Achieving Fault-Tolerant Ordered Broadcasts in CAN

Jörg Kaiser, Mohammad Ali Livani
University of Ulm, Department of Computer Structures, 89069 Ulm, Germany
{kaiser, mohammad}@informatik.uni-ulm.de

Abstract. The paper focuses on the problem to guarantee reliable and ordered message delivery to the operational sites of a CAN-Bus network. The contributions of the paper are firstly a hardware mechanism to handle rare failure situations and secondly, a protocol to guarantee the same order of messages on all nodes. After analyzing the error handling mechanism, we suggest a hardware extension to capture situations, which may lead to inconsistent views about the status of a message between the nodes. Based on this mechanism, which enhances the guarantees of the CAN-Bus with respect to reliable message transmission, we develop a deadline-based total ordering scheme. By carefully exploiting the properties of CAN, this can be achieved with very low additional message overhead.
Keywords. Real-time communication, fault-tolerance, reliable broadcast, CAN.

1 Introduction

Dependability is one of the most important design dimensions for many applications in the area of real-time control systems. A distributed real-time architecture supports in many ways the aspect of dependability as well as other related demands of process control like modularity and improved extensibility. Therefore, distributed systems, composed from a network of micro-controllers connected via a field-bus network become increasingly popular in process control and automotive applications. In these cooperative distributed systems, reliable coordination of distributed actions becomes a critical issue. Atomic broadcast protocols represent a mature technology to support this coordination of distributed actions in the presence of network and node failures. In contrast to general-purpose applications, real-time control systems put some additional challenging requirements to these protocols. Firstly, they have to meet temporal constraints dictated by the controlled external environment, usually given in terms of deadlines. This means that the communication has to be predictable and bounded in the temporal domain. Secondly, the nodes in these systems consisting of micro-controllers typically offer limited processing and memory capacity. Additionally, field-bus networks connecting these nodes generally exhibit a lower bandwidth compared to general purpose LANs. These constraints demand for efficient mechanisms, which carefully exploit the capability and functionality of the underlying hardware basis.

Reliable atomic broadcast protocols are widely used in dependable distributed computing. There is, however, a wide class of protocols, which are not designed for real-time systems and therefore completely lack respective properties [3, 4, 5]. Even those protocols which are explicitly designed for dependable real-time systems, like [6,7] cannot be used in a fieldbus environment straightforwardly. The reasons are mainly the constraints, set by low performance micro-controllers and low network bandwidth. In a control system composed from micro-controllers it is vital to free the host processor as much as possible from executing complex protocol algorithms. Equally important, these systems demand for minimizing the communication overhead usually necessary to achieve a consensus about the message status and order in a reliable broadcast protocol. Therefore, it is important to carefully analyze the properties of the network to exploit the functionality that is already available on the controller level.

The paper focuses on mechanisms to support reliable atomic broadcast on the CAN-bus. Among the established field bus networks, CAN constitutes an emerging standard and particularly supports a decentralized control architecture. Developed by BOSCH [1], CAN is primarily aimed at automotive applications which omit a single point of control or a master-slave communication relation often found in industrial automation. Because CAN was designed for reliable operation in a control environment, it has a number of comprehensive mechanisms to detect and handle communication faults. Particularly, CAN has a built-in mechanism which aims at achieving consensus about the success of a message transfer for all participants immediately, i.e. before the message is delivered to the host processor by the CAN communication controller. Thus, in most situations, the sender handles the fault transparently on the controller level by automatically discarding the faulty message and initiating retransmission. However, a careful analysis of the CAN-bus mechanisms with respect to atomic broadcast protocols in [2] reveals that in specific situations inconsistencies may arise which cannot be handled by the standard CAN protocol.

An atomic broadcast protocol provides reliable message transmission to all operational nodes under anticipated fault assumptions and guarantees that all nodes received the messages in the same order. The paper presents solutions to these two requirements. To provide reliable message transfer under omission and crash failures we describe a new hardware mechanism SHARE (SHadow REtransmitter), which guarantees an all-or-nothing property of message delivery. This means that a message is eventually correctly delivered or discarded at all operational nodes. Based on this atomicity property, we develop an ordering scheme based on deadlines. The specific feature of our scheme compared to other approaches [6, 19] is that it considers the coexistence of hard and soft real-time messages. While hard real-time messages are critical and must be delivered at some specified deadline, soft real-time messages may miss their deadlines under transient overload.

The achievement of the paper is to provide the atomic broadcast protocol for CAN with very low overhead for the host processors and the communication network. By using SHARE, error detection and retransmissions of messages is completely handled at the network controller level. Order is established by exploiting the CAN properties without additional communication overhead. Another important benefit of the protocol is that the temporal behaviour of the system is highly predictable and does not change with the number of nodes in the system.

The paper sets out with the problem analysis. It comprises a brief description of the basic CAN mechanisms, which are a prerequisite to understand the critical faults which lead to inconsistencies. These inconsistencies are analyzed in the subsequent section. Chapter 3 describes the SHARE hardware component to achieve consistent delivery of messages in spite of sender crashes. Based on these results, chapter 4 presents our mechanism to establish order between the messages based on their deadlines. Chapter 5 summarizes related work and a conclusion is presented in Chapter 6.

2 Problem Analysis

CAN handles arbitration and performs message validation and error signaling in a very specific way. Because this is exploited by our mechanisms, we will briefly introduce the basic CAN properties. CAN (ISO 11898 an 11519-1) is a broadcast CSMA-network targeted to operate in an automotive or industrial automation environment with speeds of up to 1 Mbit/sec, exchanging small real-time control messages. The CAN-specification [1] developed by BOSCH covers the functionality of layers 1 (physical layer) and 2 (data link layer) of the ISO/OSI protocol stack. CAN is a variant of a polled bus [8], which relies on three properties.

1. Every bit of a message will propagate to all connected nodes before the next bit will be sent, thus enabling all nodes to see the same bit value during a certain time window.
2. The sender monitors the bus at the same time it transmits. For every bit, a sender can check if the bus carries the signal level which was transmitted.
3. There are dominant and recessive signal levels. A single dominant signal level overrides any number of recessive signal levels.

If two stations start transmission at the same time, a collision will occur. Different from other CSMA-networks, however, collisions always lead to a well-defined signal level on the bus because of the property 1 and 3. In most implementations, the CAN-bus behaves like a wired AND circuit for all bit values sent at the same time, i.e. a “0” denotes a dominant and a “1” denotes a recessive value. This basic feature of the CAN-bus is exploited for:

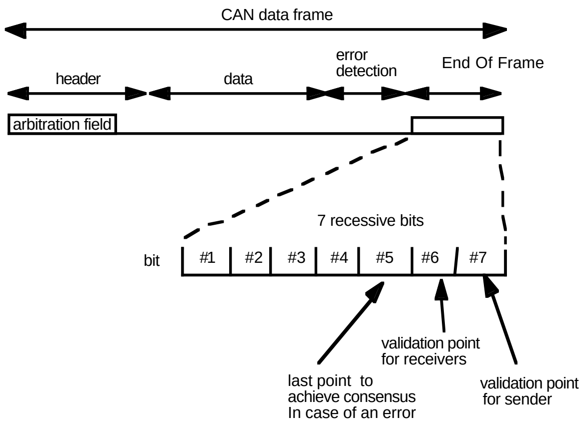
- Efficient use of available network bandwidth by providing a non-destructive arbitration of the bus based on the priority of messages.
- Immediate error detection, signaling and automatic retransmission of messages.

2.1 CAN arbitration

The priority-based non-destructive arbitration scheme assures that a collision does not destroy the messages on the bus, but the message with the highest priority will be transferred without further delay. Collisions are resolved during the arbitration process, i.e. when sending the arbitration field, which contains a unique ID for each message. If a node during the arbitration process sends out a recessive level but monitors a dominant level on the bus, it stops transmission because it knows that a message with higher priority is competing for the bus. The CAN arbitration mechanism thus acts as a global priority-based non-preemptive message dispatcher, guaranteeing that the message with the highest priority always is sent without delay. Additionally, the non-destructive arbitration scheme overcomes the drawback of general CSMA networks of low predictability and lost bandwidth because of collisions under high load conditions.

2.2 CAN error detection and fault handling

The second important feature of CAN is that it provides a validation mechanisms to achieve a consistent view about the status of a message at the end of every individual message transfer. Every message will be accepted or rejected by all participants. CAN has a comprehensive set of error detection mechanisms, which comprise bit monitoring, bit stuffing, cyclic redundancy checking and frame consistency checks [1]. Here, only the error signaling mechanism will be examined in more detail because it takes part in providing consensus about the message status in CAN. The error signaling mechanism is realized by exploiting bit monitoring and the CAN bit-stuffing mechanism. The CAN bit-stuffing mechanism always inserts an additional complementary bit in the bit-stream of a message when a string of 5 consecutive bits with the same value are detected. When a CAN controller (sender or receiver) detects an error locally it will invalidate the ongoing message transfer by sending out an error frame consisting of a string of 6 dominant bits. This violates the CAN bit-stuffing rule and consequently, will be detected by all other nodes, including the sender. As a result, the receivers will discard the current message from their local in-queues and start error signaling themselves. After error signaling is terminated, the sender will automatically retransmit the message. Thus, relating a corrupted message to its source and retransmitting it will be done at the controller level. For the host processor the automatic retransmission procedure is transparent.



CAN data frame
header
data
error detection
End Of Frame
arbitration field
7 recessive bits
bit
#1
#2
#3
#4
#5
#6
#7
validation point for receivers
last point to achieve consensus In case of an error
validation point for sender

Fig. 1. Schematic outline of a CAN data frame

To allow last moment frame error detection for nodes which are completely out of synchronization, CAN provides an end-of-frame field (EOF) consisting of 7 recessive bits (see Fig. 1). Nodes which are aware of receiving the EOF will disable the mechanism detecting a bit-stuffing violation. Only nodes which are not aware of receiving an EOF will detect a bit-stuffing error when sampling bit #6. They will start error signaling in bit #7 of the EOF.

If the receivers did not observe the start of an error frame until the bit #6 of the EOF, they assume that the message is transmitted correctly and deliver it to their respective host processors. Let us term the point when the receiver locally samples bit #6 the validation point for receivers. The error frame can be viewed as a variant of a negative acknowledge (NAK). If no NAK occurs, the message is assumed to be transmitted correctly.

2.3 Sources of inconsistencies

Now consider that some receivers detect an error just in the bit #6 of the EOF. These receivers will discard the message and start an error frame. Obviously the error frame can only start in the subsequent bit, i.e. the bit #7 of EOF. At that time all receivers which did not detect or were not affected by the error have already accepted and delivered the message. They basically ignore the error frame. To cope with this situation and to prevent that some receivers have obtained the message and some others have not, the sender of the message waits until the bit #7 of the EOF before it assumes a correct message transmission. In the situation sketched above, the sender would recognize the error frame and initiate retransmission. This leads to “inconsistent message duplicates” [2] of the message at those nodes, which previously accepted the message. If the message is retransmitted immediately, these inconsistent duplicates can easily be detected by a sequence number which consists of a single toggle bit [9] in its simplest form.

However, there may be situations in which a higher priority message “b” becomes ready to transmit on some node which now competes with the message “a” to be retransmitted. Because of the global CAN priority scheme, the message “b” (or any number of higher priority messages) will be sent before the retransmission of message “a”. In this situation, we encounter a problem affecting the order of messages. The nodes which have already received message “a” will see it before message “b” while the other nodes observe “a” after “b”. This is not solved by the raw CAN protocol nor by the mechanisms provided by popular CAN application level protocols [9, 10, 11].

The CAN protocol guarantees that if the sender remains operational, it will retransmit the message until it is eventually received by all nodes. In case of a sender crash, however, there will be nodes, which have received the message while others never will receive the message. This constitutes an “inconsistent message omission” [2]. In the following two chapters, we introduce the concepts to deal with sender crashes and inconsistent order of messages. We start with describing SHARE, a hardware component to handle inconsistent omissions transparently, i.e. without any additional load for the nodes in the system. Built on this, we sketch the protocol that provides a deadline-based ordering scheme.

3 Using Shadow Retransmitters to Mask Sender Crashes

As discussed previously, CAN provides reliable message transmission (sometimes with duplicates) unless a sender crashes after a frame was received only by a subset of the receivers. In this case the sender cannot retransmit the frame which results in a permanent frame-level inconsistency.

This problem is solved by adding dedicated nodes to the system, which act as ‘Shadow Retransmitters’ (SHARE). A SHARE, schematically sketched in Fig. 2, captures frames, which are transmitted over CAN and detect the situation when an inconsistent omission is possible. In these situations they behave just like the sender and transmit the frame *simultaneously* with the original sender. This is feasible because of the physical properties of CAN; i.e. multiple senders may transmit *identical* frames. Hence, if the original sender crashes, the SHARE will mask the fault and retransmit the message. The SHARE mechanism is transparent and can be used in a system to cope with inconsistent omissions without changing the existing system components or affecting the temporal properties of the system. Currently, a SHARE consists of a dedicated micro-controller with CAN interface and a simple additional state machine to detect a unique bit pattern on the bus. This bit pattern only is generated when a situation occurs which may lead to inconsistent omissions.

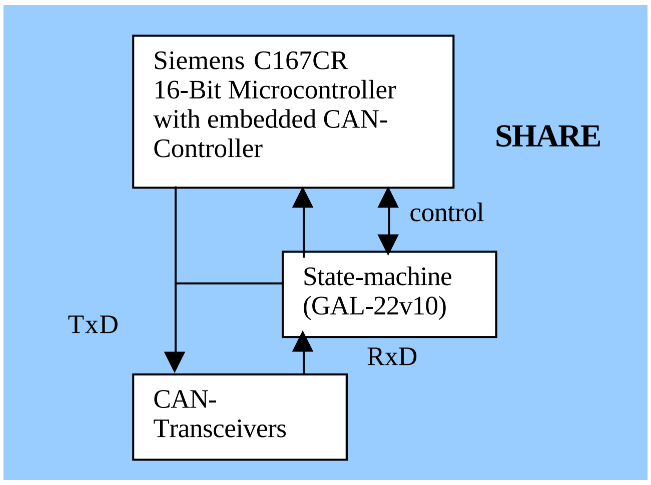


Fig. 2. Components of a SHARE

The detection of a possible inconsistent omission is based on a simple fact: a frame-level inconsistency is only possible if at least one non-faulty node observes an error, i.e. a dominant level in bit #6 in the EOF pattern. In this case, the sender is expected to retransmit the frame, thus a SHARE should also retransmit the frame.

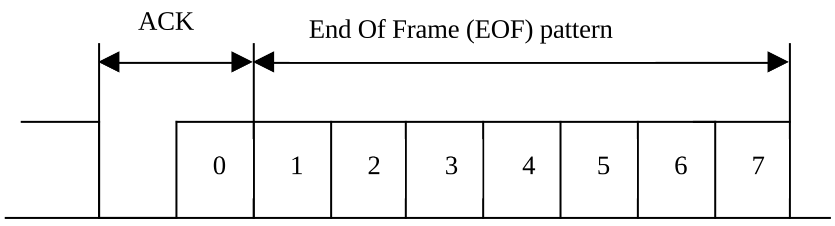


Fig. 3. Bit pattern terminating a CAN message frame

The terminating bit sequence of a CAN message frame comprises a total number of 8 recessive bits, one bit from the acknowledge field and seven bits of the EOF. It is shown in Fig. 3. As we stated before, retransmission by a SHARE is not allowed if an error occurs before bit #6 of the EOF. These situations are handled consistently by the standard CAN protocol.

The situation, in which a SHARE is expected to retransmit a frame, occurs if set of nodes *E* (with at least one element) detects an error during the bit #6 of EOF. These nodes will start an error frame at bit #7, which is observed by all nodes including the SHARE. The sender is now obliged to retransmit. To prevent an inconsistent message omission in case of a sender crash, the SHARE is expected to transmit the message simultaneously.

We observe, that the bit pattern, which is generated by this error is unique, i.e. it cannot be generated by some single error at some other section of the message frame. If some node detects an error in bit #6 the error is flagged in bit#7 of the EOF by sending a dominant bit value. Thus, the characteristic sequence is 7 recessive bits followed by a dominant bit. This sequence cannot occur during normal message transfer or error processing because of the bit stuffing rule and subsequent error signaling. Thus, a SHARE can easily be implemented by a state machine, which detects this particular sequence. Note, that at this time, the CAN-Controller of the SHARE has a valid message frame in his receive buffer.

Finally, the situation may occur that a node detects an error during bit #7. This node will be the sender or a SHARE because all receivers already have accepted the message. If any SHARE detects the message during bit #7, then it will retransmit. In this situation, a SHARE exactly behaves like any sender which detects an error in bit #7. Hence, a sender crash is always masked by a SHARE.

The actions of a SHARE are time critical. In order to start the frame retransmission simultaneously with the sender; a SHARE must be ready to retransmit a frame immediately after receiving it. To be more precise, after detecting an error at the last bit of a frame, a SHARE must start its retransmission within 3 bit-times. At the maximum bit-rate of 1Mbit/s, a SHARE has no more than 3 µ-seconds to start the retransmission of the last received frame. Currently, SHARE components are realized by dedicated hardware. The main component of a SHARE is a 16-bit Siemens microcontroller C167 [12], which has an embedded CAN controller and a special fast interrupt handling unit. This unit, called PEC (Peripheral Event Controller) enables the transfer of a word or byte from a source address to a destination address with minimal CPU interaction. SHARE uses 3 PEC transfer channels to:

1) convert the CAN receive buffer of the last received frame to a transmit buffer,
2) initiate its transmission, and
3) prepare the next free buffer for receiving another frame.

A PEC transfer only takes 2 processor clock cycles. Because the processor interrupt response delay (due to the pipeline) takes no more than 7 processor bus access cycles [12], the necessary steps (1-3) for retransmission are completed in less than 2 µs at a processor clock rate of 20MHz. Hence, a SHARE is fast enough for operating at the highest bit-rate of 1Mbit/s.

The error detection including the special treatment of bit #6 is realized by a relatively simple 22v10 Gate-Array Logic (GAL) which is directly connected to the CAN transceivers. The logic detects a characteristic sequence on the bus indicating that a dominant level occurred in bit #7 of the EOF. The error detector immediately raises an interrupt, which activates the PEC transfer channels. A listing of the GAL map can be found in [13].

The main advantage of dedicated SHARE components is their transparency. Existing CAN networks may add a SHARE to the bus in order to guarantee consistent frame delivery in the case of an inconsistent omission failure caused by a sender crash. No additional protocol overhead is necessary in fault-free situations. Even when an omission occurs, there is no additional effort to detect and handle the situation. A SHARE realizes a fault-tolerant sender, which retransmit a message with very high reliability. This is important when considering the overall bus scheduling which needs not to distinguish between consistent and inconsistent omission failures.

4 Achieving Order

With the introduction of SHARE components we solved the problem of inconsistent omission failures. We now can safely assume that each message which is received by a node will eventually be received by all operational nodes in the system even if the sender crashes and fails to retransmit. This is summarized in assumption (**A1): I**f one non-faulty node receives a message, then all non-faulty nodes eventually receive this message.

Our scheme to establish order between messages assumes that in a real-time control system, a message has to be delivered at its deadline. We further assume the co-existence of critical hard real-time messages and less critical soft real-time messages. The former have to meet their deadlines otherwise the system fails unpredictably. Soft real-time messages may miss their deadlines under transient overload. Temporal constraints like deadlines or laxities can be converted into a priority scheme [8, 14, 22], which is used by the CAN priority-based message dispatching mechanism to globally schedule the messages. We use the eight most significant bits of the CAN-ID as an explicit priority field. In [14, 15] we develop a scheme which is a combination of a reservation-based, time triggered message scheduling for hard real-time communication requirements and least laxity first (LLF) for soft real-time messages. Because the detailed description of the mechanism is beyond the scope of this paper, we will only roughly sketch the basic ideas of the scheme. Let us start with the assumption for hard real-time messages (**A2):** Hard real-time messages are received before their deadlines by all operational nodes under an omission fault assumption.

For hard real-time messages fixed time slots are reserved like in a TDMA [16] or TTP [17] approach. Conflicting resource requirements between hard real-time messages are resolved off-line, i.e. at run-time two hard real-time messages never will compete for the CAN-bus. If a time slot for a specific hard real-time message arrives, the dynamic priority mechanism assures that the message has the highest priority of all messages currently competing for the bus. To cope with communication failures, our scheme provides time redundancy and multiple transmissions of the same message. If k omissions must be tolerated for a message needing the time Δt to be transmitted, the respective reserved time slot must be of length: $k * (\Delta t + \Delta r) + \Delta t$ where Δr is the recovery time (a more detailed discussion can be found in [15]). Because a SHARE handles inconsistent omissions (assumption A1) we can assume that A2 holds.

Soft real-time messages are scheduled according to the LLF (Least Laxity First) algorithm [8]. This means that in overload situations deadlines may be missed (assumption **A3**). Finally, we assume that the communication system can rely on the fact that the local real-time operating system will provide a message timely, i.e. before the transmission deadline is expired (assumption **A4**).

Based on these assumptions, we now can develop an ordering scheme. The algorithm relies on the reliable message transfer and on the knowledge about the message transmission deadlines. The *transmission deadline (TD)* of a message denotes the point of time, at which the message must be successfully transmitted to all non-faulty destination nodes. The transmission deadline is tightly related to the point in time, where the message delivery to all receiving application objects is expected. Therefore, the ordering mechanism reflects an application specific order related to temporal requirements. We maintain the order for hard real-time messages and soft real-time messages in two totally ordered sets separately. This is because consensus about the order of soft real-time messages cannot be derived under hard real-time constraints. This property has formally been proved in [20].

Let *m* and *m'* be two real-time messages with deadlines d_m and $d_{m'}$. Then the deadline-based ordering algorithm implements the following rule:

$$\textbf{(O1):}\quad dm < dm' \Rightarrow delj(m) \rightarrow delj(m')$$

This means that if the deadline of the message *m* is before the deadline of the message *m'*, then *m* must be delivered to destination objects before *m'*. For hard real-time messages this is easy to show. Because hard real-time messages are scheduled off-line according to a time-triggered scheme and are transmitted in reserved time slots, the above rule follows immediately. The order between hard real-time messages can straightforwardly be formulated by the following delivery rule: Every hard real-time message is delivered at its deadline. Total order of hard real-time messages is guaranteed according to their deadlines.

To establish a total order between soft real-time messages is more complicated. Firstly, soft real-time messages may have identical transmission deadlines. This results in the same laxity that is mapped on the same priority. Secondly, under transient overload, multiple messages that already have missed their deadlines end up with the same laxity. In both cases we cannot use the priority field (8 most significant bits) in the CAN message ID alone to establish the order. However, because CAN IDs all must be different (to enable an unambiguous arbitration decision), some (arbitrary) order in which the messages are scheduled on the bus is determined by the unique less significant bits of the CAN-ID. Note, that in case of equal deadlines and in the situation that all laxities are "0", no relationship between the messages can exist because none of the messages has yet been sent. The following rule is used to ensure a global decision on the delivery order of messages *m* and *m'*.

$$\textbf{(O2):}\quad dm = dm' \Rightarrow (delj(m) \rightarrow delj(m') \Leftrightarrow message\text{-}IDm < message\text{-}IDm')$$

In *O2*, $message\text{-}ID_m$ consists of a tuple (d_m , $sender\text{-}ID_m$, r_m) with $sender\text{-}ID_m$ is some unique number for the sender and " r_m" are the remaining bits of the arbitration field. The $sender\text{-}ID_m$ makes sure that messages from different nodes competing for the bus have different CAN-IDs.

Soft real-time messages can become ready for transmission at any point in time. No a priori reservation or conflict resolution is performed. We may have messages with different and equal priorities. The ordering is based on the following properties:

P1: If a node has received a message *m,* and then another message with lower priority is observed on the CAN bus, or the bus is idle, then the sender of *m* will not retransmit it in future.

Assume that a message *m* is transmitted at least once on the CAN bus. Further, assume that the sending CAN controller still attempts to retransmit due to the inconsistent transmission. According to the CAN specification [1] the sender will try to retransmit *m* immediately, thus no bus idle period will be observed before the retransmission of *m*. Furthermore, no lower-priority message will be transmitted before the retransmission of *m*, because *m* will win the arbitration process against any lower-priority message. Because we assume reliable retransmission through a SHARE even in the case of sender crashes, we can enforce this property:

P2: Any receiver of a soft real-time message *m* can assume that the message has been received by all nodes when the receiver observes an idle bus or a message with a lower priority on the bus after receiving *m*.

P2 states the condition under which a receiver can detect that all nodes received a particular message. Now, to deliver this message, the receiver must be sure that it is delivered in the same order in all nodes. This order should be based on the transmission deadline. Therefore, we now must first clarify the question whether a message with an earlier deadline still can arrive. Based on our previous considerations we can conclude:

P3: A soft real-time message with a higher priority has an earlier deadline than a soft real-time message with a lower priority. If after the transmission deadline of a soft real-time message m another message with lower priority is transmitted on the CAN bus, or the bus is idle, then there is no other soft real-time message m' with an earlier or equal deadline, which is pending for transmission.

P3 is proved by the following considerations: Assume that a message *m'* with the deadline $d_{m'} < d_m$ is pending for transmission at the time $t > d_m$. Because of the deadline-based priority assignment the priority of *m'* is not lower than the priority of *m*. Due to assumption A4, *m'* has been pending for transmission at least since $d_{m'}$, hence at least since d_m. Hence the sender of *m'* must have been trying to transmit *m'* at the beginning of every bus-idle period since d_m. Therefore no idle bus can be observed before the successful transmission of *m'*. Also, no lower-priority message transmission can be observed on the bus before the successful transmission of *m'*, because *m'* would win the arbitration process against any lower-priority message. Thus, if after d_m a message with a lower priority is transmitted on the CAN bus, or the bus is unused for some period of time, then *m'* has been already transmitted successfully before and hence has been received before *m*.

We now can derive the property which enables a total order between soft real-time messages:

P4: For any soft real-time message *m*, no preceding soft real-time message will arrive later, if after the transmission deadline d_m a lower-priority message is observed on the bus.

Therefore, the delivery rule which maintains total order of soft real-time messages can finally be established as follows:

SRTD**:** *(Soft Real-Time message Delivery)* Every received soft real-time message can be delivered in total order as soon as either another message with later deadline, or a bus idle time is observed after its transmission deadline.

In situations which are not affected by overload, soft real-time messages are transmitted and delivered in strict deadline order. If there is no idle bus detected (in this case an immediate delivery is performed) the message has at most to wait for one message transmission time to be delivered. In overload situations, no temporal guarantees can be granted. The deadlines cannot be used for ordering the messages. However, total order between soft real-time messages is still preserved. Duplication of messages is handled by a sequence number.

5 Discussion and Related Work

We presented a scheme to provide reliable atomic broadcast for CAN. Considering the broadcast mechanisms specifically developed for CAN, we examined some of the popular application level protocols available for industrial automation [9, 10, 11]. These protocols provide reliable message delivery for situations, which are covered by the standard CAN error detection and fault handling mechanisms. Thus, they fail to provide measures for inconsistent omissions. Additionally, they cannot guarantee the same order of messages in all nodes even under moderate fault assumptions. This is because message duplicates cannot be safely distinguished from new messages.

Reliable message transmission to all operational receivers can be achieved by appropriate retransmission mechanisms [2, 6, 18, 21]. A problem occurs when a sender crashes before it performs the retransmission. All approaches to handle this situation are based on the assumption that the sender has correctly transmitted the message at least once and at least one receiver has correctly received it. This receiver then can retransmit the message. In an eager approach, the receivers try to retransmit the message until they have observed a certain number of retransmissions [2, 6]. In the lazy approach the message is retransmitted only if a sender crash has been detected. It is obvious that the first approach incurs additional message overhead while the second approach relies on the detection of a sender crash that requires a time-out mechanism. What distinguishes our approach is the use of masking redundancy. With the introduction of SHARE components, the retransmission in case of a sender crash becomes completely transparent. No additional overhead is added to the protocol that would increase network load or affect the basic timing properties.

Concerning the mechanism to establish order, our scheme accommodates hard and soft real-time messages. We maintain order separately for the two classes of real-time messages. This is because the delivery of a hard real-time message should not rely on the successfully established order between soft real-time messages, which in overload situations is not time-bounded. To our knowledge, the only work specifically addressing ordering schemes for CAN is described in [2] and [19]. In [19] a CAD-tool is used to determine off-line, which messages in an application may be causally related. The deadlines of related messages are adjusted according to this analysis. Then, their dynamic priority scheme [22] is used to schedule the messages on the CAN-bus according to their fixed deadlines. The approach is only valid in systems in which all causal relationships can be determined off-line. Secondly, the approach treats all messages as hard real-time messages.

In [2] a protocol suite is presented which provides fault-tolerant broadcast in CAN under varying system assumptions. The protocols are referred as EDCAN (CAN Eager-Diffusion), RELCAN (Lazy Diffusion-based Protocol), and TOTCAN (Totally Ordered CAN-protocol). The EDCAN (CAN Eager-Diffusion) protocol is based on a multiple transmission policy similar to [6]. The problem with EDCAN is the high communication overhead of about 200% under normal operation. To overcome this efficiency problem, the authors propose a more efficient protocol called RELCAN. In this protocol, efficiency in the fault-free case is traded against a certain waiting time in case of a sender crash. Although this delay is predictable, it has to be considered in the worst case estimation of every message transfer. Finally the TOTCAN protocol is proposed which provides total order of messages. TOTCAN uses a two-phase scheme. Sender crashes are not tolerated by TOTCAN, although consistency of messages is maintained b y discarding a message for which it cannot be guaranteed that it has been received by all nodes. In contrast to our hardware-based solution to reliable message transfer, the software-oriented protocols require a substantial overhead. This overhead involves not only the communication system, which has to carry a higher load, but also the processing nodes, which explicitly have to perform retransmissions.

6 Conclusion

The paper presented mechanisms to achieve fault-tolerant broadcasts in CAN. The mechanisms for error detection and fault handling in CAN already cover a considerable fault class. However, there still exist situations in which CAN alone is not able to guarantee the all-or-nothing property of message reception and the consistent order of messages. We first introduced a hardware extension SHARE to provide atomicity of message reception. A SHARE component detects the specific situation where some nodes may have accepted a message while others have not. In this situation a retransmission is necessary. In combination with the mechanisms provided by CAN, SHARE components mask sender crashes. The hardware of a SHARE is simple and can be added to any existing CAN-bus network. Multiple SHARE components can be added to increase reliability. Based on the atomicity property, we develop a deadline-based consistent ordering mechanism. We treat hard real-time messages and soft real-time messages separately. Hard real-time messages are strictly ordered according to their deadlines: This is possible because all deadlines of hard real-time messages are different and conflicts are resolved off-line. In contrast, soft real-time messages, which are created dynamically, may have identical deadlines. Moreover, soft real-time messages may miss their deadlines under transient overload. By carefully exploiting the properties of CAN, we provide total order for hard real-time and soft real-time messages, respectively, without additional message overhead. By treating hard and soft real-time messages separately, we make sure that a hard real-time message never is delayed because it has to wait on the decision about the order of a soft real-time message.

Currently the protocol is implemented and we investigate the integration with an application level protocol using the publisher/subscriber communication model [23]. For the publisher/subscriber model which is based on anonymous communication, it is highly beneficial that the protocols devised in this paper can achieve atomicity and order without explicit knowledge of the sender and receivers of messages.

7 References

[1] ROBERT BOSCH GmbH: „CAN Specification Version 2.0“, Sep. 1991.

[2] J. Rufino, P. Veríssimo, C. Almeida , L. Rodrigues: „Fault-Tolerant Broadcasts in CAN“, Proc. FTCS-28, Munich, Germany, June 1998.

[3] K.P. Birman and T.A. Joseph: "Reliable Communication in the Presence of Failures", ACM Tr. Computer Systems, 5(1):47-76, Feb. 1987.

[4] J.M. Chang and N.F. Maxemchuk: „Reliable broadcast protocols“, ACM Trans. on Computer Systems, 2(3), Aug. 1984, pp. 251-273.

[5] Weijia Jia, J. Kaiser, E. Nett:RMP: “Fault-Tolerant Group Communication”, IEEE Micro, IEEE Computer Society Press, Los Alamitos, USAS. 59-67, April 1996

[6] F. Cristian: „Synchronous Atomic Broadcast for Redundant Broadcast Channels“, The Journal of Real-Time Systems, Vol. 2, pp. 195-212, 1990.

[7] L. Rodrigues and P. Veríssimo: „xAMP: a Multi-primitive Group Communication Service“, IEEE Proc. 11th Symposium on Reliable Distributed Systems, Houston TX, Oct. 1992.

[8] C.M. Krishna, K.G. Shin: „Real-Time Systems“, McGraw-Hill, 1997

[9] CiA Draft Standards 201..207: „CAN Application Layer (CAL) for Industrial Applications“, may 1993.

[10] DeviceNet Specification 2.0 Vol. 1, Published by ODVA, 8222 Wiles Road - Suite 287 - Coral Springs, FL 33067 USA.

[11] Smart Distributed Systems, Application Layer Protocol Version 2, Honeywell Inc, Micro Switch Specification GS 052 103 Issue 3, USA, 1996

[12] Siemens AG: „C167 User’s Manual 03.96“, Published by Siemens AG, Bereich Halbleiter, Marketing-Kommunikation, 1996.
[13] M.A. Livani:”SHARE: A Transparent Mechanism for Reliable Broadcast Delivery in CAN”, Informatik Bericht 98-14, University of Ulm, 1998
[14] M.A. Livani, J. Kaiser, W. Jia: „Scheduling Hard and Soft Real-Time Communication in the Controller Area Network (CAN)“, 23rd IFAC/IFIP Workshop on Real Time Programming, Shantou, China, June 1998.
[15] M.A. Livani and J. Kaiser: „Evaluation of a Hybrid Real-time Bus Scheduling Mechanism for CAN“, 7th Int’l Workshop on Parallel and Distributed Real-Time Systems (WPDRTS’99), San Juan, Puerto Rico, Apr. 1999.
[16] Maruti 3, Design Overview 1st Edition, System Design and Analysis Group, Dept. of Comp. Science, Univ. of Maryland, 1995.
[17] H. Kopetz and G. Grünsteidl: „TTP - A Time-Triggered Protocol for Fault-Tolerant Real-Time Systems“, Res. Report 12/92, Inst. f. Techn. Informatik, Tech. Univ. of Vienna, 1992.
[18] F. Cristian et. al.: „Atomic Broadcast: From Simple Message Diffusion to Byzantine Agreement“, IEEE 15th Int'l Symposium on Fault-Tolerant Computing Systems, Ann Arbor, Michigan, 1985.
[19] K. M. Zuberi and K. G. Shin: „A Causal Message Ordering Scheme for Distributed Embedded Real-Time Systems“, Proc. Symp. on Reliable and Distributed Systems, Oct 1996.
[20] M.A. Livani, J. Kaiser:”A Total Ordering Scheme for Real-Time Multicasts in CAN”, The Proc. Third International Workshop on Active and Real-Time Database Systems, Schloß Dagstuhl, Mai 1999
[21] P. Ramanathan and K.G. Shin: "Delivery of Time-Critical Messages Using a Multiple Copy Approach", ACM Tr. Computer Systems, 10(2):144-166, May 1992.
[22] K. M. Zuberi and K. G. Shin, „Non-Preemptive Scheduling of messages on Controller Area Network for Real-Time Control Applications“, Technical Report, University of Michigan, 1995.
[23] J. Kaiser, M. Mock : "Implementing the Real-Time Publisher/Subscriber Model on the Controller Area Network (CAN)", 2nd Int’l Symposium on Object-Oriented Distributed Real-Time Computing Systems, San Malo, May 1999.