Improved Fault Tolerant Broadcasts in CAN

Luís Miguel Pinho
ISEP, Polytechnic Institute of Porto
Rua Dr. António Bernardino de Almeida, 431
4200-072 Porto, Portugal
E-mail: lpinho@dei.isep.ipp.pt

Francisco Vasques
FEUP, University of Porto
R. Dr. Roberto Frias
4200-465 Porto, Portugal
E-mail: vasques@fe.up.pt

Abstract **- It is generally considered that the Controller Area Network (CAN) guarantees atomic broadcast properties through its extensive error detection and signalling mechanisms. However, it is known that these mechanisms may fail, and messages can be delivered in duplicate by some receivers or delivered only by a subset of the receivers. This misbehaviour may be disastrous if the CAN network is used to support replicated applications.**
In order to prevent such inconsistencies, a set of atomic broadcast protocols is proposed, taking advantage of CAN synchronous properties to minimise its run-time overhead. This paper presents such set of protocols, and demonstrates how they can be used for the development of distributed real-time applications.

I. INTRODUCTION

Controller Area Network (CAN) [1] is a fieldbus network suitable for small-scale Distributed Computer Controlled Systems (DCCS), being appropriate for transferring short real-time messages. The CAN protocol implements a priority-based bus, with a carrier sense multiple access with collision avoidance (CSMA/CA) MAC. In this protocol, any node can access the bus when it becomes idle. However, contrarily to Ethernet-like networks, the collision resolution is non-destructive, in the sense that one of the messages being transmitted will succeed.

This priority-based medium access control enables the use of CAN as the communication support for real-time distributed systems. Several studies on how to guarantee the real-time requirements of messages in CAN networks are available (e.g. [2]), providing the necessary pre-run-time schedulability conditions for the timing analysis of the supported traffic, even for the case of networks disturbed by temporary errors [3].

CAN networks also have extensive error detection/signalling mechanisms, which impose the retransmission of the message when an error is detected. However, it is known that these mechanisms may fail when an error is detected in the last but one bit of the frame [4]. This problem may cause messages to be delivered in duplicate by some receivers (*inconsistent message duplicate*), or, if the sender fails before re-transmitting the message, to the message being delivered only by a subset of the receivers (*inconsistent message omission*), leading to inconsistencies in the supported applications.

This misbehaviour may be disastrous if the CAN network is used to support replicated applications, since these applications require that replicated components provide the same results, when they are correct. Thus, the consistency of the delivered messages must be guaranteed by atomic broadcast mechanisms, which guarantee that messages are delivered by all (or none) of the component replicas' and that they are delivered only once. Furthermore, there is the need to agree also in the order by which broadcasts are delivered. Thus, it is necessary to provide protocols that guarantee these properties in spite of CAN inconsistencies, while at the same time preserving CAN real-time characteristics (thus allowing the offline analysis of the messages' response time).

This paper presents a set of atomic broadcast protocols intended to guarantee reliable real-time communication in CAN networks, in spite of inconsistency in message deliveries. The paper is structured as follows. The following Section presents a brief description of the problem of inconsistency in CAN communication. The proposed set of atomic broadcast is then presented in Section 3. Section 4 presents a numerical example, while Section 5 presents a comparison with other relevant approaches. An annex is also provided, with the specification of the proposed protocols.

II. INCONSISTENCY IN CAN COMMUNICATION

The use of CAN networks to support DCCS applications requires not only time-bounded transmission services, but also the guarantee of consistency for the supported applications. In spite of the extensive CAN built-in mechanisms for error detection and recovery [1], there are some known reliability problems [4], which can lead to an inconsistent state of the supported applications.

Such misbehaviour is a consequence of different error detection mechanisms at the transmitter and receiver sides. A message is valid for the transmitter if there is no error until the end of the transmitted frame. If the message is corrupted, a retransmission is triggered according to its priority. For the receiver, a message is valid if there is no error until the last but one bit of the received frame, being the value of the last bit treated as 'do not care'. Thus, a dominant value in the last bit does not lead to an error, in spite of violating the CAN rule stating that the last 7 bits of a frame are all recessive.

In Fig. 1, the Sender node transmits a frame to Receivers A and B. Receiver B detects a bit error in the last but one bit of the frame. Therefore, it rejects the frame and sends an Error Frame (requesting the frame retransmission) starting in the following bit (last bit of the frame). As for receivers the last bit of a frame is a 'do not care' bit, Receiver A will not detect the error and will accept the frame. However, the transmitter re-schedules the frame, as there was an error. As a consequence, Receiver A will have *an inconsistent message duplicate*. The use of sequence numbers in messages can easily solve this problem, but it does not prevent messages from being received in different orders, thus not guaranteeing total order of atomic broadcasts.

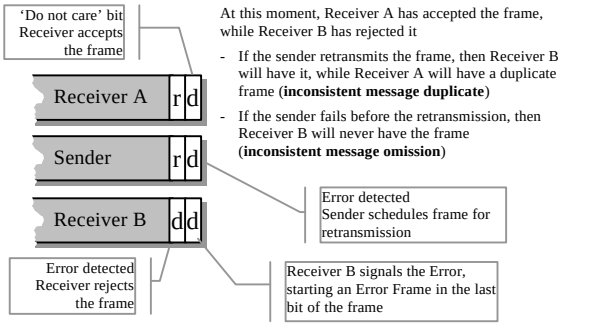


Fig. 1. Inconsistency in CAN.

On the other hand, if the Sender fails before being able to successfully retransmit the frame, then Receiver B will never receive the frame, although Receiver A has delivered it. This situation causes an *inconsistent message omission*. This is a more difficult problem to solve, than in the case of inconsistent message duplicates.

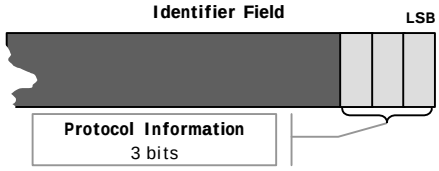
In [4], the probability of message omission and/or duplicates is evaluated, in a reference period of one hour, for a 32 node CAN network, with a network load of approximately 90%. Bit error rates were used ranging from 10^{-4} to 10^{-6}, and node failures per hour of 10^{-3} and 10^{-4}. For inconsistent message duplicates the results obtained were from 2.87×10^{1} to 2.84×10^{3} duplicates per hour, while for inconsistent message omissions the results ranged from 3.98×10^{-9} to 2.94×10^{-6} omissions per hour.

These values demonstrate that for reliable real-time communications, CAN built-in mechanisms for error recovery and detection are not sufficient. Thus, the use of CAN to support reliable real-time communications must be carefully evaluated and appropriate mechanisms must be devised.

III. FAULT-TOLERANT BROADCASTS IN CAN

The proposed set of broadcast protocols encompasses several protocols with different failure assumptions and different behaviours in the case of errors. The *IMD* (Inconsistent Message Duplicate) protocol provides an atomic broadcast that just addresses the inconsistent message duplicate problem. The *2M* (Two Messages) protocol provides an atomic broadcast addressing both inconsistent message duplicates and omissions, where messages are not delivered in an error situation. Finally, the *2M-GD* (Guaranteed Delivery) protocol is an improvement of the *2M* protocol, which guarantees the message delivery, if at least one node has correctly received it. The *Unreliable* protocol is a simple broadcast protocol that does not any guarantees.

These atomic broadcast protocols provide the system engineer with the possibility of trading efficiency by reliability, since they can be simultaneously used in the same system. The *IMD* protocol uses less bandwidth, but it does not cover the inconsistent omission failure assumption. On the other side, the use of protocols with higher assumption coverage (e. g. the 2M protocol) introduces extra overheads in the system. Hence, streams with higher criticality may use protocols with higher assumption coverage, while streams with lower criticality may use lighter protocols.



Protocol Bits	Message Type	
0 0 0	Data Msg.	*2M-GD* Protocol
0 0 1	Confirmation Msg.	
0 1 0	Retrans. Msg.	
0 1 1	Data Msg.	*2M* Protocol
1 0 0	Confirmation Msg.	
1 0 1	Abort Msg.	
1 1 0	*IMD* Protocol	
1 1 1	Unreliable Protocol	

Fig. 2. Identifier field and protocol information.

The proposed atomic broadcast protocols use the less significant bits of the frame identifier (Fig. 2) to carry protocol information, identifying the type of each particular message without interfering with the message criticality (defined by the most significant bits of the frame identifier).

Knowing that CAN frames are simultaneously received in every node, the atomic broadcast properties are guaranteed by delaying the deliver of a received frame during a specific (bounded) time. The approach is similar to the Δ-protocols [5], where, in order to obtain order, message delivery is delayed during a specific time (Δ). The difference is that, in the proposed approach, delivery delays are evaluated on a stream by stream basis, increasing the system throughput, as messages are delayed accordingly to their worst-case response times.

A) Failure Assumptions

In the proposed protocols it is assumed that:

- A single message can be disturbed by at most k_{dup} duplicates. As the probability of an inconsistent message duplicate is approximately 10^{-4} (the transmission of 2.87×10^{7} messages per hour results

in, at most, 2.84 x 10^3 duplicate messages [4]), it is not foreseen the necessity of a k_{dup} greater than 2.

- During a time *T*, greater than the worst-case delivery time of any message in the network, at most one single inconsistent message omission occurs. Considering the existence of 3.98 x 10^{-9} to 2.94 x 10^{-6} inconsistent message omissions per hour [4], the occurrence of a second omission error in a period *T* of, at most, several seconds has an extremely low probability.
- There are no permanent medium faults, such as the partitioning of the network. This type of faults must be masked by appropriate network redundancy schemes.

B) IMD Protocol

The *IMD* protocol provides an atomic broadcast that just addresses the inconsistent message duplicate problem. In order to guarantee that duplicates are correctly managed, every node, when receiving a message marks it as unstable, tagging it with a $t_{deliver}$ stamp (current time plus a $d_{deliver}$ delay). If a duplicate is received before $t_{deliver}$ (Fig. 3), the duplicate is discarded and $t_{deliver}$ is updated (since in a node not receiving the original message $t_{deliver}$ refers to the duplicate).

For the transmitter (if it also delivers the message), as the CAN controller will only acknowledge the transmission when every node has received it correctly (no more retransmissions), there will be no duplicates. Thus the transmitter can deliver the message after its $d_{deliver}$.

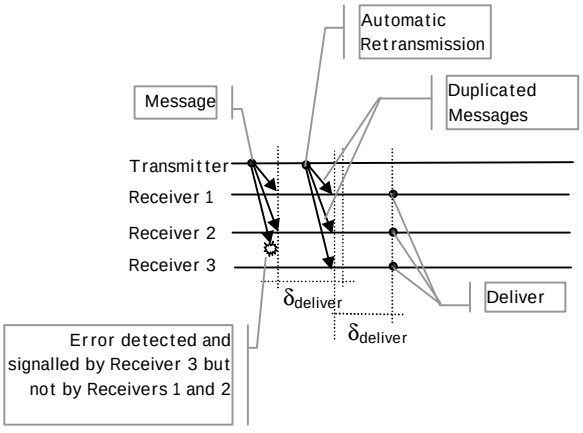


Fig. 3. Inconsistent message duplicate.

C) 2M Protocol

The *2M* protocol addresses both inconsistent message duplicates and inconsistent message omissions, guaranteeing that either all or none of the receivers will deliver the message. For the latter, not delivering a message is equivalent to a transmitting node crash before sending the message.

In the *2M* protocol, a node wanting to send an atomic broadcast transmits the data message, followed by a confirmation message, which carries no data. A receiving node before delivering the message, must receive both the message and its confirmation. If it does not receive the confirmation before $t_{confirm}$ (Fig. 4 presents an example of an inconsistent message), it broadcasts the related abort frame. This implies that several aborts can be simultaneously sent (at most one from each consumer node). A message is only delivered if the node does not receive any related abort frame (until after $t_{deliver}$) since a node receiving the message but not the confirmation, does not know if the transmitter has failed while sending the message, or while sending the confirmation. In Fig. 5 an example of this situation (for an inconsistent confirmation message) is presented.

When a message is received, the node marks it as unstable, tagging it with $t_{confirm}$ and $t_{deliver}$ stamps. A node receiving a duplicate message discards it, but updates both $t_{confirm}$ and $t_{deliver}$. As the data message has higher priority than the related confirmation (due to the protocol information field in the identifier), then all duplicates will be received before the confirmation. Duplicate confirmation messages will always be sent before any abort (confirmation messages have higher priority than related abort messages), thus they will confirm an already confirmed message.

The advantage of the *2M* protocol is that in a fault-free execution behaviour there is only one extra frame (without data) per broadcast. More protocol related messages in the bus will only be transferred in the case of an error (low probability).

Note that the transmission of an abort only occurs in the case of a previous failure of the transmitter. Therefore, from the failure assumptions presented (there is no second inconsistent message omission in the same period *T*), this abort will be free of inconsistent message omissions.

The transmitter can automatically confirm the message, since if it does not fail, every node will correctly deliver the message and the confirmation. The situation is the same as for the *IMD* protocol, since if the transmitter remains correct and delivers the message, then it will re-transmit any failed message.

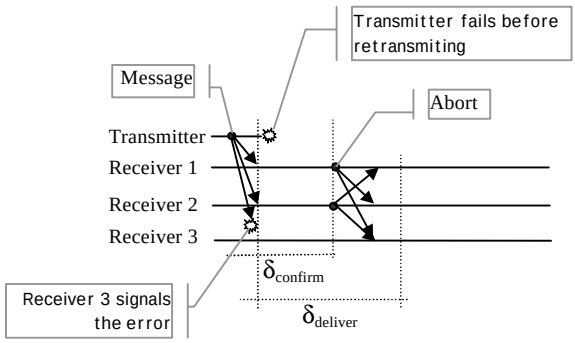


Fig. 4. Inconsistent message omission while sending the message.

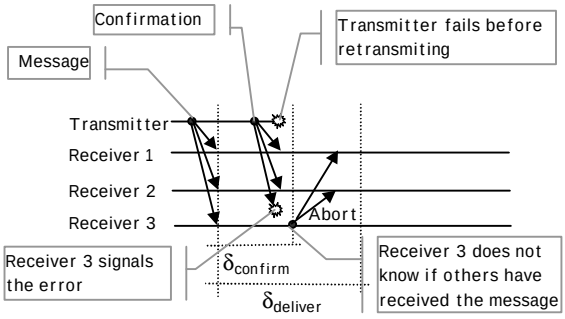


Fig. 5. Inconsistent message omission while sending the confirmation.

Using the *2M* protocol to send atomic multicasts, error situations are handled as follows:

- There is an inconsistent message omission while the transmitter is sending the data message, and it does not send the confirmation. In this case the nodes that have correctly received the message (but not the confirmation) disseminate an abort message.
- The transmitter correctly sends the data message, but a bit error causes this message to be duplicated in some of the nodes. As a duplicate message will always be sent before any confirmation, thus before the related message being delivered or aborted, it will be discarded (but $d_{deliver}$ and $d_{confirm}$ have been updated).
- The transmitter correctly sends the data message, but it crashes before sending the confirmation. In this case no receiver gets the confirmation, thus they do not deliver the message.
- The transmitter correctly sends the data message, but a fault causes the confirmation to be inconsistent. In this case some of the receivers will not receive the confirmation, thus they will abort the message, even for those nodes which receive a correct confirmation.
- The transmitter correctly sends the data message, but a bit error causes the confirmation to be duplicated in some of the nodes. As a duplicate confirmation will always be sent before any abort, it will confirm an already confirmed message.

The *2M* protocol can be modified to guarantee the delivery of a transmitted message to all nodes, if it is correctly received by at least one node. In the *2M-GD* protocol, nodes receiving the message but not the confirmation retransmit the message (instead of an abort). This protocol is however less efficient than the *2M* protocol (in error situations), since messages are retransmitted with the data field. It also requires an extra delay ($d_{deliver\_after\_error}$) to guarantee order of delivery, for the case of duplicate retransmissions.

D) Timing Analysis

In order to guarantee the timeliness requirements of real-time applications it is necessary to provide the model and assumptions for the evaluation of the message streams' response time, considering the use of the proposed protocols. As these protocols are based on delaying of the delivery phase, the response time analysis is constrained by the evaluation of these delays.

Due to the lack of space, this model is not presented here; the reader is referred to [6] where a set of pre-run-time schedulability conditions is presented, enabling the timing analysis of the supported communication protocols. These conditions allow determining the delays required for the proper behaviour of the proposed protocols, and also the worst- and best-case response times of each message stream.

One of the main targets of the proposed protocols is to introduce reliability in CAN communication, while at the same time preserving CAN real-time characteristics. Such target is achieved, since the predictability of message transfers is guaranteed [6].

IV. NUMERICAL EXAMPLE

In order to clarify the use of the presented model, a simple example is used. In this example (Fig. 6), a system where a distributed hard real-time application executes is considered. The system is constituted by four nodes, connected by a CAN network at a rate of 1 Mbit/sec.

The application is constituted by four tasks ($\tau_1..\tau_4$), which are spread over the nodes. As component replication is also used, then some of these tasks are also replicated. In this simple application, each task outputs its results to the following task.

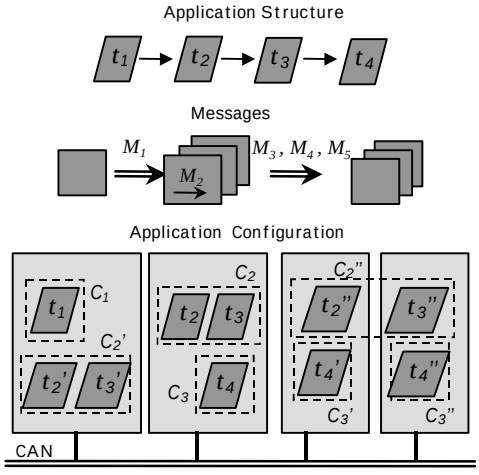


Fig. 6. Application example.

Table 1 presents each task's characteristics, while Table 2 presents the characteristics of the necessary message streams (all values are in milliseconds).

Table 1. Tasks' characteristics.

Task	Type	WCET	Period	Comp.	Nodes
τ_1	Per.	2	5	C1	1
τ_2	Per.	2	10	C2	1,2,3
τ_3	Spo.	3	10	C2	1,2,4
τ_4	Per.	4	15	C3	2,3,4

Table 2. Messages streams' characteristics.

Msg	Bytes	Period	From	To	Prot.
M_1	4	5	τ_1	τ_2,τ_2',τ_2''	2M-GD
M_2	8	10	τ_2''	τ_3''	IMD
M_3	6	10	τ_3	τ_4,τ_4',τ_4''	2M
M_4	6	10	τ_3'	τ_4,τ_4',τ_4''	2M
M_5	6	10	τ_3''	τ_4,τ_4',τ_4''	2M

Note that messages from τ_2 to τ_3 and τ_2' to τ_3' are internal to the node, since they are intra-component, and both tasks are in the same node. Since message M_1 is a *1-to-many* communication, the *2M-GD* protocol is used in order to guarantee that every replica of task τ_2 delivers the message. Therefore, there will be an extra confirmation message with the same period of M_1, but without data bytes. Since it is considered that an inconsistent message omission may occur, then it is also necessary to account for the possible 3 retransmission messages (one from each receiving node).

Message M_2 is internal to a component (although the component is spread between nodes 3 and 4), and it is a *1-to-1* communication. Therefore, it is sufficient to use the *IMD* protocol, since only duplicates are to concern. Messages M_3 to M_5 are messages from replicated τ_3 to replicated τ_4, therefore they need consolidation in every replica of τ_4. As this consolidation will mask node failures of the senders, then it is sufficient to use to *2M* protocol for the transmission of messages. Therefore there will be an extra confirmation message for each message sent (and possible abort messages).

In this analysis, the model of [3] is used, with the following error assumptions:

- a maximum of 2 errors in each 10 ms time interval, resulting from a bit error rate of approximately 10^{-4}, which is an expectable range for bit error rates in aggressive environments;
- possible existence of an inconsistent message omission during the period of analysis;
- possible existence of one duplicate in the transmission of a message ($k_{dup} = 1$);
- a Δ_{node} equal to 100 μS and a maximum deviation between clocks (e) of 100 μS.

The target of this example is to analyse the responsiveness of the proposed protocols, for both the response time and the delivery time of messages. *Response time* is considered as the time interval between requesting a message transfer until the message is fully received at the receiver side. *Delivery time* is considered as the time interval between requesting a message transfer until the Communication Manager delivers the message to the upper layers. If broadcast protocols are not used, these times are equivalent, as it can be assumed that messages are delivered when they are correctly received.

Table 3 presents the response time for each message stream and the network load when broadcast protocols are not used, that is, the *Unreliable* protocol is used instead of *IMD/2M/2M-GD* protocols. R_m^{NP} represents the worst-case response time (NP: no protocols), P is the periodicity and C_m is the actual time taken to transmit a message. U is the network utilisation.

Table 3. Messages' response time without protocols.

Msg	P	C_m	R_m^{NP}
M_1	5	0.089	0.519
M_2	10	0.127	0.630
M_3	10	0.108	0.741
M_4	10	0.108	0.852
M_5	10	0.108	0.852
U		6.590 %	

As it can be seen, the worst-case response time of messages is considerably greater than its actual transmission time. Although interference from higher priority messages is one of the factors leading to such difference, the main factor is the network bit error rate. For instance, a message of stream M_1 in an error free environment would have a worst-case response time of 0.219 ms. The possible existence of errors in the network more than duplicates its worst-case response time, even when broadcast protocols are not used.

Table 4. Protocol-related delays.

Msg	Prot.	$\delta_{confirm}$	$\delta_{deliver}$	$\delta_{del\_aft\_er}$
M_1	*2M-GD*	0.350	0.969	0.389
M_2	*IMD*	-	0.848	-
M_3	*2M*	0.901	2.013	-
M_4	*2M*	1.065	2.341	-
M_5	*2M*	1.229	2.558	-

Table 5. Messages' delivery time considering protocols.

Msg	R_m^{MP}	W_m	B_m	W_m/R_m^{MP}
M_1	0.519	3.394	1.058	6.54
M_2	0.959	2.655	0.975	2.77
M_3	1.070	3.984	2.121	3.72
M_4	1.234	4.640	2.449	3.76
M_5	1.287	5.074	2.666	3.94
U		9.09 %		

Tables 4 and 5 present the messages' delays and delivery times considering the use of the proposed broadcast protocols. R_m^{MP} represents the worst-case response time of a message stream when broadcast protocols (MP) are considered. W_m and B_m are, respectively, the worst- and best-case delivery time for message stream M_m.

As it can be seen in Table 5, the worst-case delivery time is greater than the related worst-case response time, because apart from the broadcast-related introduced delays, it is assumed that each message may be disturbed by one duplicate. For instance, the worst-case delivery time for message stream M_5 is not only given by the message stream response time plus its $d_{deliver}$, but also by summing an extra $d_{confirm}$ due to the possible existence of a message duplicate.

The last column of Table 5, presents the ratio worst-case delivery time/worst-case response time, when considering the use of broadcast protocols. It is obvious that the *IMD* protocol is the one that introduces smaller delays (Message M_2), while the *2M-GD* protocol is the one with the higher delays (Message M_1). Therefore, the system's engineer can use this reasoning to better balance reliability and efficiency in the system. Moreover, the

broadcast protocols increase network utilisation less than 50%, since broadcast-related retransmissions only occur in inconsistent message omission situations. Although this network load increase is still large, it is much smaller than in other approaches, and it is the strictly necessary to cope with inconsistent message omission using a software-based approach. Moreover, the real-time capabilities of CAN are preserved, since predictability of message transfers is guaranteed [6].

V. COMPARISON WITH OTHER APPROACHES

The problem of inconsistent messages in CAN networks has been given some research in the last years. In [4], a set of fault-tolerant broadcast protocols is proposed, which solve the message omission and duplicate problems. The RELCAN protocol is similar to the *2M-GD* protocol, being based in the transmission of a second data-free message (CONFIRM message), to signal that the sender is still correct. If this confirm message does not arrive before a specific timeout (the way to determine this timeout is not presented), the message is retransmitted. However, this retransmission is performed using a lower layer protocol (EDCAN), which is based in the retransmission of messages by every node in the system (that has correctly received the message). When a node receives a retransmission, it will retransmit it again (even if it already has retransmitted the original message). This behaviour leads to a huge number of messages in the network. Although the authors refer the possibility of several identical messages being clustered in the bus (all transmitted at the same time), this situation can not always be assumed. It is possible that some of these messages are not simultaneously transmitted, since sender nodes have distinct processing delays. Therefore, the worst-case response time grows exponentially with the number of stations in the network, which is not the case for the *2M-GD* protocol.

In the RELCAN protocol, the transmission request of the CONFIRM message is only made after receiving information from the CAN controller that the data message has already been sent. This two-phase approach is necessary to guarantee that there is no order inversion, that is, the CONFIRM message is only sent after the related data message. In the 2M (and 2M-GD) protocol this non-inversion guarantee is provided by giving to the confirmation message a lower priority than its related data message. Therefore, the request for transmission of both the data and confirmation messages can be atomically performed, reducing the worst-case response time of the related message stream.

On of the disadvantages of the RELCAN protocol is that it does not provide total order (thus it can not be used to achieve atomic broadcasts). When a data message is received, it is immediately delivered. Therefore, in the presence of inconsistent message errors, the order is not preserved. In [4], total order is addressed by the TOTCAN protocol. This protocol is also based in a two-phase approach, but the transmission of an ACCEPT message (similar to the CONFIRM message) is performed using the EDCAN protocol. Therefore, multiple retransmissions will occur in normal operation, even if no error occurs. Hence, the TOTCAN protocol incurs in a higher overhead, increasing significantly the network utilisation. For instance, when transmitting a fault-free message in a network with four nodes, in addition to the message there will be the ACCEPT message plus three retransmissions. Therefore, in the best-case (data message with 8 bytes), the overhead is approximately 150%, compared with the 40% of the *2M* protocol. In case of sender failure it does not deliver the message (it guarantees that the message is delivered by all or none of the recipients as the 2M protocol).

Another approach presented in the literature is to use a hardware-based solution [7] to prevent message inconsistencies. This approach is based in a hardware error detector, which automatically retransmits messages that could potentially be omitted in some nodes. This detector (SHARE) detects the bit pattern that occurs in an inconsistent message failure, and automatically retransmits the received frame, even if the transmitter handles this failure.

Although this hardware-based approach solves the inconsistent message omission problem of CAN, it does not provide solution to total order, as duplicates may occur (furthermore, inconsistent message omissions are transformed in inconsistent message duplicates). In order to achieve order, it is necessary to complement this mechanism with an off-line analysis [8]. In this, messages must be separated in hard and soft real-time. Only hard real-time messages have guaranteed worst-case response time inferior to the deadline, but it is necessary to use fixed time slots, off-line adjusting these messages to never compete for the bus.

In the approach proposed in this paper, a tool can easily perform the necessary off-line analysis, since it is based in the Response Time Analysis approach (as in [2]), and no message adjustment is required. However, it is clear that, by using a software-based approach in the proposed protocols, the network load increases and messages are delayed. Nevertheless, this is the strictly necessary to cope with inconsistent message omissions using a software-based approach, whilst preserving the real-time capabilities of CAN by guaranteeing the predictability of message stream transfers.

VI. CONCLUSIONS

In spite of its built-in error detection/signalling mechanisms, CAN networks may cause inconsistencies in the supported applications, as messages can be delivered in duplicate by some receivers or delivered only by a subset of the receivers. In order to preclude such incorrect behaviour, a set of atomic broadcast protocols has been proposed. Total order is guaranteed through the transmission of just an extra message (without data) for each message that must tolerate inconsistent message omissions. Only in case of an inconsistent message

omission (low probability) there will be more protocol-related retransmissions.

These protocols explore the CAN synchronous properties to minimise their run-time overhead, and thus to provide a reliable and timely service to the supported applications. The evaluation of these protocols demonstrates that the real-time capabilities of CAN are preserved, since predictability of message transfers is guaranteed.

VII. ACKNOWLEDGEMENTS

The authors would like to thank the anonymous referees for their helpful comments. This work was partially supported by FCT (projects DEAR-COTS 14187/98 and CIDER 33139/99).

VIII. REFERENCES

[1] ISO 11898. Road Vehicle - Interchange of Digital Information - Controller Area Network (CAN) for High-Speed Communication. ISO, 1993.

[2] Tindell, K., Burns, A. and Wellings, A. "Calculating Controller Area Network (CAN) Message Response Time". In Control Engineering Practice, Vol. 3, No. 8, pp. 1163-1169., 1995

[3] Pinho, L., Vasques, F. and Tovar, E. "Integrating inaccessibility in response time analysis of CAN networks". In Proceedings of the 3rd IEEE International Workshop on Factory Communication Systems, pages 77–84, Porto, Portugal, September 2000.

[4] Rufino, J., Veríssimo, P., Arroz, G., Almeida, C. and Rodrigues, L. "Fault-Tolerant Broadcasts in CAN". In Proc. of the 28th Symposium on Fault-Tolerant Computing, Munich, Germany, June 1998.

[5] Cristian, F., Aghili, H., Strong, R. and Dolev, D. "Atomic Broadcast: From Simple Message Diffusion to Byzantine Agreement". In Information and Control, 118:1, 1995.

[6] Pinho, L. and Vasques, F., "Timing Analysis of Reliable Real-Time Communication in CAN Networks". Proc. 13th Euromicro Conference on Real-Time Systems, Delft, The Netherlands, 2000.

[7] Kaiser, J. and Livani, M. "Achieving Fault-Tolerant Ordered Broadcasts in CAN". In Proc. of the 3rd European Dependable Computing Conference, Prague, Czech Republic, September 1999, pp. 351-363, 1999.

[8] M. Livani and J. Kaiser, "Evaluation of a Hybrid Real-Time Bus Scheduling Mechanism for CAN", in Proc. 7th Int. Workshop on Parallel and Distributed Real-Time Systems (WPDRTS ´99), San Juan, Puerto Rico, pp. 425-429, April 1999.

ANNEX: PROTOCOL SPECIFICATIONS

A) IMD Protocol Specification

Transmitter

```
1:  when atomic_multicast (id, data):
2:     send (id, data)

3:  when sent_confirmed (id, data):
4:     received_messages_set := received_messages_set ∪ msg(id,data)
5:     t_deliver(id) := clock + δ_deliver(id)

6:  deliver:
7:     for all id in received_messages_set loop
8:        if t_deliver(id) < clock then
9:           state(id) := delivered
10:       end if
11:    end loop
```

Receiver

```
1:  when receive (id, data):
2:     if id ∉ received_messages_set then
3:        received_messages_set := received_messages_set ∪ msg(id,data)
4:        state(id) := unstable
5:     end if
6:     t_deliver(id) := clock + δ_deliver(id)
```

```
7:  deliver:
8:     for all id in received_messages_set loop
9:        if state(id) = unstable and t_deliver(id) < clock then
10:          state(id) := delivered
11:       end if
12:    end loop
```

B) 2M Protocol Specification

Transmitter

```
1:  when atomic_multicast (id, data):
2:     send (id, message, data)
3:     send (id, confirmation)

4:  when sent_confirmed (id, message, data):
5:     received_messages_set := received_messages_set ∪ msg(id,data)
6:     state(id) := confirmed
7:     t_deliver(id) := clock + δ_deliver(id)

8:  deliver:
9:     for all id in received_messages_set loop
10:       if state(id) = confirmed and t_deliver(id) < clock then
11:          state(id) := delivered
12:       end if
13:    end loop
```

Receiver

```
1:  when receive (id, type, data):
2:     if type = message then
3:        if id ∉ received_messages_set then
4:         received_messages_set := received_messages_set ∪ msg(id,data)
5:         state(id) := unstable
6:        end if
7:        t_deliver(id) := clock + δ_deliver(id)
8:        t_confirm(id) := clock + δ_confirm(id)
9:     elsif type = confirmation then
10:       state(id) := confirmed
11:    elsif type = abort then
12:       if id ∈ received_messages_set then
13:          received_messages_set := received_messages_set - msg(id)
14:       end if
15:    end if

16: deliver:
17:    for all id in received_messages_set loop
18:       if state(id) = confirmed and t_deliver(id) < clock then
19:          state(id) := delivered
20:       elsif state(id) = unstable and t_confirm(id) < clock then
21:          send (id, abort)
22:          received_messages_set := received_messages_set – msg(id)
23:       end if
24:    end loop
```

C) 2M-GD Protocol Specification

Transmitter

```
1:  when atomic_multicast (id, data):
2:     send (id, message, data)
3:     send (id, confirmation)
```

```
4:  when sent_confirmed (id, message, data):
5:     receivedMsgSet := receivedMsgSet ∪ msg(id,data)
6:     state(id) := confirmed
7:     t_deliver(id) := clock + δ_deliver(id)

8:  deliver:
9:     for all id in receivedMsgSet loop
10:       if state(id) = confirmed and t_deliver(id) < clock then
11:          deliver( receivedMsgSet(id) )
12:       end if
13:    end loop
```

Receiver

```
1:  when receive (id, type, data):
2:     if type = message then
3:           if id ∉ receivedMsgSet then
4:              receivedMsgSet := receivedMsgSet ∪ msg(id,data)
5:              state(id) := unstable
6:        end if
7:        t_deliver(id) := clock + δ_deliver(id)
8:        t_confirm(id) := clock + δ_confirm(id)
9:     elsif type = confirmation then
10:       state(id) := confirmed
11:    elsif type = retransmission then
12:       if id ∉ receivedMsgSet then
13:          receivedMsgSet := receivedMsgSet ∪ msg(id,data)
14:       end if
15:       state(id) := confirmed
16:       t_deliver(id) := clock + δ_deliver_after_error(id)
17:    end if

18: deliver:
19:    for all id in receivedMsgSet loop
20:       if state(id) = confirmed and t_deliver(id) < clock then
21:          deliver( receivedMsgSet(id) )
22:       elsif state(id) = unstable and t_confirm(id) < clock then
23:          send (id, retransmission, data)
24:       end if
25:    end loop

26: when sent_confirmed (id, retrans, data): -- if retransmitted
27:    state(id) := confirmed
28:    t_deliver(id) := clock + δ_deliver_after_error(id)
```