009.5347227

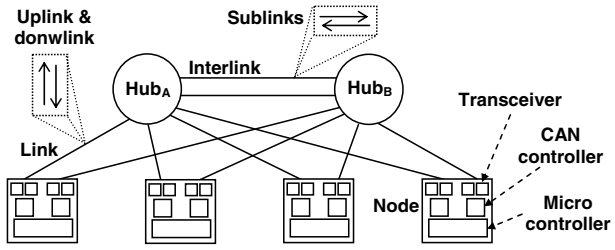

Figure 1. ReCANcentrate architecture

chitecture is depicted in Figure 1 as well. It is constituted by commercial-off-the-shelf (COTS) components only. It has one microcontroller and two CAN controllers, each of which is connected to one hub, using one transceiver for the uplink and another one for the downlink.

According to our media management strategy, the node transmits towards one of the hubs only, while receiving from both hubs at the same time. One controller acts as the *transmission controller* (*tx controller*), so that it is used to both transmit the frames of its node and receive frames sent by other nodes (the *tx controller* does not receive its own frames). The other controller, the *non-transmission controller* (*non-tx controller*), is used to receive frames transmitted by its own node and 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. Thus, in the absence of faults, the node manages transmissions and receptions as follows. First, if the node successfully transmits a frame, the *tx controller* and the *non-tx controller* notify of the transmission and reception of this frame respectively. Then, the node only needs to accept the notification of the transmission as valid and release the reception buffer of the *non-tx controller*. Second, if the node receives a frame sent from another node, it is notified of this reception by its two CAN controllers. When this happens, the node must merely consume the frame received at one of the controllers and, then, release the reception buffers of both controllers.

2.2 Management in the presence of faults

ReCANcentrate's fault model includes faults at nodes, links, interlinks or a hub that manifest, from a channel point of view, as stuck-at or bit-flipping streams [4]. Additionally, the model includes CAN controller faults that manifest as a *crash* from the node point of view, i.e. faults that lead a CAN controller to notify nothing to its node. The only fault assumption is that hubs remain coupled with each other using at least one non-faulty interlink.

Errors generated by a fault block the communication in both stars as long as hubs do not isolate it by disabling the appropriate hub ports [4]. Once isolated, if the fault affects a link or a CAN controller, it only prevents the corresponding node from communicating through the corresponding hub. But if the fault affects a hub, it can provoke that no node can communicate through that hub. Interlink faults do not prevent any node from communicating.

To tolerate a fault, it is necessary that any node that cannot communicate through a given hub as a consequence of that fault does not stop communicating through the other hub. As explained in [5], a node that cannot communicate through a given hub will observe a *notification omission discrepancy*: when a *delivery event* occurs, the node observes that the controller connected to that hub erroneously does not notify that event. Thus, in principle, the node can tolerate a fault by simply accepting as valid the transmission/reception notified by the controller that has no problems. Note that if the controller that cannot communicate is the *non-tx* controller, the node need not even diagnose that controller as faulty. However, if the controller that cannot communicate is the *tx controller*, the node must eventually diagnose it as faulty and, then, rule it out for communicating. Otherwise, the node will not be able to transmit anymore. For that, the node initiates a *transmission timer* when it requests a transmission. If the timer expires before the *tx controller* notifies of a successful transmission, the node rules it out and uses the other controller for transmitting/receiving. Additionally, we propose to rule out a CAN controller whenever its *Transmission Error Counter* (TEC) or its *Reception Error Counter* (REC) [1] reaches a given threshold. This only allows to rule out a controller that cannot communicate and that detects errors. But it enhances the node's fault diagnosis capabilities, which will allow us to improve in the future the management strategy to deal with a wider range of faults [5], e.g. CAN inconsistency scenarios [6] and forged transmission/reception notifications.

3 Driver basics

3.1 Driver architecture

We are currently implementing the media management as a driver that abstracts away the details of the node architecture and the media replication. Figure 2 depicts the basic structure of the driver. Specifically, we are implementing this driver to be executed on a dsPIC30F6014A Microchip^{TM}microcontroller [7], which includes two embedded CAN controllers: *CAN 1* and *CAN 2* in Figure 2. The driver also uses one of the timers of this microcontroller as the *transmission timer* (*tx timer* in Figure 2).

At the top part of the structure we can see the interface the driver provides to the application, i.e. the *driver interface*. It includes a set of primitives that abstract away the existence of two CAN controllers so that, from the application point of view, there is only one. In order to take full advantage of the features of the type of CAN controller embedded in the dsPIC30F6014A, it would be possible to offer all the primitives that can be found in the library Microchip^{TM}provides to interact with this kind of CAN controller. However, the objective of this driver is to demonstrate the feasibility of the media management we propose for ReCANcentrate. Thus, we decided to implement only a set of basic configuration and communication primitives, e.g. a primitive to configure basic CAN parameters such as the bit-rate, or the acceptance filters; a primitive to request a transmission, etc. Additionally, the

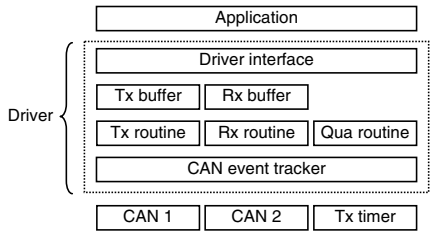

Figure 2. Basic driver structure

interface has a set of primitives that provide functionalities specific of ReCANcentrate, e.g. a primitive to check if any controller has been diagnosed as faulty.

Below the interface, we can find the *transmission buffer* (*tx buffer*) and the *reception buffer* (*rx buffer*). When the application requests to transmit a frame, the driver not only writes that frame to the hardware transmission buffer of the *tx controller*; but it stores a copy of that frame in the *tx buffer*. The driver needs this copy for different management operations, e.g. if the driver diagnoses the *tx controller* as faulty before that controller successfully transmits the requested frame, the driver automatically transfers a copy of that frame to the surviving controller. Regarding the *rx buffer*, it is a size-variable buffer that accommodates every frame that is received through ReCANcentrate. When the driver accepts a frame reception, it immediately copies the frame from the hardware reception buffer of one of the controllers and releases the hardware reception buffers of both controllers.

The major part of the driver functionality is implemented by the *management routines*: the *transmission routine* (*tx routine*), the *reception routine* (*rx routine*), and the *quarantine routine* (*qua routine*). Each of them is an interrupt service routine (ISR) that handles a given CAN controller or timer notification. The *tx routine* and the *rx routine* are executed when any of the two CAN controllers notifies of a transmission and a reception respectively. The *qua routine* is executed when the TEC/REC of any of the two CAN controllers reaches a specific threshold or when the *tx timer* expires. To simplify the routines we considered that they cannot be nested, which requires that all of them have the same execution priority.

However, none of these ISRs is directly triggered when a notification occurs. Instead, what a notification triggers is another ISR called *CAN event tracker*. This ISR has the maximum execution priority so that it can preempt any *management routine*. When a notification occurs, the *CAN event tracker* annotates that it has occurred and, then, triggers the execution of the appropriate *management routine* by generating an interrupt. The new *management routine* will be pending until the *CAN event tracker* and any previously preempted *management routine* end.

The *CAN event tracker* can be seen as a dispatcher that decides which routine must handle each notification. But its most important functionality is to annotate what notifications occur (and thus what routines it has triggered). For that, it uses a boolean variable for each type of notification, which we call *tracking variables*. They are needed to handle each *delivery event*, as explained next.

3.2 Management routines

Due to space limitations, we cannot explain the details of the *management routines*. Some additional information about their logic structure can be found in [5]. However, it is important to note here some aspects about how the *tx routine* and the *rx routine* manage a *delivery event*.

As explained before, when a *delivery event* occurs, it is expected that each CAN controller notifies about a transmission/reception, thereby triggering an execution of the *tx routine* or *rx routine* for each notification. If this actually happens, one of the routines will execute before the other, but both must collaborate with each other to handle the event. In contrast, if only one routine is triggered because a CAN controller omits due to a fault, that routine must handle the event alone. In any case, a routine must know if it has to collaborate with another routine to handle an event. For instance, imagine that two *rx routines* are triggered to manage the reception of a frame sent by another node. If they do not collaborate and each one of them merely transfers to the driver's *rx buffer* the frame received at its corresponding CAN controller, they will incorrectly accommodate two copies of the frame in the *rx buffer*. In order to elucidate whether or not the other routine has been launched and, thus, if it has to cooperatively handle an event, the *rx routine* (or *tx routine*) uses the *tracking variables* provided by the *CAN event tracker*.

However, we still have to check two technical aspects to complete the implementation of this cooperation. On the one hand, note that before a routine consults the *tracking variable* that indicates if the other routine is also going to be executed, it must wait enough time to allow the other routine to be triggered. We think that this time should be very short, of the order of a fraction of the bit time. On the other hand, it is worth noting that the routines that handle a given *delivery event* must finalize and reset their respective *tracking variables* before a new *delivery event* occurs. Otherwise, a routine corresponding to a new *delivery event* could incorrectly cooperate with routines that are handling a previous event. Specifically, the time for handling a *delivery event* must not exceed the time for transmitting the shortest CAN frame.

4 Hardware prototype

We built a ReCANcentrate prototype provided with two hubs, two interlinks and three nodes. Each hub is implemented using the VHSIC Hardware Description Language (VHDL) and synthesized in an FPGA; except its hardware interface, which is built using COTS transceivers. One UTP ethernet cable and a pair of RJ45 connectors are used to implement each link/interlink, which includes an uplink and an independent downlink (or two independent sublinks). Each uplink, downlink and sublink uses two-wire differential lines.

Each node consists of two different boards that are attached to each other. The first one is a printed board

called dsPICDEM™that Microchip™provides for evaluation purposes. It includes a dsPIC30F6014A microcontroller, which, as already said, has two CAN controllers. The second board is called *starLink* and we designed it specifically for ReCANcentrate. It basically includes COTS transceivers for connecting the dsPIC30F6014A's CAN controllers to the replicated star.

Apart from the fact that the dsPIC30F6014A includes two CAN controllers, we chose it for other reasons. First, it offers the possibility of specifying ISRs to handle notifications performed by each of its CAN controllers, which in particular allows triggering the *CAN event tracker* when necessary. Second, it also has many interrupt sources, each of which can be controlled by means of one of the dedicated interrupt registers, and handled by a user-defined ISR. The *CAN event tracker* can thus use otherwise unused interrupts to trigger (from software) the execution of the appropriate *management routine* in the form of an ISR. Third, the microcontroller allows specifying different ISR priorities, so that we can assign a higher priority for the *CAN event tracker* than for the *management routines*. Fourth, interrupt nesting can be enabled, which, combined with the *CAN event tracker*'s higher priority, allows the *CAN event tracker* to interrupt an executing *management routine* to update the *tracking variables*. And fifth, its CAN controllers notify when their TEC/REC reache a threshold, which is required for the fault-diagnosis strategy described in section 2.2.

Finally, we think that the speed of the dsPIC30F6014A is also appropriate to operate at the maximum CAN bit rate (1 Mbps). For that, as explained before, a *delivery event* must be handled before a new *delivery event* occurs. Most of the routines' instructions execute in a single cycle on the dsPIC30F6014A, with exceptions such as program flow changing instructions, which execute in two or three instruction cycles. With the oscillators that come with the dsPICDEM™board we cannot reach exactly the maximum CAN bit rate (1 Mbps), but we can come close. For instance, with a clock of 29.48 MHz we can reach around 983 Kbps and get an instruction cycle of about 33.92 ns. With this configuration we will be able to execute, on average, 1410 instructions before the shortest CAN frame is exchanged through the network. This should be enough, since in a preliminary implementation, without any optimizations, the management routines need less than 700 instruction cycles to handle a given event.

5 Conclusions and future work

This paper focuses on the work we are carrying out to demonstrate the feasibility of a media management approach we recently proposed for the ReCANcentrate replicated star. This work consists in implementing a driver (to be executed at each node of a real ReCANcentrate prototype) that hides the replicated star architecture, and that easily overcomes the typical problems a node has to face when using replicated CAN channels in parallel.

Once the driver is finalized, we will use it together with the ReCANcentrate hardware prototype to carry out some functional and performance tests to demonstrate the feasibility of the media management we have proposed. Regarding functional tests, we are going to evaluate whether or not the driver performs correctly in the absence and in the presence of faults. Specifically, we will consider faults affecting one of the CAN controllers or one of the links of a node, as well as one of the hubs. We want to experimentally verify that, even in the presence of faults, nodes can exchange a set of frames while keeping the *Logical Link Control* (LLC) level properties CAN presents [9] when inconsistency scenarios [6] do not occur: *validity*, *agreement*, *at-most-once delivery*, *non-triviality* and *total order*. In what concerns performance tests, we want to measure the delay with which the driver is able to diagnose a CAN controller, a link or a hub as being faulty. Additionally, we are going to evaluate what should be the minimum time a *tx routine* and a *rx routine* must wait before consulting the *tracking variables* to decide how they must handle an event. Finally, we will test if the driver is actually able to operate at the maximum CAN bit rate.

Besides these tests, we also plan to formally verify the correctness of the proposed management and to extend the driver functionality to cope with a wider range of faults, e.g. with CAN inconsistency scenarios [6].

6 Acknowledgement

This work was supported in part by the Spanish Science and Innovation Ministry with grant DPI2008-02195, and in part by FEDER funding.

References

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

[2] H. Kopetz, "Time-Triggered Protocols for Safety-Critical Applications", Presentation, March 2003.

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

[4] 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.

[5] M. Barranco, J. Proenza, and L. Almeida, "Designing and Verifying Media Management in ReCANcentrate", in *WFCS'08. IEEE Workshop on Factory Communication Systems, Dresden, Germany*, 2008.

[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] Microchip Technology Inc., *dsPIC30F6011A / 6012A / 6013A / 6014A Data Sheet - High-Performance, 16-Bit, Digital Signal Controllers*, 2006.

[8] Microchip Technology Inc., *dsPICDEM^{TM} 80-Pin Starter Development Board, Users' Guide*, 2006.

[9] 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.